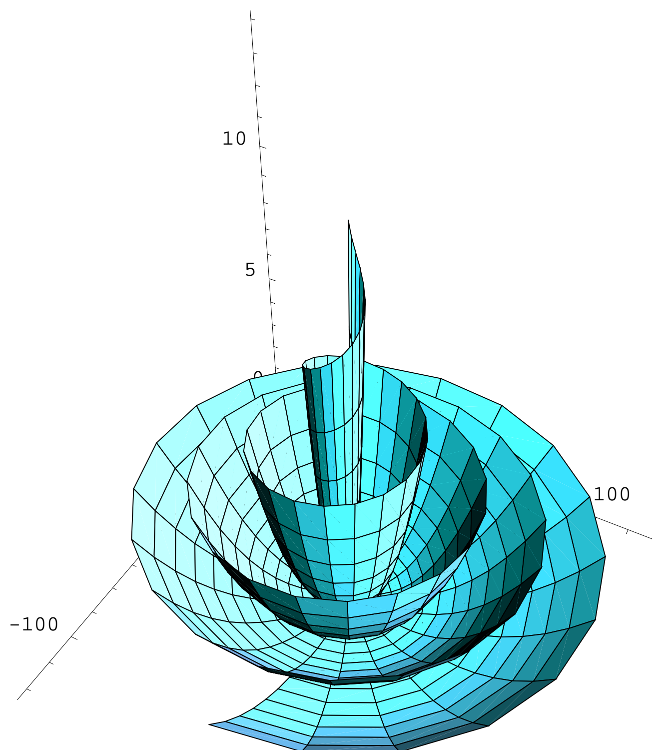
Appunti di Mathematica



Daniele Lupo

Studente di Ingegneria Elettronica (speriamo ancora per poco...)

Università degli studi di Palermo

danwolf80@libero.it

Appunti riguardanti il programma *Mathematica*. Spero vivamente che possano esservi utili. Contattatemi se trovate errori di qualsiasi genere, oppure avete commenti o suggerimenti da darmi.

Sommario

- *Introduzione*……1
  - ⇒ **Perché Mathematica?**……2
  - ⇒ **Suggerimento**……5
- *Primi Passi*……6
  - ⇒ **Cominciamo**……6
  - ⇒ **Numeri e Precisione**……10
  - ⇒ **Liste**……27
    - Creazione……27
    - Ricerca……37
    - Manipolazione……42
  - ⇒ **Vettori e Matrici**……45
  - ⇒ **Nota sulla Visualizzazione delle formule**……49
- *Calcolo Simbolico 1*……51
  - ⇒ **Introduzione**……51
  - ⇒ **Manipolazioni Algebriche**……53
  - ⇒ **Manipolazioni Avanzate**……64
  - ⇒ **Altre Manipolazioni**……70
  - ⇒ **Nota per Dimensioni Fisiche**……75
- *Calcolo Simbolico 2*……77
  - ⇒ **Introduzione**……77
  - ⇒ **Funzioni**……77
  - ⇒ **Sommatorie e Produttorie**……84
  - ⇒ **Equazioni**……85
    - Scrittura di equazioni……85
    - Disequazioni……86
  - ⇒ **Risolvere le Equazioni**……88
    - Equazioni Semplici……88
    - Sistemi di Equazioni……94
    - Disequazioni……98
    - Equazioni Differenziali……100
  - ⇒ **Serie di Potenze**……102
  - ⇒ **Limiti**……103
  - ⇒ **Trasformate Integrali**……106
- *Calcolo Numerico*……107
  - ⇒ **Introduzione**……107
  - ⇒ **Sommatorie, Produttorie, Integrali**……108
  - ⇒ **Soluzione di Equazioni**……109
  - ⇒ **Equazioni Differenziali**……111

⇒ **Ottimizzazione Numerica**……………………………………………113
⇒ **Dati Numerici**………………………………………………………...116
⇒ **Precisione**…………………………………………………………...118
❖ *Grafica*…………………………………………………………………..122
⇒ **Introduzione**……………………………………………………….....122
⇒ **Funzione ad una Variabile**……………………………………………122
⇒ **2D Generale**…………………………………………………………..134
⇒ **Funzioni a due Variabili**……………………………………………...148
⇒ **Introduzione al 3D**…………………………………………………...152
⇒ **Visualizzazione Dati**…………………………………………………..162
⇒ **Grafici Parametrici**……………………………………………………166
⇒ **Suoni**…………………………………………………………………177
❖ *Importazione ed Esportazione*…………………………………………..180
⇒ **Introduzione**…………………………………………………………..180
⇒ **Importazione**…………………………………………………………180
⇒ **Esportazione**………………………………………………………….187
- Dati……………………………………………………………………187
- Formule………………………………………………………………..189

⇒ **Packages**……………………………………………………………..192
- Introduzione……………………………………………………………192
- Salvataggio e Caricamento……………………………………………193

❖ *Programmazione*………………………………………………………..200
⇒ **Introduzione**…………………………………………………………..200
⇒ **Comandi Base**………………………………………………………...200
- Inizio…………………………………………………………………...200
- Input ed Output………………………………………………………...202
- Flusso di Programma…………………………………………………...207
- Stringhe………………………………………………………………...217

❖ *Formattazione*…………………………………………………………..225
⇒ **Introduzione**…………………………………………………………..225
⇒ **Celle**…………………………………………………………………..225
⇒ **Sezioni**………………………………………………………………..227
❖ *Mathematica Avanzata*…………………………………………………231
⇒ **Introduzione**…………………………………………………………..231
⇒ **Espressioni**……………………………………………………………231
- Cosa Sono……………………………………………………………..231
- Valutazione delle Espressioni…………………………………….….236

⇒ **Pattern**………………………………………………………………..241
- Cosa Sono……………………………………………………………..241
- Ricerca di Pattern, e Modifica………………………………………..…246

⇒ **Operazioni su Funzioni**………………………………………………262
- Introduzione…………………………………………………………...262
- Manipolazioni Base…………………………………………………....262
- Modifica Strutturale delle Espressioni………………………………...277
- Funzioni Pure…………………………………………………………289

- Definizioni di Espressioni e Funzioni……292
- Regole di Trasformazione……306

⇒ **Programmazione Avanzata**……314
- Introduzione……314
- Visibilità di Variabili Locali……314

❖ ***Appendice A: Packages***……320

⇒ **Algebra`AlgebraicInequalities`**……320
⇒ **Algebra`ReIm`**……322
⇒ **Algebra`RootIsolation`**……323
⇒ **Calculus`FourierTransform`**……325
⇒ **Calculus`VectorAnalysis`**……333
⇒ **DiscreteMath`Combinatorica`**……344
⇒ **DiscreteMath`GraphPlot`**……363
⇒ **Graphics`Animation`**……370
⇒ **Graphics`ComplexMap`**……374
⇒ **Graphics`FilledPlot`**……378
⇒ **Graphics`Graphics`**……384
⇒ **Graphics`Graphics3D`**……400
⇒ **Graphics`ImplicitPlot`**……416
⇒ **Graphics`InequalityGraphics`**……419
⇒ **Graphics`Legend`**……426
⇒ **Graphics`PlotField`**……431
⇒ **Graphics`PlotField3D`**……440
⇒ **Graphics`SurfaceOfRevolution`**……444
⇒ **LinearAlgebra`FourierTrig`**……448
⇒ **LinearAlgebra`MatrixManipulation`**……449
⇒ **LinearAlgebra`Orthogonalization`**……463
⇒ **Miscellaneous`Units`**……467
⇒ **NumberTheory`Recognize`**……473

❖ ***Appendice B: Eq. Differenziali***……476

⇒ **Introduzione**……476
⇒ **Tipi di Equazioni**……476
⇒ **Risoluzione Simbolica**……477
- DSolve……477

⇒ **ODE**……484
⇒ **Sistemi ODE**……499
⇒ **PDE**……505
⇒ **DAE**……516
⇒ **Problema del Valore al Contorno**……522
⇒ **NDSolve**……536

Appunti di Mathematica

Introduzione

Benvenuti a tutti quanti. Questi appunti cercheranno, in qualche modo, di farvi capire qualche aspetto più approfondito riguardo il programma *Mathematica*: cercherò di farvelo conoscere ad un livello un tantinello più avanzato riguardo l'uso come semplice calcolatrice :-)

Purtroppo sono soltanto un semplice studentello, e ci sono un'infinità pazzesca di cose di questo programma che non conosco neach'io. Dopotutto, il programma è così vasto che credo sia impossibile comprenderlo tutto appieno se non si usa tutto il giorno, tutti i giorni... Cosa che io, almeno per il momento, evito di fare, dato che gli studi ancora non mi hanno tolto del tutto la vita sociale, e riesco ancora ad uscire. Tuttavia, saper usare questo programma semplificherà notevolmente la vita a tutti quanti, ve lo garantisco.

■ Perchè *Mathematica*?

Nella mia facoltà ci sono principalmente due scuole di pensiero riguardo il miglior programma di calcolo attualmente disponibile su computer: la maggioranza degli studenti e dei professori preferisce Matlab, mentre una minoranza preferisce *Mathematica*.

Io personalmente sono convinto che un confronto diretto fra i due programmi non si possa fare: sono entrambi ottimi, sono standard de facto, e le preferenze si devono più che altro a flussi di lavoro diversi, ed al tipo di risultati che si vogliono ottenere.

Una della cose che può scoraggiare un utente che si trova per la prima volta davanti ad un programma come *Mathematica*, è la sua interfaccia spartana: icone, comandi, sono quasi del tutto assenti, e si lavora scrivendo formule e trasformazioni a mano, come se si usasse il blocco note di Windows. Anche le formule, normalmente, si sccrivono in questo modo, dando un 'aspetto poco elegante a tutto quanto. Ci sono comandi ed icone che permettono di ottenere gli stessi risultati di una formula stampata con LaTeX, ma, una volta perse quelle due orette ad abituarsi al programma, tutto quanto procede molto più velocemente di quanto si riesca a fare che andare a cercare sempre la giusta icona col mouse.

Matlab, d'altro canto, è pure troppo user-friendly; intendiamoci, lo è quanto può esserlo un programma di matematrica e tecnica che ti fa utilizzare i più avanzati algoritmi e di creare schemi e formule fra le più complicate del mondo, ma è sempre dominato da icone e toolbox. All'avvio, sembra anch'esso abbastanza spartano: ci sono più finestre ed icone, è vero, ma se devi fare una somma devi sempre scrivere numeretti come se usassi il blocco note. Invece, basta trovare il pulsantino Start per accedere ad un mondo immenso fatto di finestre e di Toolbox già preconfezionati per quasi tutto; se ti serve una GUI per poter studiare il luogo delle radici di una funzione di trasferimento, c'è. Se vuoi pulsantini che ti creino filtri numerici, ci sono anche quelli. Tool per l'analisi dei segnali preconfezionati: ci sono anche quelli. In *Mathematica*, invece, queste cose non ci sono, o meglio, ci sono a livello testuale; deci sempre scrivere i comandi a mano etc. In realtà, dalla versione 4 sono spuntati comandi per poter creare interfacce grafiche con Java e, dalla versione 5.1, con il Framework .NET di Microsoft, ma sono aspetti parecchio avanzati che, se semplificano di molto la vita in fase di esecuzione, la complicano esponenzialmente in fase di progettazione, se non si sa programmare bene ad oggetti. Se volete qualcosa di aspetto gradevole con *Mathematica*, provate a cercare qualcosa di già fatto su Internet.

Da un punto di vista di facilità d'uso, quindi, Matlab è oggettivamente migliore. A mio avviso, però, è computazionalmente meno potente. Attenzione, però! Non dico che diano risultati diversi, oppure che sia un programma peggiore, ma che utilizzano delle filosofie completamente diverse. Matlab è un programma che esegue principalmente calcolo numerico. Dategli una matrice numerica, e vi da l'inversa numerica. Dategli due liste di valori numerici, e Matlab vi farà la convoluzione numerica in un attimo.

Mathematica, dal canto suo, ha un motore di calcolo che è prettamente simbolico. Macina calcoli numerici alla grande e, fra l'altro, permette anche di superare la precisione di macchina, permettendo di fare qualsiasi calcolo concepibile dalla mente umana ed anche oltre con una precisione arbitraria che potete decidere da soli. Volete calcolare l'area di un cerchio con una precisione di 31413531531 cifre decimali? *Mathematica* ve lo fa in un attimo... provate a farlo con Matlab!

Ma non è questo quello che volevo dire, anche se da questo si può intuire la maggior flesibilità del suo kernel. Parlavamo del calcolo simbolico; se, con matlab, provate a calcolarvi un integrale di questo tipo

$$\int \frac{1}{\sqrt{\log(x)}}\, dx$$

Matlab vi chiederebbe per forza il valore iniziale e finale, per poi calcolarvi numericamente l'integrale. Dato in pasto a *Mathematica*, invece, vi da direttamente la soluzione completa:

$$\sqrt{\pi}\ \mathrm{erfi}\left(\log^{\frac{1}{2}}(x)\right)$$

Le possibilità di calcolo già si fanno più interessanti, a questo punto. Senza andare a scomodare centri di ricerca e scienziati, quello che interessa a noi studenti è la forma simbolica che può essere usata per verificare (od anche risolvere direttamente), problemi dati per casa, dando risultati che possono essere facilemente verificati e scritti in bell'aspetto direttamente sul quadernone. Inoltre, i Toolbox di Matlab, nella loro forza hanno anche il loro limite: sono specifici. Se cominciate a lavorare su un sistema di controllo, difficilmente l'equazione trovate potete usarle in altri toolbox per eseguire altri calcoli che vi servono, ammesso che riusciate a farvi dare da Matlab come risultati formule e non numeri: con *Mathematica*, invece, potete direttamente utilizzare tutte le formule che avete in tutti i contesti che volete, fare quello che vi pare. In *Mathematica* avete il controllo assoluto. E scusate se è poco! Farete calcoli e risolverete cose che non credevate possibili, senza approssimazioni od altro.

Per farvi capire in maniera molto grossolana la differenza, con Matlab si fanno solo gli esercizi. Con *Mathematica* si fa la teoria, quella pesante e quella importante. Inoltre, si fanno anche gli esercizi con un'elasticità che ben pochi programmi possono permettersi, fissati nel loro rigido modo di pensare imposto dai programmatori.

Ma la potenza di *Mathematica* non si ferma qui. Effettivamente, la potenza del programma è in qualche modo infinita... Riuscirete a fare cose inimmaginabili, se sapete come usarlo, e se ci perdete un po' di tempo scoprirete che riuscite a fare qualsiasi cosa vogliate. *Mathematica* non permette solamente di fare calcoli. Il frontend, ovvero la parte di *Mathematica* che vi permette inviare gli input al kernel, e di ricevere il corrispondente output, permette di creare anche documentazione dall'aspetto professionale; potrete scrivere tesi, tesine, relazioni, esercitazioni e quant'altro con un

aspetto che l'Equation Writer di Word si può soltanto sognare. Con, oltretutto, il vantaggio che *Mathematica* vi permette anche di fare i calcoli al posto vostro!

L'importante è non sottovalutare la potenza di questo programma. Se volete soltanto qualcosa che vi permetta di fare calcoli, al posto di spendere poco più di 100€ per la versione studenti di *Mathematica*, compratevi una calcolatrice HP 49G+, che costa sulle 130 e vi permette di fare cose che nessun altra calcolatrice vi permette di fare, ma pur sempre nell'eseguire calcoli tutto sommato semplici, anche se di tutto rispetto. Inoltre, potete anche portarvela per gli esami... Se, invece, volete un potente strumento di calcolo che, chissà, magari potrà servirvi anche per dopo, allora siete nel posto giusto e, anche se scalfiremo solamente la superfice, quello che imparerete in questi pochi appunti vi permetterà di risolvere facilmente la quasi totalità dei problemi che affronterete nel corso dei vostri studi.

Ovviamente, anche *Mathematica* ha qualche limite, a mio avviso. Per esempio, mi dispace veramente che non ci sia qualcosa di analogo al Simulink di Matlab. Sicuramente, avere una rappresentazione grafica delle funzioni, linkabili a piacimento renderebbe più semplice il lavoro e ne mostrerebbe una rappesentazione, perchè no, anche più chiara ed elegante. Così come il fatto che manchi, per esempio, il syntax highlighting, cioè la colorazione del codice, permettendo di distinguere variabili, funzioni e così via, che, assieme magari ad un editor separato con qualche funzione in più, semplificherebbe molto la scrittura dei programmi, un poco come l'editor dei file m, sempre di Matlab (anche *Mathematica* ha la sua versione di file m, ma vedremo verso la fine a cosa servono).

Inoltre, all'inizio non sarà semplicissimo da usare, in quanto usa un interfaccia prettamente testuale: questo ha il vantaggio di non dover andare in giro con il mouse a cercare pulsantini ed amenità simili, come accade, per esempio, in Mathcad, ma allo stesso tempo, se non si conosce una funzione, bisogna per forza andare a cercarla nell'help, per vedere se è disponibile, mentre nel caso di interfacce grafiche, basta esplorare i menù ed i pulsanti fin quando non si trova. Se non c'è, pazienza. Comunque la scoperta di nuovi comandi è certamente più intuitiva per interfacce grafiche che non per quelle testuali, anche ammesso che, una volta impratichiti, l'interfaccia di *Mathematica* è snella e veloce, permettendovi di fare quello che fanno altri programmi in una frazione di tempo.

Quindi, se proprio *Mathematica* vi fa antipatia, niente vi impedisce di imparare un altro programma con cui avete più feeling: di certo, a livello di studenti, non useremo molto le funzioni avanzate, e quelle base sono comuni a tutti quanti i programmi.

Io personalmente ho scelto *Mathematica* perchè mi piace un sacco; il mio primo programma di calcolo scientifico è stato Mathcad 7, che avevo scoperto mentre ero al superiore, giocandoci un pochino a scuola. Dopo, quando andavo in giro per il Web alla ricerca di un buon programma per la creazione di frattali, sono venuto a conoscenza di *Mathematica*, ed le cose per me sono drasticamente cambiate (anche se per i frattali uso Ultrafractal: http://www.ultrafractal.com).

Durante i corsi universitari ho anche avuto a che fare con Matlab, e ho capito la differenza fra i due programmi: sinceramente, ho preferito di gran lunga l'approccio simbolico di *Mathematica*, che mi permetteva di trovare soluzioni a problemi teorici, piuttosto che Matlab, per il quale se non ho i dati da dargli in pancia, è in grado di fare ben poco (anche se all'inizio mi stavo affezionando al suo comando 'magic'). Certamente è grande e forse migliore per l'elaborazione dei dati (e chi fa Elaborazione numerica dei segnali ne sa qualcosa, vero Giovanni?), ma sa fare solo questo, oltre ad avere un'interfaccia che sì, da moltissimi strumenti, ma è anche vero che è molto frammentata, e per fare cose diverse bisogna aprire altri toolbox etc.

Qua si parla di *Mathematica* (naturalmente!!!), ma il mio consiglio comunque è di provarli entrambi, e magari anche qualche altro, come Mathcad oppure Maple. Se, dopo, mi darete ragione, allora sarò felice di riceve ringraziamenti ed assegni in bianco come ringraziamento per avervi scritto queste righe!!!

- **Suggerimento**

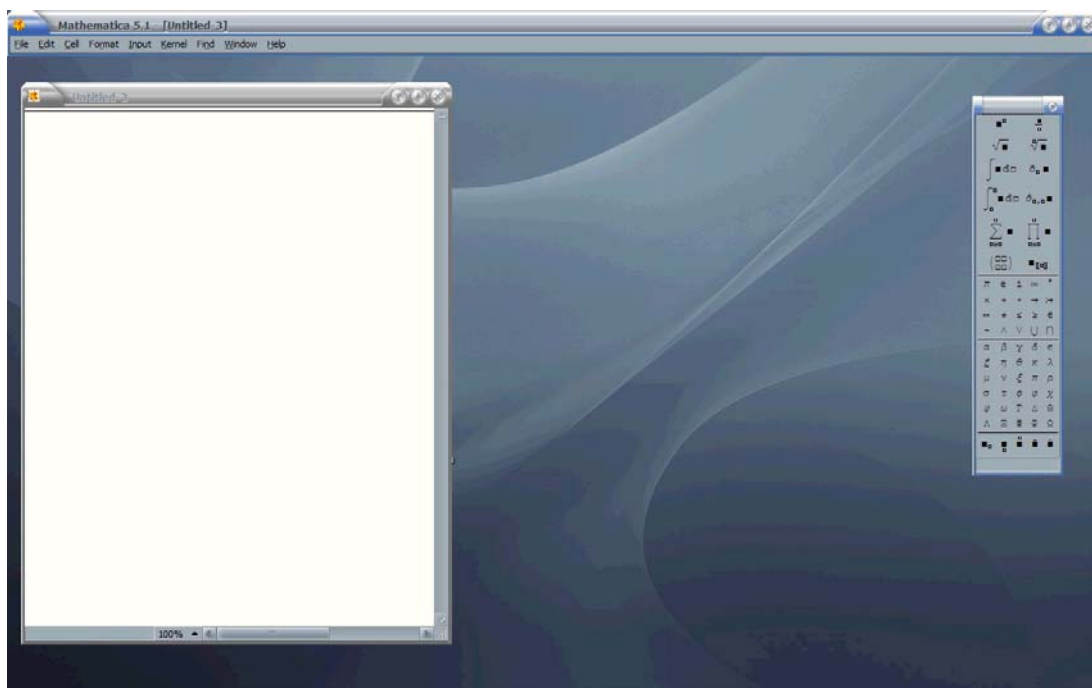
Il mio consiglio spassionato riguardo *Mathematica* è il seguente: USATE QUANTO PIU' SPESSO POTETE L'HELP IN LINEA!!! Scusate il tono brusco, ma è una cosa che veramente poche persone fanno; saper usare *Mathematica* è questione di poco, ma saperlo usare bene è difficile, per la quantità di funzioni e di aspetti e caratteristiche nascoste che non si trovano subito. Appena avete un problema con qualsiasi cosa, come una funzione che non ricordate o di cui non sapete il nome, andate a vedervi l'help, cercando la parola chiave di quello che state facendo o volete (in inglese, ovvio): se cercate funzioni per i numeri primi, cercate "prime", e così via. Quello che farò io è farvi capire il funzionamento di *Mathematica*, non quello di presentarvi ogni funzione presente: oltre che perdere trecento anni a descrivere le funzioni, sarebbe inutile, dato che tutto è già scritto nell'aiuto in linea. Probabilmente alcuni avranno difficoltà a destreggiarsi con la lingua inglese, ma non posso farci niente: dovete rendervi conto che l'inglese è una lingua fondamentale, specialmente per chi affronta temi scientifici oppure tecnici come noi. Di documentazione inglese se ne trova a bizzeffe, e ve la consiglio caldamente. Io non sono in grado di farvi capire fino in fondo *Mathematica:* Cerco solamente di darvi uno spunto, ed un punto d'inizio (anche se abbastanza solido) per cominciare ad usare questo programma. Con quello che imparerete qui saprete fare veramente un sacco di cose... Figuratevi se poi decidete di imparare il restante 99,9% di quello che il programma vi offre!

Primi passi

■ Cominciamo

Bene bene: dato che state leggendo, starete sicuramente cercando un modo per capire facilmente (ed in italiano), qualche base per poter cominciare ad usare questo simpatico programma.
Prima di tutto dovrei dirvi come installarlo, etc etc, ma credo che non abbiate bisogno di sapere queste banalità, vero? In fondo, siamo nati nell'era del computer!

Comunque, una volta avviato il programma per la prima volta, dovrebbe apparirvi una schermata simile alla seguente:



Effettivamente, non è che sia così esaltante. Comunque, quello che vedete a sinistra è il 'Notebook', cioè la finestra dove andrete a scrivere il vostro lavoro, mentre la finestra piccola a destra è una 'palette', ovvero un insieme di pulsanti che contengono comandi per semplificarvi il lavoro e per evitare di dover scrivere comandi; nel menù File->Palettes potete trovare altre palettes che magari riescono a semplificarvi il lavoro. Personalmente non la uso molto: trovo più veloce e semplice scrivere direttamente tutti i comandi a mano, anche se, devo ammetterlo, la uso ancora per inserire le lettere alcune greche, anche se ci sono opportune combinazioni da tastiera molto semplici per poter fare la stessa cosa.

Provate ad inserire, adesso, il seguente, complicatissimo comando ed osservate la potenza di *Mathematica* nel risolverla in un tempo inferiore alle due ore:

```
In[1]:= 2 + 2

Out[1]= 4
```

Magicamente, il risultato *Mathematica* ve lo da effettivamente in meno di due ore. notate come, per far eseguire il comando, dovete premere Shift+Invio oppure il tasto Invio del tastierino numerico; il tasto Invio normale serve solo per andare a capo senza eseguire operazioni, il che può sembrare strano, ma sicuramente utile non appena faremo qualcosa id più complicato della somma eseguita sopra. Intanto, si possono notare già da adesso alcune cose caratteristiche di *Mathematica*; se notate, alla sinistra delle espressioni potete notare i nomi In/Out, del tipo In[1]:= e Out[1]:=. queste rappresentano l'ordine con cui vengono eseguite le operazioni dal kernel. Infatti, mentre eseguite operazioni, potete, con il mouse, selezionare varie parti del Notebook e continuare a scrivere nel mezzo, come volete. Però eseguirete sempre le operazioni nell'ordine con cui sono state calcolate. considerate, per esempio, questo piccolo esempio:

```
In[5]:= 2 + 2

Out[5]= 4

In[8]:= m = 3

Out[8]= 3

In[6]:= m = 7

Out[6]= 7

In[7]:= m + 2

Out[7]= 9

In[9]:= m + 2

Out[9]= 5
```

Possiamo vedere già una cosa nuova, ma che credo non stupisca nessuna delle persone che leggano queste righe, ovvero la possibilità di poter dichiarare delle variabili. Le variabili possono essere qualsiasi cosa possibile, in *Mathematica*: da numeri, a simboli, formule, funzioni, liste, e chi più ne ha più ne metta, ma vedremo dopo queste cose. Per adesso vedete come sono combinati i risultati ed i numeri; dopo l'esempio iniziale, abbiamo dichiarato la variabile *m* inizializzandola al valore 3, e poi al valore 7: tuttavia, vedendo i tag a sinistra, si vede che l'inizializzazione al valore 7 viene prima: dopo viene In[7], che esegue l'espressione dando il risultato giusto. La prossima espressione che

viene eseguita è In[8], che riscrive il valore della variabile *m* e lo pone uguale a 3. Infine, rieseguendo l'espressione In[9], si vede che il risultato è esatto (ovviamente) con l'ultima inizializzazione di *m*. Inoltre, qualsiasi cosa appaia in un notebook è racchiuso in 'celle', come si vede alla destra. Le celle permettono di raggruppare il lavoro, e possono essere compresse ed espanse facendo doppio clic sulla linea corrispondente. Inotre, come si vede, possono essere anche annidate fra di loro. Comunque, servono solo per dare una particolare organizzazione al file, per cui non cene occuperemo, dato che *Mathematica* ha un buon gestore automatico di celle, che fa tutto da solo.

L'ordine è importante anche per un operatore importante in *Mathematica*, cioè l'operatore percentuale (%). In parole povere % rappresenta l'ultimo risultato ottenuto da *Mathematica*:

```
In[12]:= Sqrt[40]
Out[12]= 2 √10
In[13]:= % + 3
Out[13]= 3 + 2 √10
```

Qua possiamo vedere come l'operatore percentuale rappresenti l'ultimo risultato ottenuto. Questo è importante, perchè permette di usare velocemente un risultato senza riscriverlo e senza dover utilizzare una variabile per memorizzarla, e bisogna stare attenti ad utilizzarla quando ci si sposta a scrivere avanti ed indietro nel notebook. A questo si raggiungono altri risultati: per esempio %% si riferisce al penultimo risultato, %%% al terzultimo e così via; inoltre, %n indica il risultato Out[n]. Inoltre, potete anche vedere un'altra cosa MOLTO IMPORTANTE, che i principianti scordano facilmente e cadono facilmente in errore: quando si definisce e si usa una funzione in *Mathematica*, gli argomenti devono essere racchiusi entro parentesi QUADRE, non tonde. Le parentesi tonde sono utilizzate esclusivamente per raggruppare i termini di un'espressione, le parentesi quadre per determinare gli argomenti di una funzione, e (poi vedremo), le parentesi quadre per definire le liste e le matrici. Questo, sebbene in apparenza strano, ha un ben preciso significato. Infatti Nathematica permette manipolazioni 'simboliche' molto potenti, e permette quindi di usare variabili e funzioni non ancora inizializzate, lasciandole incognite; allora, se si scrive qualcosa come var$(3 + 5\,I)$, allora ci si troverebbe in difficoltà a capire se rappresenta la funzione var con il suo argomento, oppure la variabile var che moltiplica quello che è racchiuso fra parentesi. Ora capite meglio come mai si usano le parentesi quadre, vero? Inotre, *Mathematica* è case-sensitive, vuol dire che distingue fra maiuscole e minuscole: le funzioni predefinite (e, credetemi, ce ne sono un sacco), cominciano tutte con la lettera maiuscola, per convenzione, e per questo si scrive Sqrt invece che sqrt. Inoltre, quasi tutte le funzioni hanno il nome esteso, a parte qualche caso standard: infatti, dato il considerevole numero di funzioni, dare loro delle abbreviazioni renderebbe molto difficile ricordarsele. Si usano abbreviazioni sono per funzioni che le hanno standardizzate nel mondo matematico: quindi, il seno

si indicherà con Sin, invece che con Sinus, mentre, per fare un esempio, le funzioni di bessel si indicheranno con BesselJ[*n*, *q*], invece che con J, per esempio, perchè non è ancora convenzione internazionale usare soltanto la J. Questo ha anche un vantaggio; se conoscete il nome della funzione (in inglese), allora basta scriverla per intero, e *Mathematica* la userà direttamente senza alcun problema, perchè sicuramente l'avrà già dentro la pancia.

Qua sotto potete vedere una lista di alcune delle funzioni più comuni che si utilizzano di solito:

`Sqrt[x]`	radice quadrata ($\sqrt{x}$)
`Exp[x]`	esponenziale (e^x)
`Log[x]`	logaritmo naturale ($\log_e x$)
`Log[b, x]`	logaritmo in base b ($\log_b x$)
`Sin[x], Cos[x], Tan[x]`	funzioni trigonomentriche (con argomento in radianti)
`ArcSin[x], ArcCos[x], ArcTan[x]`	funzioni trigonometriche inverse
$n!$	fattoriale
`Abs[x]`	valore assoluto
`Round[x]`	intero più vicino ad x
`Mod[n, m]`	n modulo m (resto della divisione)
`Random[ ]`	numero pseudocasuale fra 0 ed 1
`Max[x, y, … ], Min[x, y, … ]`	massimo e minimo fra x, y, ...
`FactorInteger[n]`	scomposizione in fattori primi

mentre, qua in basso, sono indicate alcune funzioni che magari vedrete meno spesso, e che sono più specialistiche:

`Beta[a, b]`	funzione beta di Eulero $\mathrm{B}(a,b)$
`Beta[z, a, b]`	funzione beta incompleta $\mathrm{B}_z(a,b)$
`BetaRegularized[z, a, b]`	funzione beta incompleta regolarizzata $I(z,a,b)$
`Gamma[z]`	funzione Gamma di Eulero $\Gamma(z)$
`Gamma[a, z]`	funzione Gamma incompleta $\Gamma(a,z)$
`Gamma[a, z0, z1]`	funzione Gamma incompleta generalizzata $\Gamma(a,z_0)-\Gamma(a,z_1)$
`GammaRegularized[a, z]`	funzione Gamma incompleta regolarizzata $Q(a,z)$
`InverseBetaRegularized[s, a, b]`	funzione beta inversa
`InverseGammaRegularized[a, s]`	funzione Gamma inversa
`Pochhammer[a, n]`	simbolo di Pochhammer $(a)_n$
`PolyGamma[z]`	digamma function $\psi(z)$
`PolyGamma[n, z]`	n^{th} derivative of the digamma function $\psi^{(n)}(z)$

Naturalmente, le funzioni non finiscono certo qua!!! Quello vhe vi consiglio è di andare a cercare, di volta in volta, quello che vi serve dall'help in linea: per il numero di funzionalità e funzioni, vi posso assicurare che l'help di *Mathematica* è più importante di qualsiasi altro programma, credetemi.

■ Numeri e precisione

Quello che ci interessa, naturalmente, è trovare soluzioni a problemi. A volte, avendo a che fare con la matematica, avremo a che fare con i numeri, quindi... :-)

Mathematica cerca di riconoscere il tipo di numero con cui abbiamo a che fare, di volta in volta, utilizzando diversi algoritmi per ottimizzare velocità di calcolo e risultato in funzione dei dati iniziali. Per esempio possiamo distinguere fra numeri interi e complessi. I tipi di numeri predefiniti in *Mathematica* sono 4:

`Integer`	numeri interi esatti di lunghezza arbitraria
`Rational`	*integer* / *integer* ridotto ai minimi termini
`Real`	numero reale approssimato, con precisione qualsiasi
`Complex`	numero complesso nella forma *number* + *number* `I`

Sebbene sembri strano dover distinguere fra diversi tipi di numeri, dato che un numero è sempre tale, in realtà non è così; come quelli che fra di voi programmano ben sanno, di solito il computer gestisce in maniera diversa i diversi tipi di numeri, quando andiamo a scrivere un programma. Per questo, quando definiamo una variabile, dobbiamo specificare, in C come in Pascal come in Fortran, se il numero è intero oppure a virgola mobile: in un caso o nell'altro verranno utilizzate diverse istruzioni per eseguire lo stesso calcolo.

In *Mathematica* succede qualsosa di simile; sono presenti diversi algoritmi che ci permettono di ottimizzare il calcolo ed utilizzare la maniere più opportuna di trattare un numero. Inoltre, c'è in più il vantaggio che *Mathematica* fa tutto questo automaticamente: a noi basta semplicememte scrivere i numeri, ed il programma riconoscerà automaticamente il tipo di numero con cui abbiamo a che fare, utilizzando l'algoritmo opportuno.

Per esempio, possiamo eseguire un calcolo che sarebbe improponibile in precisione macchina:

In[2]:=
```
130 !
```

Out[2]=
```
6466855489220473672507304395536485253155359447828049608975952322944
 781961185526165512707047229268452925683969240398027149120740074042
 10584473774779945931002963578099177461298380315096514560000000000000
 00000000000000000000
```

Come potete vedere, *Mathematica* riconosce il numero come intero esatto, applicando quindi la precisione arbitraria. Se vogliamo un risultato approssimato, invece, dobbiamo far capire al programma che il numero usato come input è anch'esso approssimato:

```
In[3]:=  130.!

Out[3]=  6.46686×10^219
```

Mettendo il punto, *Mathematica* interpreta il numero non come esatto, ma come approssimato; di conseguenza effettua il calcolo con un algoritmo alternativo.

Inoltre, possiamo eseguire calcoli avanzati anche con i numeri approssimati. Infatti, un aspetto molto importante in *Mathematica*, è che permette di passare sopra uno dei limiti principali del calcolo al computer, ovvero la precisione di macchina. Quando andate ad eseguire dei calcoli, come degli integrali numerici, dovrete sempre stare attenti alla precisione. *Mathematica* permette di ottenere due tipi di precisione di calcolo:

1) La precisione standard di macchina. Quando scrivete delle espressioni in virgola mobile, *Mathematica* le tratta come numeri che hanno insita la precisione di macchina, per cui considera inutlie utilizzare una precisione maggiore:

```
In[4]:=  Sqrt[12.]

Out[4]=  3.4641
```

Tuttavia, se non si utilizza il punto, *Mathematica* considera il numero con una precisione assoluta, e utilizza la sua precisione, che può dare un risultato simbolico, oppure anche numerico, ma con una precisione impostabile a piacere. se riscriviamo l'espressione di sopra senza il punto otteniamo:

```
In[5]:=  Sqrt[12]
```

Out[5]= $2\sqrt{3}$

che rappresenta il risultato che scriveremmo nel quaderno. Tuttavia, possiamo anche avere un risultato numerico con precisione arbitraria, utilizzando la funzione N. questa funzione prende come argomento un'espressione che da un risultato esprimibile in numero (niente incognite, quindi), e fornisce il risultato in forma approssimata. Scrivendo

```
In[6]:=  N[Sqrt[12]]

Out[6]=  3.4641
```

oppure

```
In[7]:=  Sqrt[12] // N

Out[7]=  3.4641
```

il risultato che si ottiene è il seguente

```
3.4641
```

Il risultato è identico a prima, perchè, se non si specifica nella funzione N anche il numero di cifre significative che si desidera, allora usa la precisione di macchina (per i calcoli interni, dato che da un risultato con un minor numero di cifre, considerando solo quelle significative). Se proviamo anche a scrivere il numero di cifre significative richieste, *Mathematica* ci accontenta:

```
In[8]:=  N[Sqrt[12], 30]

Out[8]=  3.46410161513775458705489268301
```

Come vedete, il risultato da un numero di cifre significative maggiore di quelle raggiungibili con la precisione di macchina. Possiamo tranquillamente richiedere 10000 cifre significative, se vogliamo...

Mathematica utilizza la precisione arbitraria anche per le costanti. Per esempio, Pi definisce il pigreco (dato che la lettera greca è difficile da scrivere con la tastiera), E rappresenta il numero di Nepero. Se vogliamo una precisione di 100 cifre, per dirne una, possiamo scrivere semplicemente

```
In[9]:=  N[Pi, 100]

Out[9]=  3.141592653589793238462643383279502884197169399375105820974944592307
          8164062862089986280348253421170 68
```

La precisione arbitraria, abbiamo visto, si applica anche per espressioni non razionali, in quanto *Mathematica* tratta l'espressione simbolicamente, fornendo il risultato sotto forma facilmente leggibile:

```
In[10]:=  Sin[Pi / 3] Sqrt[5]
```

Out[10]= $\frac{\sqrt{15}}{2}$

Notate che la moltiplicazione può essere indicata anche, oltre al canonico asterisco, lasciando uno spazio fra i fattori, in questo caso fra le funzioni. Questi sono solo esempi banalissimi. Le cose possibili sono davvero uno sproposito.

Possiamo anche scegliere la precisione da utilizzare anche durante i calcoli. Se, per esempio

eseguiamo un'operazione con due numeri approssimati, otteniamo un risultato anch'esso approssimato:

```
In[11]:= 2. / 3.

Out[11]= 0.666667
```

Questo specifica la precisione standard utilizzata, pari a quella macchina. Tuttavia, se specifchiamo numeri approssimati con maggior precisione, *Mathematica* automaticamente esege il risultato non più in precisione macchina, ma adeguandola alla precisione dei numeri in ingresso:

```
In[12]:= 2.0000000000000000000000 / 3.00000000000000000000000000

Out[12]= 0.6666666666666666666667
```

Tuttavia, se la precisione dei numeri non è uguale, *Mathematica* esegue il calcolo restituendo un risultato con il giusto numero ci cifre significative:

```
In[13]:= 2.0000000000000000000000 / 3.

Out[13]= 0.666667
```

Adegua, in questo modo, la precisione per evitare risultati scorretti.

Inoltre, *Mathematica* tratta anche con estrema naturalezza i numeri complessi:

```
In[14]:= Sqrt[-4]

Out[14]= 2 ⅈ
```

```
In[15]:= ArcSin[13.54758759879480487048794]

Out[15]= 1.57079632679489661923132 - 3.29799076132040785635379 ⅈ
```

L'unità immaginaria si indica con la lettera maiuscola I, maiuscola mi raccomando: vi ricordi che, per convenzione, tutte le funzioni e costanti predefinite cominciano con una lettera maiuscola.

Alcune semplici funzioni che riguardano i numeri complessi sono quelle standard:

`x + I y`	il numero complesso $x+i\ y$
`Re[z]`	parte reale
`Im[z]`	parte immaginaria
`Conjugate[z]`	complesso coniugato z^* or $\bar{z}$
`Abs[z]`	valore assoluto $\lvert z \rvert$
`Arg[z]`	argomento φ in $\lvert z \rvert e^{i\varphi}$

Se poi abbiamo determinati valori, possiamo testare di che tipo siano:

`NumberQ[x]`	testa se x è una quantità numerica di qualsiasi tipo
`IntegerQ[x]`	testa se x è un numero intero
`EvenQ[x]`	testa se x è un numero pari
`OddQ[x]`	testa se x è un numero dispari
`PrimeQ[x]`	testa se x è un numero primo
`Head[x]===`*type*	testa il tipo di numero

I primi comandi servono per testare se un numero appartiene ad un determinato tipo; per esempio se è intero, oppure se è pari:

```
In[16]:= NumberQ[3]

Out[16]= True
```

```
In[17]:= NumberQ[a]

Out[17]= False
```

```
In[18]:= EvenQ[4]

Out[18]= True
```

```
In[19]:= EvenQ[4.]

Out[19]= False
```

L'ultimo risultato deriva dal fatto che, essendo un numero reale, *Mathematica* non è in grado di capire se è pari o dispari, perchè è approssimato. 4. potrebbe anche essere 4.00000000000000000001, che non sarebbe nè pari nè dispari, applicando questi due concetti solamente a numero interi.

Per verificare se un numero è, invece, razionale o qualt'altro, occorre utilizzare l'ultimo comando:

```
In[20]:= Head[4 / 7] === Rational

Out[20]= True
```

Head è un comando avanzato, che vedremo meglio più avanti... Qua serve solo per analizzare il tipo di numero che contiene. In questo caso testiamo se il suo argomento è un numero razionale, dandoci risposta affermativa. Possiamo anche eliminare il test, ed in questo caso il comando restituisce il tipo di numero che ha come argomento:

```
In[21]:= Head[3 + 5 I]

Out[21]= Complex
```

Vediamo adesso il seguente esempio:

```
In[22]:= NumberQ[π]

Out[22]= False
```

Qua qualcosa potrebbe andare per il verso sbagliato, perchè, anche se in forma simbolica, π rappresenta a tutti gli effetti un numero, mentre non viene riconosciuto come tale. Il fatto è che NumberQ testa se l'argomento è un numero esplicito, cioè se è scritto in forma di numero come 3.14, per intenderci. Rigirosamente, π non è un numero, ma rappresenta invece una quantità numerica:

NumberQ[*expr*]	testa se *expr* rappresenta in maniera esplicita un numero
NumericQ[*expr*]	testa se *expr* è un valore numerico

Per vedere se l'argomento, pur non essendo scritto sotto forma di numero, rappresenta una quantità numerica, dobbiamo utilizzare il secondo comando:

```
In[23]:= NumberQ[Sqrt[2]]

Out[23]= False
```

```
In[24]:= NumericQ[Sqrt[2]]

Out[24]= True
```

```
In[25]:= NumericQ[Sqrt[x]]

Out[25]= False
```

Vediamo che adesso il cerchio si chiude, e che possiamo testare effettivamente se un espressione rappresenta un numero, oppure non è definita.

Dato che la precisione dei risultati assume un ruolo importante, *Mathematica* pone la giusta importanza alle cifre significative di un numero; esistono anche delle funzioni che determinano la precisione oppure l'accuratezza di un numero:

Precision[*x*]	il numero totale di cifre significative di *x*
Accuracy[*x*]	il numero di cifre decimali significative di *x*

Quando si parla di precisione di un numero approssimato, bisogna sempre distinguere fra precisione ed accuratezza. Nel primo caso, infatti, si parla di numero totale delle cifre che compongono un numero. Nel secondo, invece, si considerano solamente il numero di cifre significative che compongono la parte decimale di un numero, ignorando le cifre che compongono la parte intera di un numero. Vediamo, per esempio:

```
In[26]:=  a = N[Sqrt[2] + 1987, 40]

Out[26]=  1988.414213562373095048801688724209698079
```

Vediamo il numero di cifre significative di questo numero:

```
In[27]:=  Precision[a]

Out[27]=  40.
```

Questo risultato l'abbiamo ottenuto proprio perchè lo abbiamo imposto dal secondo parametro di N. Vediamo adesso l'accuratezza di questo stesso numero:

```
In[28]:=  Accuracy[a]

Out[28]=  36.7015
```

Approssimando, abbiamo 36 cifre significative, il che significa che 36 cifre decimali sono significative; questo concorda con il risultato precedente, dato che la parte intera è composta da quattro cifre, e che $36 + 4 = 40$.

Questo permette di definire sempre l'incertezza su di un numero. D'altronde, per questo motivo, due numeri possono anche essere considerati uguali se la loro differenza è inferiore alla precisione oppure all'accuratezza dei due numeri.

Se un determinato valore numerico x ha un'incertezza pari a δ, allora il valore vero di x, cioè non approssimato, può essere uno qualunque nell'intervallo che va da $x - \delta/2$ a x+δ/2. Se l'accuratezza di un numero è pari ad a, allora l'incertezza legata a quel numero sarà data da 10^{-a}; mentre, se al posto dell'accuratezza abbiamo la precisione di un numero, definito come p, allora l'incertezza sarà

pari a $|x| 10^{-p}$.

Supponiamo di avere un numero con cinque cifre significative:

In[29]:=
```
x = N[Sqrt[2], 5]
```
Out[29]=
```
1.4142
```

Andiamo a sommarci, adesso, un numero minore della sua incertezza:

In[30]:=
```
b = N[1 / 10000000, 7]
```
Out[30]= 1.000000×10^{-7}

In[31]:=
```
x + b
```
Out[31]=
```
1.4142
```

Come possiamo vedere, il risultato non è variato, perchè sommo ad un numero un altro più piccolo della precisione del precedente. Se li sommassi, otterrei un risultato senza significato, dato che non so a cosa sto sommando il secondo numero: potrebbe essere un qualsiasi numeri dell'intervallo di incertezza. Se non si specifica niente, come precisione *Mathematica* utilizza quella di macchina del computer:

MachinePrecision	la precisione predefinita per un numero calcolato con precisione di macchina
$MachinePrecision	variabile di sistema che contiene il valore della precisione di macchina
MachineNumberQ[*x*]	testa se *x* è un numero macchina

Dato che il numero di cifre significatice può variare a seconda del computer e del sistema operativo (si pensi ai nuovi processori a 64 bit, per esempio), il valore della precisione di macchina può variare. Per questo *Mathematica*, invece di restituire la precisione di un numero macchina sotto forma di numero di cifre significative, lo restituisce con il simbolo MachinePrecision: specifica che il numero è un numero macchina e basta, restando coerente con la sua logica interna e non con l'archichettura del computer. La variabile di sistema $MachinePrecision, invece, contiene questo numero, che può variare da computer a computer:

In[32]:=
```
N[E]
```
Out[32]=
```
2.71828
```

```
In[33]:= Precision[N[E]]

Out[33]= MachinePrecision

In[34]:= $MachinePrecision

Out[34]= 15.9546
```

L'esempio di sopra mostra che N, senza argomenti, restituisce il valore approssimato in precisione di macchina e che su questo computer, in Athlon 64 con WindowsXP Home, la precisione di macchina è data dal valore riportato. Probabilmente, quando installerò un sistema operativo a 64 bit, questo valore varierà...

Notate, adesso, questo particolare:

```
In[35]:= Precision[4.3]

Out[35]= MachinePrecision

In[36]:= Precision[4.389609860986096971629763409128709870897]

Out[36]= 39.6424
```

Questa è una particolarità interessante; se il numero di cifre significative è superiore a quello di precisione di macchina, *Mathematica* lo memorizza come valore con la sua corretta precisione, mentre se il numero ha una precisione inferiore a quella macchina, il programma, invece di dargli la precisione che gli spetterebbe, gli assegna invece la precisione macchina.

Così, se andiamo a sommare due numeri con precisione diversa, ma inferiore a quella macchina, il risultato avrà sempre la stessa precisione:

```
In[37]:= a = 1.45

Out[37]= 1.45

In[38]:= Precision[a]

Out[38]= MachinePrecision

In[39]:= b = 1.4545

Out[39]= 1.4545
```

In[40]:= **Precision[b]**

Out[40]= MachinePrecision

In[41]:= **a + b**

Out[41]= 2.9045

In[42]:= **Precision[a + b]**

Out[42]= MachinePrecision

Come vedete, la somma considera tutte le cifre signigficative della macchina, sebbene abbiamo scritto il valore di *a* con un numero di cifre significative inferiore...

Vediamo adesso questo esempio:

In[43]:= **N[Sqrt[2]]**

Out[43]= 1.41421

E' sempre quello, va bene... tuttavia notiamo come sia rappresentato con un numero di cifre significative inferiore a quelle di macchina. Questo perchè, anche se *Mathematica* lo calcola con tutte le cifre, restituisce un risultato con cifre più significative. Se vogliamo vedere tutto il numero, dobbiamo utilizzare il comando InputForm;

In[44]:= **InputForm[N[Sqrt[2]]]**

Out[44]//InputForm= 1.4142135623730951

Questo comando visualizza il suo argomento come appare a *Mathematica* stesso, non come compare a noi nel notebook. Questo rappresenta un numero macchina. Visualizziamo invece il seguente esempio:

In[45]:= **InputForm[N[Sqrt[2], 50]]**

Out[45]//InputForm=
```
1.41421356237309504880168872420969807856967187\
537694807317667973799072`50.
```

In questo caso c'è qualcosa di diverso. Non solo viene visualizzato il numero per intero, ma gli viene pure attaccato un valore, che rappresenta la precisione di quel numero. Questo è il modo che *Mathematica* ha di riconoscere la precisione dei numeri. Purtroppo il simbolo ` nella tastiera italiana non esiste: per stamparlo dovete utiilzzare la combinazione Alt+096 con i numeri del tastierino numerico oppure, come faccio io quando mi serve parecchie volte, scriverlo solo questo carattere in

un notebook, e copiarlo ed incollarlo ogni volta che serve. Sebbene il primo metodo sia abbastanza veloce, io personalmente utilizzo il secondo perchè utilizzo il portatile, e non ho tastierino numerico... Un aiutino per chi come me si sbraitava per capire come fare!!!

In questa maniera possiamo specificare la precisione anche per quei numeri che DEVONO averla inferiore a quella macchina, perchè magari il risultato dell'esperimento non raggiunge quella precisione:

```
In[46]:= a = 1.01`3

Out[46]= 1.01

In[47]:= b = 0.0003`2

Out[47]= 0.00030

In[48]:= a + b

Out[48]= 1.01
```

Qua potete notare due cose: prima di tutto, che specificando le precisioni inferiori a quelle di macchina, il calcolo viene eseguito con la precisione apposita. Secondo, che abbiamo specificato *b* con una precisione minore delle sue cifre... Questo perchè la cifra, in quel caso, è solamente una, dato che la mantissa del numero è ad una sola cifra:

```
In[49]:= Precision[b]

Out[49]= 2.

In[50]:= Accuracy[b]

Out[50]= 5.52288
```

In questo caso, con Accuracy abbiamo visto il numero di cifre 'decimali' significative, e si ottiene calcolando il numero per intero, senza lasciarlo nella forma mantissa ed esponente. In questo caso il numero di cifre significative è 6, perchè è dato dalla cifra 3 più lo zero alla sua destra dato dalla precisione.

Analigamente a prima, possiamo anche scrivere dei numeri con le ultime cifre pari a 0, tenendo conto di quest'ultime se sono anch'esse cifre significative:

```
In[51]:= 3`40

Out[51]= 3.000000000000000000000000000000000000000
```

Un'altra maniera per definire il numero di cifre significative o di accuratezza di un numero, più consono alla nostra tastiera, p di usare i comandi opportuni:

SetPrecision[*x*, *n*]	crea un numeri con *n* cifre decimali di precisione, completandolo con degli 0 se risultasse necessario
SetAccuracy[*x*, *n*]	crea un numero con *n* cifre decimali di accuratezza

Si creano in questa maniera risultati analoghi ai precedenti:

```
In[52]:= SetPrecision[3, 40]

Out[52]= 3.000000000000000000000000000000000000000
```

```
In[53]:= SetAccuracy[342321.23, 7]

Out[53]= 342321.230000
```

Dato che abbiamo parlato di mantissa ed esponente, vediamo come possiamo scrivere i numeri in questa maniera. Possiamo utilizzare, naturalmente, la forma classica:

```
In[54]:= 2 * 10 ^ 5

Out[54]= 200000
```

Ma *Mathematica* permette una piccola scorciatoia per scrivere la stessa cosa:

```
In[55]:= 2*^5

Out[55]= 200000
```

Vediamo come viene applicata la precisione:

```
In[56]:= a = 3.1351364265764745873578*^34
```

Out[56]= $3.1351364265764745873578 \times 10^{34}$

```
In[57]:= InputForm[a]

Out[57]//InputForm= 3.1351364265764745873578`22.496256444056755*^\
34
```

Come possiamo vedere a conferma di quanto detto poco fa, la precisione viene applicata alla mantissa, non all'esponente del numero.

Notiamo anche come il simbolo ` definisce il numero di cifre di precisione. Se invece vogliamo definire il numero di cifre di accuratezza, dobbiamo utilizzare ``:

In[58]:= **124.134315`6**

Out[58]= 124.134

In[59]:= **a = 124.134315``6**

Out[59]= 124.13432

In[60]:= **InputForm[a]**

Out[60]//InputForm= 124.134315`8.09389185204773

Notiamo che le cifre di accuratezza sono sempre quelle, anche se capita che ne vengano visualizzate di meno per convenzione...

A volte, le piccole quantità sono dati da errori di approssimazione, e si vorrebbero eliminarli. Per esempio, capita che venga scritto 0.0000000000000000231 quando sappiamo che il risultato sicuramente sarà pari a 0. Per questo problema *Mathematica* ha un apposito comando:

Chop[*expr*]	sostituisce tutti i numeri reali in *expr* con modulo minore di 10^{-10} con 0
Chop[*expr*, *dx*]	sostituisce i numeri con modulo minore di *dx* con 0

Questo può essere utile quando cerchiamo, per esempio, numericamente delle radici di equazioni; in questo caso ci saranno numeri molto piccoli che possono essere eliminati:

In[61]:= **Chop[2 + Sqrt[0.000000000000000000000003]]**

Out[61]= 2.

Mathematica, quando esegue un'operazione, cerca di ottenere sempre il maggior numero di cifre significative, però compatibilmente con il calcolo da eseguire. Possono esserci calcoli che fanno perdere precisione ad un valore numerico per proprietà intrinseche, come sa chi ha fatto calcolo numerico:

In[62]:= **a = x = N[1 - 10^-30, 40]**

Out[62]= 0.9999999999999999999999999999990000000000

In[63]:= **1 + a**

Out[63]= 1.9999999999999999999999999999999000000000

In[64]:= **1. + a**

Out[64]= 2.

In[65]:= **b = (a ^ 2 - 1) / (3 a - 1)**

Out[65]= $-1.000000000\times10^{-30}$

In[66]:= **Precision[b]**

Out[66]= 10.

Come possiamo vedere, il valore dell'espressione ha una precisione minore, dovuto alla differenza $a^2 - 1$, che è quasi nulla, portando ad un grande errore di cancellazione.

Per poter ottenere risultati con la precisione massima possibile, *Mathematica* a volte deve aumentare forzatamente la precisione durante il processo dei suoi algoritmi interni. La variabile di sistema $MaxExtraPrecision rappresenta il limite massimo di aumento di precisione. A volte, però, può non bastare per ottenere un risultato con la precisione voluta:

In[67]:= **N[Tan[10 ^ 1000], 50]**

— *N::meprec : Internal precision limit $MaxExtraPrecision = 49.99999999999999` reached while evaluating Tan[100000000000000000000000000 <<945>> 000000000000000000000000000]. More…*

Out[67]= ComplexInfinity

In questo caso, bisogna provare ad aumentare il valore della variabile, per ottenere il risultato voluto:

In[68]:= **$MaxExtraPrecision = 2000**

Out[68]= 2000

In[69]:= **N[Tan[10 ^ 1000], 50]**

Out[69]= -0.86303668636289036146207322773061805888611871314800

In questo modo, durante il calcolo, abbiamo approssimato l'argomento della funzione con un numero di cifre significative sufficiente per ottenere il risultato con il numero di cifre significative richiesto, e questo è utile quando si vogliono molte cifre significative per un risultato che invoca operazioni

che diminuiscono la precisione del numero approssimato.

Per quanto riguarda l'utilizzo dell'estensione della precisione dei calcoli, va fatta con la dovuta accortezza. Il fatto che, quando definiamo numeri con precisione inferiore a quella macchina, vengano definiti con una precisione uguale a quella macchina, ha un suo perchè: infatti, con precisione standard *Mathematica* può utilizzare direttamente i comandi per il calcolo in virgola mobile del processore, aumentando l'efficienza del calcolo. Introducendo precisioni maggiori, *Mathematica* usa altri algoritmi che rallentano il calcolo, tenendo conto del fatto che, per fare una stessa operazione, occorrono più passaggi e cicli del processore per tenere in considerazione la precisione maggiore, dato che il processore calcola sempre numeri in precisione macchina. L'implementazione è software, con i rallentamenti del caso.

Per tener conto della precisione del computer su cui gira, *Mathematica* ha delle variabili di sistema che tengono conto di queste caratteristiche:

`$MachinePrecision`	il numero di cifre significative di precisione
`$MachineEpsilon`	il numero macchina più piccolo possibile che, sommato ad 1.0, restituisce un numero diverso da 1.0
`$MaxMachineNumber`	il numero macchina più grande rappresentabile
`$MinMachineNumber`	il numero macchina più piccolo rappresentabile
`$MaxNumber`	il modulo più grande di un numero macchina rappresentabile
`$MinNumber`	il modulo più piccolo di un numero macchina rappresentabile

Questi valori sono importanti se si vuole tenere in considerazione la precisione di macchina nei propri calcoli:

In[70]:= **`$MachinePrecision`**

Out[70]= `15.9546`

In[71]:= **`$MaxNumber`**

Out[71]= $1.920224672692357 \times 10^{646456887}$

In[72]:= **`$MachineEpsilon`**

Out[72]= 2.22045×10^{-16}

In[73]:= **`a = 1 + $MachineEpsilon`**

Out[73]= `1.`

In[74]:= **InputForm[a]**

Out[74]//InputForm= 1.0000000000000002

In[75]:= **b = 1 + 1.2*^-17**

Out[75]= 1.

In[76]:= **InputForm[b]**

Out[76]//InputForm= 1.

Come potete vedere, in quest'ultimo caso ho sommato ad 1 un valore minore del più piccolo rappresentabile in precisione macchina, quindi il risultato non è variato.

Mathematica è anche in grado di gestire i numeri in base diversa da quella decimale:

b^^*nnnn*	un numero rappresentato in base *b*
BaseForm[*x*, *b*]	stampa *x* in base *b*

Questo ci permette di utilizzare altre basi utilizzate in vari ambiti, per esempio quello informatico:

In[77]:= **2^^11010010001 + 2^^10010001111**

Out[77]= 2848

Come potete vedere, il risultato viene sempre restituito in forma decimale. Se vogliamo invece otterere anche quest'ultimo, nella stessa base, occorre utilizzare BaseForm:

In[78]:= **BaseForm[2848, 2]**

Out[78]//BaseForm= 101100100000_2

Con *Mathematica* possiamo anche trattare, nelle diverse basi, anche numeri reali, oltre che interi:

In[79]:= **16^^ffa39.c5**

Out[79]= 1.0471×10^6

Possiamo anche eseguire delle operazioni con numeri di base diversa fra di loro:

```
In[80]:=  16^^aa34 * 8^^3143

Out[80]=  71240220
```

Un ultimo appunto per i numeri riguarda i risultati indeterminati ed infiniti. Infatti, non c'è precisione che tenga per un risultato di questo tipo:

```
In[81]:=  0 / 0

  — Power::infy : Infinite expression 1/0 encountered. More…

  — ∞::indet : Indeterminate expression 0 ComplexInfinity encountered. More…

Out[81]=  Indeterminate
```

```
In[82]:=  0 ∞

  — ∞::indet : Indeterminate expression 0 ∞ encountered. More…

Out[82]=  Indeterminate
```

In questo caso non possiamo calcolare il risultato, che non ha nessun significato. Pur non potendo restituire nessun numero, *Mathematica* capisce che è dovuto non ad un'approssimazione di calcolo, ma ad una regola matematica, e restituisce Indeterminate come risultato, ovvero un risultato indeterminato.

Analogamente *Mathematica* riconosce un risultato matematicamente infinito:

```
In[83]:=  Tan[π / 2]

Out[83]=  ComplexInfinity
```

Dato che il programma tratta il calcolo simbolico alla pari di quello numerico, il risultato è restituito non come numero, ma come simbolo corrispondente all'equazione, mentre Matlab avrebbe restituito un errore.

Oltre al simbolo per il risultato indeterminato, ce ne sono diversi per l'infinito:

`Indeterminate`	un risultato numerico indeterminato
`Infinity`	una quantità infinita positiva
`-Infinity`	una quantità infinita negativa (`DirectedInfinity[-1]`)
`DirectedInfinity[r]`	una quantità infinita con direzione complessa *r*
`ComplexInfinity`	una quantità infinita di direzione indeterminata
`DirectedInfinity[ ]`	equivalente a `ComplexInfinity`

I simboli di infinito possono essere usati nei calcoli che li comprendono:

```
In[84]:= Sum[1 / (x^3), {x, Infinity}]

Out[84]= Zeta[3]
```

```
In[85]:= 4 / Infinity

Out[85]= 0
```

```
In[86]:= -3 * Infinity

Out[86]= -∞
```

```
In[87]:= Indeterminate * 4

Out[87]= Indeterminate
```

Come vedete, la potenza di questo programma supera già in questo quella di molti altri... E siamo solamente all'inizio dell'inizio...

■ Liste

Creazione

Uno degli aspetti più importanti in *Mathematica* è rappresentato dalle liste. Definirle è semplice; una serie di elementi raggruppati in parentesi graffe:

```
In[88]:= {1, 5, 6, Sin[E^x], {1, 4, 5}, variabile};
```

Quello che si nota dall'esempio è che le liste possono contenere di tutto, come se fossero semplici contenitori. *Mathematica* usa le liste per poter rappresentare una quantità differente di dati, come vettori, matrici, tavole, ecc.

Le operazioni sulle matrici sono molto potenti e sofisticate, e ne permettono una gestione molto avanzata. Facciamo un esempio, e creiamo una lista:

In[89]:= **lista = {a, b, c, d, e, f, g};**

Quando applichiamo una funzione ad una lista, la funzione in generale viene applicata ad ogni elemento della lista, come in questo caso:

In[90]:= **Log[lista]**

Out[90]= $\{2.22045\times10^{-16}, 0., \text{Log}[c], \text{Log}[d], \text{Log}[e], \text{Log}[f], \text{Log}[g]\}$

In[91]:= **2 ^ lista**

Out[91]= $\{2., 2., 2^c, 2^d, 2^e, 2^f, 2^g\}$

In[92]:= **% + 2**

Out[92]= $\{4., 4., 2+2^c, 2+2^d, 2+2^e, 2+2^f, 2+2^g\}$

Come potete vedere, abbiamo anche utilizzato l'operatore percento, per considerare l'ultimo risultato: tanto per tenervi fresca la memoria...

Mathematica usa le liste per la maggior parte dei suoi calcoli. Sono importanti per diversi punti di vista, e si troveranno quaso ovunque. Non bisogna pensarli, in effetti, come semplici contenitori. Si può, per esempio, anche modificare la struttura delle stesse e molto altro. In effetti, dalla manipolazione delle strutture di questo tipo dipende buona parte della potenza del calcolo simbolico di *Mathematica*.

Possiamo anche effettuare delle operazioni sulle liste: partendo da quelle più semplici, è possibile estrarre dalla lista un valore che si trova in una posizione specifica, indicando la posizione entro doppie parentesi quadre:

In[93]:= **lista[[3]]**

Out[93]= c

In[94]:= **lista[[-3]]**

Out[94]= e

In[95]:= **lista[[{1, 4}]]**

Out[95]= {1., d}

Le doppie parentesi quadre sono, a tutti gli effetti, un modo alternativo di scrivere una funzione (effettivamente, tutto in *Mathematica* è equivalente a scrivere funzioni, e proprio in questo sta la sua potenza). La funzione equivalente è Part:

```
In[96]:= Part[lista, 3]

Out[96]= c
```

Come potete vedere, se indichiamo il numero (che potrebbe essere anche una variabile) entro le DOPPIE parentesi quadre (sempre per evitare le ambiguità di scrittura), andiamo a ricavarci il valore corrispondente a quel valore: se l'indice è positivo, *Mathematica* restituisce l'elemento della lista contando a partire dall'inizio mentre, se lo indichiamo con un numero negativo, il programma conterà l'indice a partire dall'ultimo elemento della lista restituendo, nell'esempio, il terzultimo elemento. Inoltre, come indice possiamo anche inserire una lista (possiamo inserire liste quasi ovunque, ed è uno degli aspetti che rende *Mathematica* così potente), in modo da poter selezionare gli elementi che ci servono.

Possiamo anche creare delle liste di liste, cioè delle liste nidificate:

```
In[97]:= nid = {{3, 4, 5}, {21, 3, 643}};
```

In questo caso, per specificare un elemento avremo bisogno di più indici: uno per la posizione nella lista esterna, ed uno per quella corrispondente interna. In questo caso particolare, se volessimo per esempio estrarre il numero 21, dobbiamo considerare che si trova nella seconda 'sottolista', ed in quest'ultima, nella prima posizione, per cui per estrarre questo elemento dovremo scrivere:

```
In[98]:= nid[[2, 1]]

Out[98]= 21
```

Visto? Questo ragionamento si può fare per un qualsiasi livello di nidificazione; oltretutto, non è neanche necessario che le sottoliste abbiano lo stesso numero di elementi:

```
In[99]:= nid2 = {{{2, 4, 5, 2, 4}, {{24, 542, {43, 13}}, 3}, 4, 6, {5, 2, 22}}};
```

Vediamo che, se vogliamo prendere il 43:

```
In[100]:= nid2[[1, 2, 1, 3, 1]]

Out[100]= 43
```

Naturalmente, non dobbiamo scervellarci per scegliere un elemento!!! *Mathematica* ha delle potenti funzioni di ricerca di elementi, che vedremo più avanti, oltre al fatto che naturalmente non avremo

mai a che fare con liste casuali, ma avranno una struttura fortemente organizzata e dipendente dal nostro problema...

Un modo veloce ed efficiente per creare liste è usare la funzione Table:

Table[f, {i_{max}}]	restituisce una lista di i_{max} elementi tutti pari ad f
Table[f, {i, i_{max}}]	restituisce una lista di valori della funzione f[i], con i che varia da 1 a i_{max}
Table[f, {i, i_{min}, i_{max}}]	girestituisce una lista di valori della funzione f[i], con i che varia i_{min} to i_{max}
Table[f, {i, i_{min}, i_{max}, di}]	specifica il passo *di*
Table[f, {i, i_{min}, i_{max}}, {j, j_{min}, j_{max}}, …]	genera una lista multidimensionale

Per esempio, se vogliamo una lista di 7 elementi tutti pari ad 1, possiamo semplicemente scrivere

```
In[101]:= Table[1, {7}]

Out[101]= {1, 1, 1, 1, 1, 1, 1}
```

Se, invece, voglio usare una lista dal 3° al 10° numero primo, posso utilizzare la funzione Prime:

```
In[102]:= Table[Prime[n], {n, 3, 10}]

Out[102]= {5, 7, 11, 13, 17, 19, 23, 29}
```

Facile come bere un bicchier d'acqua, vero? Possiamo anche darle una rappresentazione un tantinello più leggibile, usando la notazione postfissa del comando TableForm:

```
In[103]:= lista // TableForm

Out[103]//TableForm=
    1.
    1.
    c
    d
    e
    f
    g
```

Possiamo anche creare con tranquillità liste di centinaia di migliaia di elementi. In questo caso, però, l'output sarebbe leggermente meno gestibile, no? Se concludiamo il comando di *Mathematica* con il punto e virgola, eviteremo di avere l'output del comando. Questo è particolarmente utile quando dovrete definire liste, vettori e matrici molto lunghi:

In[104]:= **listalunga = Table[Sin[x], {x, 0, 1000, 0.01}];**

con il comando Lenght possiamo velutare il numero di elementi di una lista:

In[105]:= **Length[listalunga]**

Out[105]= 100001

Avete intenzione di visualizzare a schermo una lista così lunga? Io spero di no, e comunque avete appena visto come fare. Questo non vale soltanto per le liste, ma in generale per tutti i comandi di cui non desiderate per un motivo o per un altro l'output.

Un'altra funzione utilissima per creare velocemente delle liste è Range, che permette di creare velocemente liste numerice che contengono un determinato intervallo: con un solo argomento si crea una lista da 1 a n, con due argomenti da n ad m, e con tre argomenti da m ad n con passo d:

Range[*n*]	restituisce la lista {1, 2, 3, … , *n*}
Range[*n*,*m*]	restituisce la lista {*n*, … , *m*}
Range[*n*,*m*,*d*]	restituisce la lista {*n*, … , *m*} con passo *d*

Questo permette di creare dei determinati intervalli di valori molto velocemente, anche di grande dimensioni, senza andare a scomodare Table

In[106]:= **Range[6]**

Out[106]= {1, 2, 3, 4, 5, 6}

In[107]:= **Range[3, 10]**

Out[107]= {3, 4, 5, 6, 7, 8, 9, 10}

In[108]:= **Range[5, 7, .3]**

Out[108]= {5, 5.3, 5.6, 5.9, 6.2, 6.5, 6.8}

Ovviamente, niente ci vieta di andare a creare liste lunghe decine di migliaia di elementi, ma in questo caso di solito non è conveniente visualizzare il risultato:

In[109]:= **a = Range[4, 40000, .01];**

In questa maniera possiamo utilizzare la lista appena ottenuta nella maniera che più ci piace e, soprattutto, che più ci serve...

Un altro comando utile nella creazione delle liste è il seguente:

Array[f, n]	crea una lista di lunghezza n, i cui elementi sono dati da $f[i]$
Array[f, {n_1, n_2, … }]	crea una lista di dimensione $n_1 \times n_2 \times$... , con gli elementi formati da $f[i_1,i_2...]$
Array[f, {n_1, n_2, … }, {r_1, r_2, … }]	genera una lista come nel caso precedente, ma con gli indici che cominciano da {r_1, r_2, … }, invece che da 1 come avviene di default
Array[f, *dims*, *origin*, h]	usa *l'* head h invece che List per ogni livello dell' array

Questa funzione ci permette di creare, quindi, facilmente delle liste che hanno gli elementi che sono dati in funzione dei loro indici

In[110]:= **Array[f, 10]**

Out[110]= {f[1], f[2], f[3], f[4], f[5], f[6], f[7], f[8], f[9], f[10]}

Potrei utilizzare anche più indici, se necessario:

In[111]:= **Array[g, {3, 4}]**

```
Out[111]= {{g[1, 1], g[1, 2], g[1, 3], g[1, 4]},
           {g[2, 1], g[2, 2], g[2, 3], g[2, 4]},
           {g[3, 1], g[3, 2], g[3, 3], g[3, 4]}}
```

Posso creare, per esempio una lista dove compaiano i seni degli indici:

In[112]:= **Array[Sin, 7] // N**

```
Out[112]= {0.841471, 0.909297, 0.14112,
           -0.756802, -0.958924, -0.279415, 0.656987}
```

Possiamo anche utilizzare delle funzioni personalizzate, se lo vogliamo: vedremo più avanti come fare per poter creare delle funzioni personalizzate. Bisogna solamente stare attendi ad utilizzare un numero di indici pari al numero di argomenti della funzione; nel caso seguente, ci sono due indici, quindi chiama la funzione coseno con due indici, mentre tutti noi sappiamo che in realtà ne richiede soltanto uno. In questo caso *Mathematica* capisce che c'è qualcosa che non va e ci avverte:

```
In[113]:= Array[Cos, {4, 4}]

— Cos::argx : Cos called with 2 arguments; 1 argument is expected. More…

— Cos::argx : Cos called with 2 arguments; 1 argument is expected. More…

— Cos::argx : Cos called with 2 arguments; 1 argument is expected. More…

— General::stop :
   Further output of Cos::argx will be suppressed during this calculation. More…

Out[113]= {{Cos[1, 1], Cos[1, 2], Cos[1, 3], Cos[1, 4]},
           {Cos[2, 1], Cos[2, 2], Cos[2, 3], Cos[2, 4]},
           {Cos[3, 1], Cos[3, 2], Cos[3, 3], Cos[3, 4]},
           {Cos[4, 1], Cos[4, 2], Cos[4, 3], Cos[4, 4]}}
```

I messaggi di errore indicano che *Mathematica* non è in grado di valutare la funzione, perchè ci sono due argomenti, mentre la funzione ne richiede uno solo. Notate, tuttavia, come in ogni caso viene restituito il risultato del comando, anche se non in forma valutata. Questo è particolarmente utile quando andiamo ad utilizzare particolari funzioni personalizzate che non sono state pienamente definite, o lo saranno in seguito.

Inoltre, al posto delle liste, possiamo utilizzare delle funzioni personalizzate. Per esempio, notate quello che restituisce il seguente comando:

```
In[114]:= Array[f, {3, 3}, {0, 0}, g]

Out[114]= g[g[f[0, 0], f[0, 1], f[0, 2]],
           g[f[1, 0], f[1, 1], f[1, 2]], g[f[2, 0], f[2, 1], f[2, 2]]]
```

In questo caso abbiamo utilizzato la funzione nella sua forma più completa: abbiamo specificato la funzione da applicare agli indici, che sarebbe la f; poi abbiamo specificato il valore massimo degli indici, ed anche quello minimo. Inoltre, al posto di creare delle liste abbiamo utilizzato la funzione g, mettendo g[...] ogni volta che compariva una lista. Questo è un metodo veloce ed efficace per costruire espressioni complesse, invece che delle liste.

Un altro modo interessante di creare delle liste, utile in casi particolari, è il seguente:

NestList[*f*, *x*, *n*]	{*x*, *f*[*x*], *f*[*f*[*x*]], … } con livelli di annidamente che arriva ad *n*

Questo consente di creare una lista dove gli elementi sono dati dalla ricorsione di una funzione:

In[115]:= **NestList[Sqrt, y, 6]**

Out[115]= $\{y, \sqrt{y}, y^{1/4}, y^{1/8}, y^{1/16}, y^{1/32}, y^{1/64}\}$

Questo permette di creare delle radici quadrate annidate.

In[116]:= **NestList[Exp, 3, 9]**

Out[116]= $\{3, e^3, e^{e^3}, e^{e^{e^3}}, e^{e^{e^{e^3}}}, e^{e^{e^{e^{e^3}}}}, e^{e^{e^{e^{e^{e^3}}}}}, e^{e^{e^{e^{e^{e^{e^3}}}}}}, e^{e^{e^{e^{e^{e^{e^{e^3}}}}}}}, e^{e^{e^{e^{e^{e^{e^{e^{e^3}}}}}}}}\}$

Qua viene visto come la funzione esponenziale viene annidata, e come un elemento della lista sia dato dall'elemento che lo precede a cui sia applicata la funzione selezionata, come da defínzione di ricorsione.

A volte è necessario costruire una lista a partire da un'altra. Per esempio, quando vogliamo applicare una funzione ad ogni elemento della lista; in questo caso molte volte basta applicare la lista come argomento della funzione, ma questo non è vero per tutte le funzioni. Oppure bisogna creare una lista come sottolista di una originale, i cui elementi soddisfino determinati criteri. Ci sono determinati comandi che permettono di creare liste siffatte:

Map[*f*, *list*]	applica *f* a qualsiasi elemento di *list*
MapIndexed[*f*, *list*]	restituisce la funzione *f*[*elem*, {*i*}] per l'elemento *i-simo*
Select[*list*, *test*]	selezione gli elementi per la quale *test*[*elem*] restituisce True

Supponiamo di avere la seguente lista:

In[117]:= **lista = {3, 5, 6, 2, 4, 5, 6, 4, 3, 1};**

Se vogliamo applicare una funzione, possiamo creare per esempio:

In[118]:= **Map[f, lista]**

Out[118]= {f[3], f[5], f[6], f[2], f[4], f[5], f[6], f[4], f[3], f[1]}

Se la funzione è particolare, e necessita di due argomenti, di cui uno è l'indice:

In[119]:= **MapIndexed[g, lista]**

Out[119]= {g[3, {1}], g[5, {2}], g[6, {3}], g[2, {4}], g[4, {5}],
g[5, {6}], g[6, {7}], g[4, {8}], g[3, {9}], g[1, {10}]}

Come possiamo vedere, in questa maniera possiamo creare una lista con delle funzioni in maniera più paricolare rispetto a Map

Certe volte le liste devono essere estremamente lunghe, ma soli una piccola parte dei loro elementi deve essere diversa da zero. Di solito questo non è un problema, ma ci possono essere casi un cui, in una lista di centinaia di migliaia di elementi, solamente un centinaio devono essere diversi da zero; oppure la lista è piena di valori tutti uguali e diversi da zero, e solamente alcuni sono diversi da questo valore. In questo caso, la gestione normale delle liste crea un inutile spreco di memoria, considerando anche che, se in un problema ci serve una lista del genere, probabilmente ce ne serviranno anche altre. Per ovviare a questo problema *Mathematica* fornisce dei comandi che permettono di creare liste sparse di questo tipo con un notevole guadagno di memoria, andando a memorizzare solamente i valori che servono, e non la lista intera:

SparseArray[{i_1->v_1, … }]	crea una lista sparsa, dove l'elemento i_k assume il valore v_k
SparseArray[{pos_1, pos_2, … }-> {val_1, val_2, … }]	restituisce la stessa lista ottenuta con il comando di sopra
SparseArray[*list*]	crea una lista sparsa corrispondente a quella normale data da *list*
SparseArray[*data*, {d_1, d_2, … }	Crea una lista sparsa nidificata, con gli elementi specificati, di dimensione {d_1, d_2, … }
SparseArray[*data*, *dims*, *val*]	Crea una lista di dimensioni specificate, in cui l' elemento che non viene specificato, al posto di assumere valore nullo assume valore *val*

In questo caso, nella creazione delle matrici *Mathematica* crea una rappresentazione interna fatta di puntatori ed altro, una struttura che crea un notevole guadagno di memoria quando si creano appunto matrici con molti elementi, mentre, se la lista è densa, cioè con molti elementi non nulli, questo metodo di memorizzazione diventa inefficiente, dato che per ogni elemento deve esplicitarne la posizione, mentre nel caso normale vengono semplicemente accodate fra di loro. Le matrici sparse sono utilizzate in molti programmi tecnici dove è solito risolvere sistemi lineari con matrici sparse ad elevata dimensione, come ad esempio i CAD di elettronica.

Come abbiamo visto, ci sono diversi modi di creare una matrice sparsa; vediamo questo semplice esempio:

```
In[120]:= listasparsa = SparseArray[{2 → 5, 30 → 3}]

Out[120]= SparseArray[<2>, {30}]
```

Abbiamo creato in questo modo una matrice sparsa. Viene visualizzata in maniera diversa, specificando che ha dimensione 30, e che ci sono 2 elementi non nulli. *Mathematica* non la

rappresenta perchè appunto questa rappresentaizione viene usata per liste di dimensioni elevate, quindi praticamente inutili da visualizzare. Tuttavia possiamo eseguire le stesse operazioni si du una lista:

```
In[121]:= listasparsa[[4]]

Out[121]= 0
```

```
In[122]:= listasparsa[[30]]

Out[122]= 3
```

Abbiamo visto come possiamo estrarne facilmente gli elementi, esattamente come se si trattasse di una lista normale.

```
In[123]:= 2 + listasparsa

Out[123]= SparseArray[<2>, {30}, 2]
```

Anche se tutti gli elementi adesso sono diversi da zero, *Mathematica* continua a mantenere la struttura sparsa, perchè è questa la maniera più conveniente. L'unica differenza è che adesso il risultato contiene pure il valore di tutti gli elementi non specificati, che in questo caso è pari a 2, dato che è stato sommato a tutti gli elementi nulli.

Però, possiamo vedere che in questa maniera, quando definiamo la lista, la dimensione è la minima necessaria a contenere l'ultimo elemento non nullo. Nel nostro caso ha dimensione 30, esattamente la posizione del nostro ultimo elemento. Se vogliamo creare una lista di dimensioni definite, il cui ultimo elemento può anche essere zero, dobbiamo specificarlo nel comando. Il modo grezzo è quello di definire un elemento pari a zero come ultimo elemento. Per esempio, in una matrice di settecento elementi, in cui l'ultimo elemento non nullo è nella centesima posizione, posso scrivere:

```
In[124]:= listasparsa2 = SparseArray[{1 → s, 34 → f, 100 → r, 700 → 0}]

Out[124]= SparseArray[<4>, {700}]
```

Tuttavia, questo metodo ve lo sconsiglio. Invece, vi conviene direttamente esplicitare le dimensioni all'interno del comando, come secondo argomento:

```
In[125]:= listasparsa3 = SparseArray[{1 → s, 34 → f, 100 → r}, 700]

Out[125]= SparseArray[<3>, {700}]
```

Questo metodo è di certo più intuitivo e chiaro, oltre a evitare errori e dimenticanze. Usate sempre questo, mi raccomando.

Nel caso poi ce ne fosse bisogno, possiamo trasformare una lista sparsa in una lista normale, tramite il comando Normal:

```
In[126]:= Normal[listasparsa]

Out[126]= {0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3}
```

Come possiamo vedere, tutti gli elementi specificati sono nulli. Se invece vogliamo specificare un altro valore, dobbiamo scriverlo nella funzione, esattamente come accade per le dimensioni:

```
In[127]:= nonnullo = SparseArray[{3 → 5, 24 → 7}, 25, 3]

Out[127]= SparseArray[<2>, {25}, 3]
```

In questo caso, abbiamo specificato una lista sparsa con gil elementi specificati, di dimensione pari a 25, e in cui gli elementi non specificati sono tutti uguali a 3. Possiamo vederlo meglio andando a scrivere la lista in forma normale:

```
In[128]:= Normal[nonnullo]

Out[128]= {3, 3, 5, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 7, 3}
```

Non sembra esserci tutto questo vantaggio, effettivamente, ma considerate che il vantaggio aumenta all'aumentare delle dimensioni della matrice sparsa, specialmente se è dell'ordine delle migliaio di elementi per dimensione (matrici di centinaia di elementi sono ancora facilmente gestibili da *Mathematica* su di un computer attuale).

Ricerca

Una volta create le liste, è opportuno conoscere qualche basilare funzione e metodo per poterle manipolare; raramente una lista nel corso di uno studio dovrà rimanere costante: spesso sarà necessario aggiungere elementi, toglierli, cambiarli, estrarre dei dati e così via.

Abbiamo già visto l'operazione forse più semplice e quella usata più spesso: cioè l'estrazione di un elemento dalla lista, usando le doppie parentesi quadre:

```
In[129]:= {6, 2, 3, a, 67, 2}[[5]]

Out[129]= 67
```

Le funzioni per estrarre direttamente elementi di una lista sono poche e semplici, e sono qua sotto riassunte:

Take[*list*, *n*]	restituisce l' *n-simo* elemento in *list*
Take[*list*, -*n*]	restituisce l' *n-simo* elemento contando dalla coda
Take[*list*, {*m*, *n*}]	restituisce la lista di elementi dalla posizione *m* alla posizione *n* (incluse)
Rest[*list*]	*list* con il primo elemento scartato
Drop[*list*, *n*]	*list* con i primi *n* elementi scartati
Most[*list*]	*list* l'ultimo elemento scartato
Drop[*list*, -*n*]	*list* con gli ultimi *n* elementi scartati
Drop[*list*, {*m*, *n*}]	*list* con gli elementi da *m* a *n* scartati
First[*list*]	il primo elemento in *list*
Last[*list*]	l'ultimo elemento
Part[*list*, *n*] or *list*[[*n*]]	l'elemento *n-simo* della lista
Part[*list*, -*n*] or *list*[[-*n*]]	l'elemento *n-simo* della lista contando dalla fine
Part[*list*, {n_1, n_2, … }] or *list*[[{n_1, n_2, … }]]	la lista degli elementi in posizione n_1, n_2, …

Creiamoci una lista, e vediamo come possiamo estrarre elementi da essa:

```
In[130]:= Clear[a, b, c, d, e, f, g, h]
```

```
In[131]:= lista = {a, b, c, d, e, f, g, h};
```

Non lasciatevi ingannare dalla semplicità dell'esempio; le lettere possono essere in realtà qualsiasi cosa, da numeri a variabili a funzioni, ad altre liste e così via. Voi sperimentate e poi ditemi se non ho ragione. Comunque, una volta creata la lista, è semplice, per esempio, andare a vedere il primo oppure l'ultimo elemento della lista:

```
In[132]:= First[lista]

Out[132]= a
```

```
In[133]:= Last[lista]

Out[133]= h
```

Invece, per prendere, per esempio, i primi tre elementi della lista, possiamo scriverli direttamente fra le doppie parentesi quadre, oppure usare la funzione Take:

```
In[134]:= lista[[{1, 2, 3}]]

Out[134]= {a, b, c}
```

In[135]:= **Take[lista, 3]**

Out[135]= {a, b, c}

Insomma, avete capito come funzionano le cose, vero?

Tuttavia, a volte desideriamo non tanto estrarre gli elementi, ma solo testare e sapere se qualcosa è presente nella matrice, dopo che abbiamo eseguito un determinato numero di calcoli; per esempio, ci piacerebbe sapere se un elemento è presente nella lista. Certo, potremmo semplicemente visualizzare la lista e cercare, ma se la lista contiene, per esempio, 10000 elementi, come facciamo? E se vogliamo scoprire la posizione di un elemento? Sarebbe scomodo, ed oltretutto, queste funzioni hanno anche l'immenso vantaggio di automatizzare la ricerca che è una cosa utile, per esempio, quando andiamo a scrivere dei programmi in *Mathematica*. A volte non possiamo proprio farne a meno. Ecco quindi alcune utili funzioni per la ricerca di elementi in una lista:

Position[*list*, *form*]	la posizione in cui *form* compare in *list*
Count[*list*, *form*]	il numero di volte che *form* compare come elemento in *list*
MemberQ[*list*, *form*]	verifica se *form* è un elemento di *list*
FreeQ[*list*, *form*]	verifica se *form* non compare da nessuna parte in *list*

Al solito, creiamo una lista, e vediamo come possiamo estrarre e vedere elementi in essa:

In[136]:= **lista = {a, bb, c, a, d, f, d};**

Supponiamo, adesso, di voler contare il numero di volte che *a* compare nella lista:

In[137]:= **Count[lista, a]**

Out[137]= 2

Adesso, vogliamo sapere se, per esempio, *z* è un elemento della lista:

In[138]:= **MemberQ[lista, z]**

Out[138]= False

Il che ci fa capire che z non è un elemento della nostra lista

Inoltre, se la nostra lista è composta da numeri, a volte ci piacerebbe sapere dove si trova, per esempio, l'elemento più grande, e sapere qual'è: in questo caso ci torneranno utili le seguenti funzioni di ricerca nelle liste:

`Sort[`*list*`]`	ordina gli elementi di *list*
`Min[`*list*`]`	l'elemento più piccolo che si trova in *list*
`Ordering[`*list*`,` *n*`]`	la posizione degli *n* elementi più piccoli in *list*
`Max[`*list*`]`	l'elemento più grande in *list*
`Ordering[`*list*`, -`*n*`]`	la posizione degli *n* elementi più grandi in *list*
`Ordering[`*list*`]`	visualizza l'ordine degli elementi in *list*
`Permutations[`*list*`]`	tutte le possibili permutazioni di *list*

Costruiamoci una lista numerica:

```
In[139]:= lista = {3, 5, 4, 8, 1, 3, 6}

Out[139]= {3, 5, 4, 8, 1, 3, 6}
```

Supponiamo di vole conoscere l'elemento più piccolo che si trova nella lista:

```
In[140]:= Min[lista]

Out[140]= 1
```

Adesso, supponiamo di voler conoscere i tre elementi più grandi della lista. Effettivamente non esiste una funzione che permette di ricavarceli direttamente, ma possiamo fare così: prima di tutto, con Ordering calcoliamo la posizione dove si trovano i tre elementi più grandi della lista:

```
In[141]:= Ordering[lista, -3]

Out[141]= {2, 7, 4}
```

Adesso, possiamo usare il risultato come indici per trovare gli elementi:

```
In[142]:= lista[[%]]

Out[142]= {5, 6, 8}
```

Avete capito cos'ho hatto, vero? con Ordering ho trovato gli indici della lista che corrispondono ai tre valori più grandi, e dopo ho usato le doppie parentesi per visualizzare gli elementi corrispondenti ad ogni singolo indice trovato con il comando precedente. Ed, essendo il risultato precedente, perchè riscriverlo quando posso usare l'operatore percento?

Un'altra interessante funzione è Permutations, che permette di osservare tutte le combinazioni degli elementi della lista.

```
In[143]:= Permutations[{a, b, c, d}]

Out[143]= {{a, b, c, d}, {a, b, d, c}, {a, c, b, d}, {a, c, d, b},
  {a, d, b, c}, {a, d, c, b}, {b, a, c, d}, {b, a, d, c}, {b, c, a, d},
  {b, c, d, a}, {b, d, a, c}, {b, d, c, a}, {c, a, b, d}, {c, a, d, b},
  {c, b, a, d}, {c, b, d, a}, {c, d, a, b}, {c, d, b, a}, {d, a, b, c},
  {d, a, c, b}, {d, b, a, c}, {d, b, c, a}, {d, c, a, b}, {d, c, b, a}}
```

Chi si occupa di statistica apprezzerà particolarmente questa funzione, assieme ad un altro centinaio di funzioni implementate in *Mathematica*...

Sebbene questi tipi di ricerca siano molto veloci e completi, tuttavia sono ancora abbastanza limitate, perchè non abbiamo affrontato l'argomento dei pattern, cosa che faremo più avanti. Per adesso faccio soltanto un esempio riguardante le strutture. Possiamo specificare di cercare non dei valori, ma delle espressioni con delle determinate strutture; per esempio, possiamo cercare i casi in cui gli elementi siano degli esponenziali:

```
In[144]:= lista =
  {a, b, c^e, (a x^2 + v)^(3 + t), Sin[v], Cos[b]^Sin[b], x + y, nm*er};

In[145]:= Cases[lista, _^_]

Out[145]= {c^e, (0.99999999999999999999999999999980000000000 a + v)^(3+t), Cos[b]^Sin[b]}
```

In questo modo ho estratto tutti gli elementi della lista che sono stati scritti in forma di esponenziale, identificandio gli elementi con _^_, che significa: 'qualsiasi espressione elevata a qualsiasi altra espressione'. Vedremo molto più avanti come trattare meglio i pattern, anche perchè è con questo metodo che si possono fare profonde ed avanzate ricerche di specifici elementi in liste di milioni di elementi....

Manipolazione

Una volta creata la lista, può essere utile eseguire delle manipolazioni su di essa, come per esempio inserire oppure eliminare qualche specifico elemento. Le funzioni base per poter eseguire queste operazioni sono le seguenti:

`Prepend[`*list*, *element*`]`	aggiungi *element* all'inizio di *list*
`Append[`*list*, *element*`]`	aggiungi *element* alla fine di *list*
`Insert[`*list*, *element*, *i*`]`	inserisci *element* nella posizione *i* in *list*
`Insert[`*list*, *element*, -*i*`]`	inserisci *element* nella posizione *i* contando a dalla fine di *list*
`Delete[`*list*, *i*`]`	elimina l'elemento in posizione *i* in *list*
`ReplacePart[`*list*, *new*, *i*`]`	sostituisci l'elemento in posizione *i* in *list* con *new*
`ReplacePart[`*list*, *new*, {*i*, *j*}`]`	sostituisci *list*`[[`*i*, *j*`]]` con *new*

Possiamo usare come lista di esempio la stessa che abbiamo usato prima:

```
In[146]:= lista = {a, b, c, d, e, f, g, h};
```

e vedere come possiamo modificarla. Supponiamo, per esempio, di voler inserire l'elemento x prima all'inizio, e poi alla fine della lista; la sintassi per poter eseguire queste due operazioni sono elementari, come potete intuire:

```
In[147]:= Prepend[lista, x]

Out[147]= {0.99999999999999999999999999999990000000000, a, b, c, d, e, f, g, h}
```

```
In[148]:= Append[lista, x]

Out[148]= {a, b, c, d, e, f, g, h, 0.99999999999999999999999999999990000000000}
```

Invece, se vogliamo sostituire un elemento della lista, possiamo usare ReplacePart:

```
In[149]:= ReplacePart[lista, ciao, 3]

Out[149]= {a, b, ciao, d, e, f, g, h}
```

Effettivamente, al posto di usare ReplacePart, possiamo semplicemente utilizzare le doppie parentesi per ottenere lo stesso risultato:

```
In[150]:= lista[[3]] = ciao

Out[150]= ciao
```

In[151]:= **lista**

Out[151]= {a, b, ciao, d, e, f, g, h}

Sembrano la stessa cosa, ma ad un esame un attimino più attento si nota una fondamentale importanza: nel primo caso, infatti, con la funzione noi creiamo semplicemente una lista dove viene sostituito un elemento con un altro, ma la lista originale rimane invariata; nel secondo caso, invece, andiamo a modificare l'elemento della lista in esame, che risulta quindi permanentemente modificata: una volta modificato l'elemento, l'elemento sovrascritto viene definitvamente perso, a meno che ovviamente non sia stato salvato prima in un'altra variabile. State attenti, quindi, a decidere quale metodo di sostituzione vorrete utilizzare ogni volta, e dipende principalmente da cosa volete: se vi serve modificare il valore di un elemento della lista, usate le doppie parentesi; se vi serve una lista con l'elemento modificato, ma non volete modificare la lista originale perchè vi servirà in seguito, per esempio, per eseguire dei test, usate ReplacePart, magari memorizzando la nuova lista con un altro nome.

Inoltre, le liste possono anche essere unite assieme: in questo caso, ci sono due funzioni per unire le liste, apparentemente simili, ma con un'importante differenza:

Join[$list_1$, $list_2$, …]	concatena assieme le liste
Union[$list_1$, $list_2$, …]	combina le liste, rimuovendo gli elementi duplicati e riordinando i restanti elementi
Intersection [$list_1$, $list_2$, …]	crea una lista contenente elementi comuni a tutte le liste
Complement[*universal*, $list_1$, …]	crea una lista con tutti gli elementi presenti in *universal*, ma che non sono contenuti in nessuna delle altre liste elencate
Subsets[*list*]	Crea una lista contenente tutte le possibili sottoliste di *list*

Quando vogliamo semplicemente unire due liste, conviene usare il comando Join, mentre, se invece occorre prendere tutti gli elementi delle liste, per unirle in un'unica lista dove compaiono soltanto gli elementi distinti, allora occorre usare il comando Union. Prestate attenzione al differente modo di operare di queste due funzioni, mi raccomando. Le altre funzioni non credo che abbiano bisogno di particolari commenti o delucidazioni. comunque, il mio consiglio è quello di sperimantare quanto più potete, per vedere quello che potete fare, quello che avete capito e quello che invece vi manca. A proposito, vi ho già suggerito di leggervi l'help on line ogni volta che ne avete bisogno, vero? :-)

Un altro modo per elaborare le liste consiste nel poterle riordinare come meglio ci aggrada, in ordine crescente oppure decrescente, e implementando anche lo scorrimento della lista:

`Sort[`*list*`]`	ordina gli elementi di *list* in maniera standard
`Union[`*list*`]`	sort riordina gli elementi, eliminando i duplicati
`Reverse[`*list*`]`	inverte l'ordine degli elementi in *list*
`RotateLeft[`*list*`, `*n*`]`	scorre gli elementi di *list* di *n* posizioni verso sinistra
`RotateRight[`*list*`, `*n*`]`	non riuscirete mai ad indovinare cosa fa questa funzione...

Notate il riutilizzo del comando Union : prima l'avevamo utilizzato per unire le liste scartando gli elementi duplicati, ma aveva anche l'effetto di riordinarli. Qua fa effettivamente la stessa cosa, ma limitandosi ad una lista soltanto. Vediamo con il solito esempiuccio come funzionano queste funzioni:

```
In[152]:= lista = {a, b, g, r, t, s, w, r, x, v, a, b};
```

Come potete vedere, la lista questa volta è formata da elementi casuali e disordinati.

```
In[153]:= Sort[lista]

Out[153]= {0.99999999999999999999999999999990000000000,
           a, a, b, b, g, r, r, s, t, v, w}
```

```
In[154]:= Union[lista]

Out[154]= {0.99999999999999999999999999999990000000000, a, b, g, r, s, t, v, w}
```

Qua potete vedere la differenza fra le due funzioni, e come il secondo comando permetta, oltre all'ordinamento, anche l'eliminazione degli elementi duplicati. Bisogna fare, invece, un attimino più di attenzione al comando Reverse:

```
In[155]:= Reverse[lista]

Out[155]= {b, a, v, 0.99999999999999999999999999999990000000000,
           r, w, s, t, r, g, b, a}
```

Come possiamo vedere, non esegue l'ordinamento degli elementi di una lista, come potrebbe ad alcuni sembrare, dato che qua parliamo di ordinamento: semplicemente, inverte gli elementi della lista; se vogliamo avere un'ordinamento inverso, però, niente paura: basta invertire la lista precedentemente ordinata, no?

```
In[156]:= Reverse[Sort[lista]]

Out[156]= {w, v, t, s, r, r, g, b, b, a, a,
           0.99999999999999999999999999999990000000000}
```

Tanto semplice quanto efficace. Ricordate solo che quando effettuate operazioni di questo tipo, la lista originale rimane invariata, dato che i comandi forniscono solamente un risultato: se volete che la lista cambi effettivamente, dovete scrivere qualcosa del tipo:

```
In[157]:= lista = Union[lista]

Out[157]= {0.9999999999999999999999999999990000000000, a, b, g, r, s, t, v, w}
```

e, andando a vedere il contenuto della lista, si vede che ora è effettivamente cambiato:

```
In[158]:= lista

Out[158]= {0.9999999999999999999999999999990000000000, a, b, g, r, s, t, v, w}
```

Comunque, sono sicuro che non avrete difficoltà a capire questo, se avete, almeno una volta nella vostra vita, messo mano ad un qualsiasi linguaggio di programmazione. Fatevi qualche eserciziettto e sperimentate, dato che le liste sono elementi molto importanti, in *Mathematica*, specialmente quando si devono, per esempio, eseguire operazioni su dei dati sperimentali di un esperimento oppure di una simulazione.

■ Vettori e Matrici

Le liste sono anche il modo che ha *Mathematica* per poter scrivere vettori e matrici: il vettore si scrive come una lista semplice, mentre la matrice si scrive come una lista di liste, dove le liste elementi della lista principale rappresentano le righe della matrice in esame:

```
In[159]:= vet = {a, b, c};

In[160]:= mat = {{a, b}, {c, d}};
```

Per estrapolare gli elementi, si usano pure in questo caso, dato che in fondo di tratta di liste, le doppie parentesi quadre per indicare gli indici.

```
In[161]:= vet[[1]]

Out[161]= a

In[162]:= mat[[1]]

Out[162]= {a, b}
```

Questo perchè, dal punto di vista del programma, una matrice è una lista di liste, e il primo elemento di una lista è proprio la prima sotto-lista, che rappresenta la prima riga. Tutto regolare, quindi. Per poter estrapolare un singolo elemento, bisogna usare due indici

In[163]:= **mat[[1, 2]]**

Out[163]= b

Semplice, no?

Le operazioni con i vettori sono molto semplici; possiamo sommare i vettori:

In[164]:= **vet + {e, r, t}**

Out[164]= {a + e, b + r, c + t}

Tuttavia, se proviamo a moltiplicare gli elementi dei vettori, non utilizziamo il risultato voluto:

In[165]:= **vet {e, r, t}**

Out[165]= {a e, b r, c t}

Questo perchè la moltiplicazione viene eseguita fra elementi delle liste aventi lo stesso indice. per poter effettuare il prodotto scalare o, più in generale, il prodotto righe per colonne, dobbiamo utilizzare il comando del prodotto vettoriale, che è semplicemente il punto:

In[166]:= **vet.{e, r, t}**

Out[166]= a e + b r + c t

In[167]:= **Sin[{x, y, z}].{2, 3, 4}**

Out[167]= 1.68294196961579301330500464325951739463 3 + 3 Sin[y] + 4 Sin[z]

In[168]:= **mat.{1, 2}**

Out[168]= {a + 2 b, c + 2 d}

Un modo utile di definire i vettori (ma anche liste: ricordate che in fondo in *Mathematica* sono la stessa cosa...) è usare il comando Array: richiere cone primo argomento il nome della funzione, e come secondo argomento un numero, od una lista di numeri, che andranno a rappresentare l'argomento: in pratica, avendo gli indici i,j, costruisce la matrice A dove aij=funz[i,j]

In[169]:= **Array[funz, 5]**

Out[169]= {funz[1], funz[2], funz[3], funz[4], funz[5]}

In[170]:= **Array[funz, {3, 2}]**

Out[170]= {{funz[1, 1], funz[1, 2]},
{funz[2, 1], funz[2, 2]}, {funz[3, 1], funz[3, 2]}}

Per le matrici, due funzioni utili sono quelle che permettono di creare matrici identità oppure diagonali: nel primo caso occorre come argomento la dimensione n della matrice (ne basta una, è quadrata...), nel secondo caso occorre la lista di elementi diagonali. Vedremo la scrittura anche per poter ottenere un'output più consono a quello delle matrici:

In[171]:= **IdentityMatrix[3] // MatrixForm**

Out[171]//MatrixForm=

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

In[172]:= **DiagonalMatrix[{1, 2, r, 4}] // MatrixForm**

Out[172]//MatrixForm=

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & r & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

E possiamo anche calcolarci le dimensioni di una matrice o di un vettore:

In[173]:= **Dimensions[%]**

Out[173]= {4, 4}

Qua sotto sono riportate pochissime ma indispensabili formule per poter operare con le matrici:

c m	moltiplicazione per uno scalare
a . *b*	prodotto matriciale
Inverse[*m*]	inversa di una matrice
MatrixPower[*m*, *n*]	*potenza* n^{th} di una matrice
Det[*m*]	determinante
Tr[*m*]	traccia
Transpose[*m*]	trasposta
Eigenvalues[*m*]	autovalori di una matrice
Eigenvectors[*m*]	autovettori di una matrice

Sempre considerando la matrice mat creata in precedenza, possiamo scrivere, per fare qualche veloce esempio:

In[174]:= **Det[mat]**

Out[174]= $-b\,c+a\,d$

In[175]:= **Eigenvalues[mat]**

Out[175]= $\left\{\frac{1}{2}\left(a+d-\sqrt{a^2+4\,b\,c-2\,a\,d+d^2}\right),\ \frac{1}{2}\left(a+d+\sqrt{a^2+4\,b\,c-2\,a\,d+d^2}\right)\right\}$

In[176]:= **Inverse[mat]**

Out[176]= $\left\{\left\{\frac{d}{-b\,c+a\,d},\ -\frac{b}{-b\,c+a\,d}\right\},\ \left\{-\frac{c}{-b\,c+a\,d},\ \frac{a}{-b\,c+a\,d}\right\}\right\}$

In questi semplici esempi potete vedere come *Mathematica* tratta con la stessa semplicità sia calcoli numerici che calcoli simbolici. Vi risparmo la scrittura degli autovettori, ma potete provarla da soli per vedere come facilmente si possono ottenere risultati di tutto rispetto. Sono pochi i programmi che permettono un calcolo simbolico così sofisticato...

Adesso, dopo aver visto qualche nozione di base di *Mathematica*, andiamo a scalfire un poco più a fondo le sue capacità, andando a vedere un pochino di calcolo simbolico. Non allarmatevi, ma siate rilassati e contemplate la potenza di questo programma. D'altra parte, prima di scrivere il resto mi prenderò una pausa ed andrò a farmi un bagno al mare. Non dispiace a nessuno, vero? A presto, allora!!!

■ Nota sulla visualizzazione delle formule

In *Mathematica*, si scrivono le equazioni e tutto il resto sotto forma si testo: L'integrale si scriverà, per esempio, con il comando Integral: potete effettivamente usare la palette che compare di default per poter scrivere direttamante in forma tradizionale, ma io personalmente ve lo sconsiglio. Anche se le prime ore sarete tentati a farlo, dopo che vi sarete scritti una decina di integrali scoprirete che è molto più veloce battere Integral sulla tastiera che dover andare a cliccare col mouse prima nel simbolo d'integrale e dopo in ogni quadratino per poter scrivere tutto quanto.

Tuttavia, se dovete, per esempio, stampare qualche lavoro, una forma tradizionale di presentazione delle equazioni è certamente preferibile. In questo caso, la soluzione esiste ed è anche particolarmente semplice. Una volta finito il lavoro, potete cliccare col mouse sulla linea a destra del Notebook che definisce la formula, ed a questo punto selezionare la voce di menù Cell->Convert To->Traditional Form (o, più velocemente, premendo Shift+CTRL+T). In questo modo, la formula sarà convertita in una scrittura standard, di certo più comprensibile a chi non usa *Mathematica*. Però, in questo caso, la formula non sarà più univocamente definita, per esempio per il problema delle parentesi di cui vi ho accennato. In questo caso *Mathematica* lo segnala evidenziando in modo particolare la cella. Se volete di nuovo convertire la formula in formato di *Mathematica*, potete andarci sempre dal menù Cell->Convert To->Input Form, ma in questo caso *Mathematica* userà metodi euristici per convertitre di nuovo il risultato, con una piccola probabilità di errore:

Per esempio, vediamo come si scrive una formula per l'integrale:

```
In[177]:= Integrate[(x + 1) ^ Sin[x], {x, 0, 3 Pi}]

— Integrate::ilim : Invalid integration variable or limit(
     s) in {0.99999999999999999999999999999990000000000, 0, 3 π}. More…
```

$$\text{Out[177]=} \int_0^{3\pi} 1.7918762238279220192681893154261648908 19 \, d0.99999999999999999999999999999990000000000$$

Sebbene sia una formula semplice, non è certo questa la forma di più facile comprensione per il profano, e si vorrà certamente avere una scrittura più da libro: in questo caso, basta convertire la formula con il metodo che vi ho appena spiegato, per ottenere:

$$\int_0^{3\pi} (x+1)^{\sin(x)} \, dx$$

E di certo, in questo modo è di più facile comprensione per tutti (notate come la cella è segnata diversamente). Dato che, però, occorrerà a volte ritornare sul lavoro per ulteriori elaborazioni, io personalmente, quando devo stampare qualcosa, salvo il file con un altro nome e uso la copia per convertire la scrittura delle formule. Se poi dovessi riutilizzare il file, vado a modificare di nuovo

l'originale, in modo che non ci siano problemi di mancato riconoscimento di formule.

Vedremo, comunque, melgio questo aspetto nel capitolo riguardante il front-end e la formattazione.