Calcolo Numerico

Introduzione

Quello che abbiamo visto finora riguarda principalmente il calcolo simbolico. Anche se è uno degli element che contraddistinguono questo programma dalla massa, sia per il numero di funzioni che per la loro potenza, le caratteristiche del programma non si fermano certo in quest'aspetto, anzi....

Mathematica contiene una quantità spropositata di algorotmi dedicati per poter risolvere la totalità dei problemi numerici che possono capitare a questo mondo, ed in questo universo. Le caratteristiche fondamentali sono la scelta automatica dell'algoritmo e la precisione arbitraria. Con il primo punto, possiamo stare (quasi) sempre tranquilli e lasciare decidere al programma il metodo migliore per la risoluzione di un problema: per esempio, per un integrale numerico ci sono i metodi di Simpson, quello dei trapezi e molti altri che neanche io conosco... *Mathematica* riesce ad analizzare la funzione, ed è in grado di decidere da solo quale metodo utilizzare per avere la soluzione migliore al problema. Chi ha usato qualche volta, in elettronica il programma SPICE (quello originale della Berkley, non uno dei tanti programmi presenti con grafici etc), sa che vengono usati due principali algoritmi per la risoluzione delle formule integrali: il metodo trapezoidale e quello di Gear. Sono due differenti metodi per trovare integrali numerici. Di solito le formule di Gear danno risultati più precisi, ma hanno problemi di stabilità: per sistemi stiff, cioè con poli lontani dall'origine, può capitare che il metodo diverga anche quando in realtà converge, e, in minor parte, convergere quando in realtà diverge, dato che il sistema teorico e quello numerico hanno regioni diverse di stabilità, dando quindi dei risultati errati, mentre il metodo trapezoidale è a volte meno preciso, ma rispetta rigorosamente il concetto di stabilità di un sistema, divergendo solo quando effettivamente siamo di fronte a delle instabilità. Questo porta a dover giocare con le opzioni per essere sicuri di trovare il metodo che più ci aggrada, e quello che serve per farci avvicinare con maggior precisione al risultato. *Mathematica*, d'altrone, per ogni funzione numerica è in grado di trovare da sola l'algoritmo ideale, applicandolo in maniera del tutto trasparente. Quando andremo a risolvere un'equazione differenziale, dovremo 'solo' preoccuparci di porre il problema nella maniera corretta, con le giuste condizioni al contorno, e *Mathematica* farà il resto. Ci concentriamo, insomma, sul cosa, non sul come. Questo non toglie il fatto che siamo perfettamente in grado di dire al programma come risolvere un problema, forzando ad esempio un determinato algoritmo, ma non sono cose di cui ci preoccupiamo, almeno per ora. Personalmente, non ho mai detto a *Mathematica* quale algoritmo utilizzare, e non credo che lo farò mai, a meno di curiosità.

In aggiunta a questo, possiamo anche sfruttare la precisiona arbitraria di *Mathematica*. Non solo il programma è in grado di decidere quale algoritmo applicare, ma lo fa con una precisione che possiamo decidere da soli. Non siamo costretti ad accontentarci della risoluzione dovuta alla precisione macchina, se 32 o 64 bit. Se vogliamo una soluzione con 10000 cifre significative, con

Mathematica possiamo farlo tranquillamente; il programma basa i suoi calcoli su potenti algoritmi a precisione arbitraria, che penalizzano quasi di niente la velocità di esecuzione degli algoritmi. Questi ultimi, fra l'altro, sono altamente ottimizzati, dandoci possibilità di eseguire un numero elevato di operazioni in relativamente poco tempo.

Adesso, andiamo a vedere le funzioni numeriche di *Mathematica*. Sono per lo più simili a quelle che abbiamo già visto per quanto riguarda il calcolo simbolico, almeno nel loro funzionamento base, e quando non andiamo a toccare le opzioni di default. Se poi vogliamo, possiamo spulciarle per risolvere anche l'impossibile, utilizzando funzioni quali NSolve ed NDSolve che sono sicuramente fra i comandi matematici più potenti in assoluto non soltanto in *Mathematica*, ma in qualsiasi altro software scientifico che possiate concepire.

- **Sommatorie, Produttorie, Integrali**

Abbiamo visto come si risolvono le equazioni simboliche: se ci serve un rusiltato numerico, oppure il programma non è in grado di trovare la soluzione simbolica, possiamo utilizzare delle funzioni che sono molto simili, in quanto a sintassi:

`NSum[f, {i, imin, Infinity}]`	approssimazione numerica di $\sum_{i_{min}}^{\infty} f$
`NProduct[f, {i, imin, Infinity}]`	approssimazione numerica di $\prod_{i_{min}}^{\infty} f$
`NIntegrate[f, {x, xmin, xmax}]`	approssimazione numerica di $\int_{x_{min}}^{x_{max}} f\, d\,x$
`NIntegrate[f, {x, xmin, xmax}, {y, ymin, ymax}]`	approssimazione di $\int_{x_{min}}^{x_{max}} d\,x \int_{y_{min}}^{y_{max}} d\,y\, f$

Proviamo a effettuare qualche calcolo:

```
In[1]:= NSum[Log[x] / x^5, {x, 2, Infinity}]
Out[1]= 0.0285738 + 0. i
```

```
In[2]:= NIntegrate[1 / Sqrt[Gamma[x] (1 - x)], {x, 0, 1}]
Out[2]= 1.61784
```

Gli integrali possono essere anche calcolati in regioni infite, naturalmente:

```
In[3]:= NIntegrate[Cos[x] Exp[-Abs[x^2]], {x, -Infinity, Infinity}]
Out[3]= 1.38039
```

Il risultato ottenuto è numerico: è anche possibile lavorare sulla precisione, ma ne parlaremo poco più avanti, dato che bisogna ragionarci un attimino per poter ottenere il risultato voluto.

■ **Soluzione di equazioni**

Anche in questo caso, le funzioni sono simili, anche se si aggiunge una funzione interessante, fatta apposta per il calcolo numerico:

`NSolve[`*lhs*`==`*rhs*`, `*x*`]`	risolve un'equazione polinomiale numericamente
`NSolve[ {`lhs_1 `= =`rhs_1 `, `lhs_2`==`rhs_2`, … }, {`*x*`, `*y*`, … }]`	risolve un sistema di equazioni polinomiali nuericamente
`FindRoot[`*lhs*`==`*rhs*`, {`*x*`, `x_0`}]`	cerca una soluzione numerica con seme $x = x_0$
`FindRoot[ {`lhs_1 `= =`rhs_1 `, `lhs_2`==`rhs_2`, … }, {{`*x*`, `x_0`}, {`*y*`, `y_0`}, … }]`	cerca soluzioni numeriche per un sistema di equazioni

Abbiamo visto che non esistono soluzioni generali per equazioni polinomiali per un grado superiore a 4. Tuttavia, possiamo ancora, con il comando NSolve, trovare il risultato numerico per tutte le radici:

```
In[4]:= eq = x^7 + 5 x^6 - 34 x^5 + 3 x^4 - 18 x^3 - 64 x + 2 == 0;

In[5]:= NSolve[eq, x]

Out[5]= {{x → -8.88949}, {x → -0.73661 - 0.860533 i},
  {x → -0.73661 + 0.860533 i}, {x → 0.0312415},
  {x → 0.716525 - 0.962437 i}, {x → 0.716525 + 0.962437 i}, {x → 3.89842}}
```

Ovviamente, anche in questo caso noi siamo in grado, tramite questo comando, di trovare soluzioni di un sistema di equazioni algebrico:

```
In[6]:= sis = {x^3 - 4 y^2 == 3 z^5,
     x - y == z^2,
     x^2 + y^2 + z^1 == 0};

In[7]:= NSolve[sis, {x, y, z}] // TableForm
```

Out[7]//TableForm=

x → -43.596 - 36.3488 i	y → -36.4276 + 43.5925 i	z → -6.0454 +
x → -43.596 + 36.3488 i	y → -36.4276 - 43.5925 i	z → -6.0454 -
x → 0.118835	y → -1.028	z → -1.0709
x → -0.276836 - 0.943164 i	y → 1.19596 - 0.727046 i	z → -0.088803
x → -0.276836 + 0.943164 i	y → 1.19596 + 0.727046 i	z → -0.088803
x → 0.670104	y → 0.346348	z → -0.568996
x → -0.607146 + 0.662123 i	y → 0.000898087 + 0.351493 i	z → 0.193327 +
x → -0.607146 - 0.662123 i	y → 0.000898087 - 0.351493 i	z → 0.193327 -
x → 0.0855246 - 0.855851 i	y → -0.428419 - 0.468186 i	z → 0.760822 -
x → 0.0855246 + 0.855851 i	y → -0.428419 + 0.468186 i	z → 0.760822 +
x → 3.76926 × 10^-12	y → 3.76926 × 10^-12	z → 0.
x → 0.	y → 0.	z → 0.

Effettivamente, sono un pochino di soluzioni in più di quanto pensassi... Non devo più fare degli esempi a caso, mi sa...

Vediamo come possiamo facilmente trovare tutte le soluzioni, e senza bisogno di andare a mettere punti iniziali per tentare di trovarli ad uno ad uno. Purtroppo, questo però è l'unico metodo se le equazioni non sono algebriche, ma di tipo generale. Questo significa che abbiamo bisogno di un punto iniziale per iniziare la ricerca, e che troveremo soltanto una soluzione, anche se ne sono presenti più di una. Per trovare altre eventuali soluzioni, dobbiamo creare un altro punto iniziale ed effettuare un'altra ricerca. In questo caso, bisogna usare il comando FindRoot:

```
In[8]:= eq = 10 Cos[ Cos[x]] == Log[x^2];

In[9]:= FindRoot[eq, {x, 1}]

Out[9]= {x → -33.8957}
```

Tuttavia possiamo vedere come, se usiamo un altro seme, possiamo andare a trovare un altra soluzione:

```
In[10]:= FindRoot[eq, {x, 5}]

Out[10]= {x → -41.5941}
```

Si può vedere che le soluzioni, in questo caso, sono parecchie, e occorre un'oculata scelta del punto iniziale. Fra l'altro, potete anche vedere che non è vero che basta prendere un punto vicino ad una soluzione, per ottenere proprio quella cercata. Questo perchè nell'algoritmo iterativo che *Mathematica* sceglie, può capitare che il punto, durante le iterazioni, si allontani.

E, tanto per cambiare, possiamo usare FindRoot anche per poter trovare soluzioni di un sistema di equazioni qualsiasi:

```
In[11]:= sis = {Cos[x] == Log[Sqrt[x y]], Tan[y] == Gamma[x]};

In[12]:= FindRoot[sis, {{x, 1}, {y, 3}}]

Out[12]= {x → 0.889268, y → 3.96425}
```

Potete vedere come le possibilità di trovare la soluzione numerica sono sempre quasi certe, anzi praticamente sempre, a parte problemi di convergenza, che comunque capitano davvero raramente, data la robustezza degli algoritmi impiegati.

- **Equazioni differenziali**

Quello che abbiamo visto per le equazioni, possiamo averlo anche per quelle differenziali, anche se con una piccola differenza. Infatti bisogna considerare che adesso, invece di dover restituire dei valori, il comando restituisce una funzione interpolata, dato che viene calcolata per punti: inoltre, dato che la soluzione è numerica, dobbiamo sempre definire completamente le condizioni al contorno:

`NDSolve[`*eqns*`, `*y*`, {`*x*`, `x_{min}`, `x_{max}`}]`	risolve numericamente la funzione *y*, con la variabile indipendente *x* nel raggio da x_{min} a x_{max}
`NDSolve[`*eqns*`, {`y_1`, `y_2`, …`}, {`*x*`, `$x_{min}$`, `$x_{max}$`}]`	risolve numericamente la funzione nelle variabili y_i

In questo modo, vediamo come possiamo risolve numericamente equazioni differenziali anche abbastanza difficili:

```
In[13]:= diff = y'[x] + Log[y[x]] == x;

In[14]:= NDSolve[{diff, y[1] == 2}, y, {x, 2, 7}]

Out[14]= {{y → InterpolatingFunction[{{2., 7.}}, <>]}}
```

Vediamo un'importante differenza, cioè che dobbiamo specificare y come funzione pura, dato che troviamo *y* e non *y*[*x*]: tuttavia per ora non importa molto, se sappiamo come utilizzarla. Una volta trovata la soluzione, infatti, per poter trovare un valore all'interno dell'intervallo dove l'abbiamo risolta, dobbiamo usare la regola:

```
In[15]:= y[4] /. %

Out[15]= {5.91218}
```

Viene detto a *Mathematica*, in pratica, di sostituire al nome della funzione la funzione pura, in modo che la funzione si trasformi in quella desiderata. Le regole, infatti, oltre che alle variabili, si applicano anche ai nomi delle funzioni; in effetti, si applicano a qualsiasi cosa...

Se andiamo fuori range, la funzione non è definita, e *Mathematica* non sa come calcolarla:

```
In[16]:= y[40] /. %%

  – InterpolatingFunction::dmval :
     Input value {40} lies outside the range of data in the
       interpolating function. Extrapolation will be used. More…

Out[16]= {723.148}
```

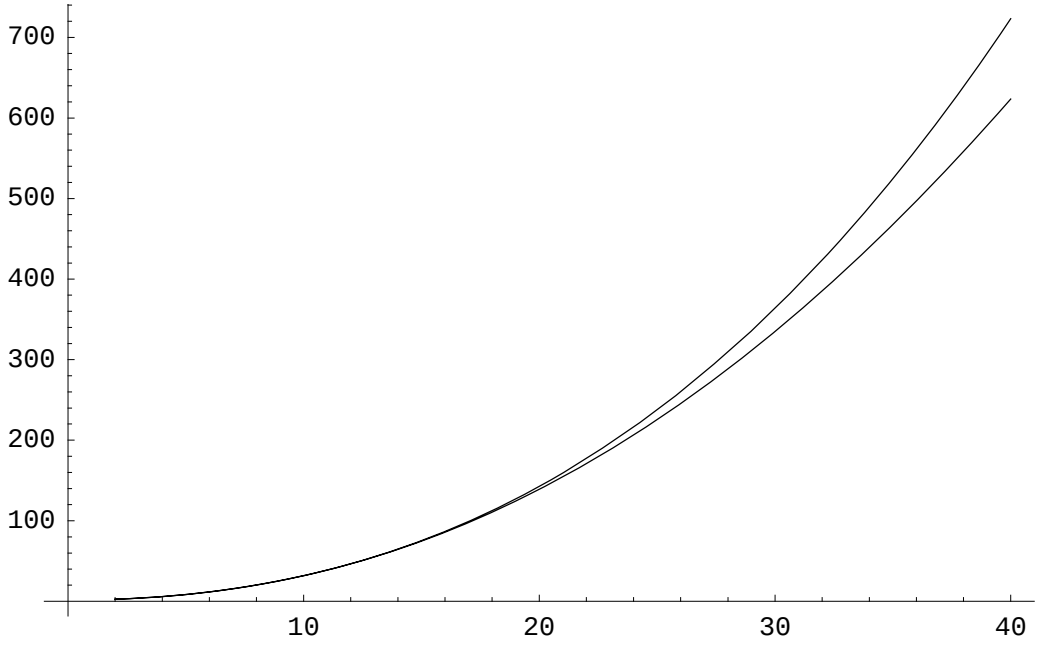
In questo caso si decide di eseguire una estrapolazione, soluzione che io sconsiglio sempre. Possiamo vedere di calcolare la funzione anche in quest'ultimo punto:

```
In[17]:= NDSolve[{diff, y[1] == 2}, y, {x, 2, 40}]

Out[17]= {{y → InterpolatingFunction[{{2., 40.}}, <>]}}
```

E vedere di quanto differisce la soluzione estrapolata da quella interpolata all'interno dell'intervallo in cui abbiamo definito la funzione:

```
In[18]:= y[40] /. %

Out[18]= {623.523}
```

Possiamo vedere come differiscano di una quantità notevole, molto al di là del semplice errore di approssimazione numerica.



Notiamo come non abbiamo dovuto specificare nessun particolare algoritmo per trovare la soluzione, dato che *Mathematica* lo sceglie automaticamente tramite dei criteri molto rigorosi. In effetti, NDSolve è uno dei comandi più potenti che ci siano all'interno di *Mathematica*, perchè permette di risolvere problemi che sono veramente complicati, di quelli per cui si utilizzano i supercomputer; infatti, del programma ne esiste una versione apposita da far girare su quei mostri, ed utilizza gli stessi algoritmi presenti nella versione standard.

Il motivo principale della potenza di questo comando risiede principalmente nel poter stabilire, nella quasi totalità dei problemi, automaticamente il tipo di algoritmo da utilizzare, evitando problemi di stabilità e precisione che derivano dall'utilizzo di un algoritmo non adatto al problema. Un esempio

classico si ha nella scelta fra le formule di Gear e quella trapezoidale; mentre la prima ha una precisione maggiore, la seconda presenta un diagramma di stabilità che coincide con quello teorico, per cui, se una soluzione deve mettersi ad oscillare, con il trapezoidale lo farà, anche se magari avrò uno scarto maggiore fra la soluzione vera e quella approssimata. Invece, con Gear, possiamo raggiungere una maggior precisione, ma ha un diagramma di stabilità diverso, il che implica che potrebbe non oscillare la soluzione di un'equazione dofferenziale, cosa che invece accade poi nella realtà. Capite quindi l'importanza di scegliere il giusto algoritmo, ma chi ha fatto calcolo numerico oppure materie di progettazione assistita da calcolatore, conosce bene questi problemi. il bello è che *Mathematica* li conosce pure, e fa la fatica di scegliere il giusto algoritmo al posto nostro. Come ci riesce, non lo so... bisognerebbe chiederlo direttamente a Sthepen Wolfram, se qualcuno ne è capace...

- **Ottimizzazione numerica**

Naturalmente, nessun programma che vanti propietà di calcolo numerico si può definire tale senza delle funzioni specifiche per trovare i massimi ed i minimi delle funzioni. *Mathematica* offre comandi specifici nel caso si voglia trovare il massimo oppure il minimo globale di una funzione e, nel caso non esista o non sia fattibile, anche funzioni per trovare massimi e minimi locali:

`NMinimize[`*f*`, {`*x*`, `*y*`, … }]`	minimizza *f*
`NMaximize[`*f*`, {`*x*`, `*y*`, … }]`	massimizza *f*
`NMinimize[{`*f*`, `*ineqs*`}, {`*x*`, `*y*`, … }]`	minimizza *f* sotto le condizioni di *ineqs*
`NMaximize[{`*f*`, `*ineqs*`}, {`*x*`, `*y*`, … }]`	massimizza *f* sotto le condizioni di *ineqs*
`FindMinimum[`*f*`, {`*x*`, `x_0`}]`	cerca il minimo locale di *f*, partndo da $x = x_0$
`FindMinimum[`*f*`, {{`*x*`, `x_0`}, {`*y*`, `y_0`}, … }]`	cerca il minimo locale in più variabili
`FindMaximum[`*f*`, {`*x*`, `x_0`}]`	cerca per un massimo locale

La funzione NMinimize e NMaximize, oltre che trovare il punto dove la funzione è minima, restituisce anche il valore della funzione in quel punto:

```
In[19]:= NMaximize[x / (1 + Exp[x]), x]
Out[19]= {0.278465, {x → 1.27846}}
```

Possiamo imporre anche delle regioni dove trovare il minimo oppure il massimo:

```
In[20]:= eq = Cos[x] Sin[x y];
In[21]:= regione = x^2 + y^2 < 2;
```

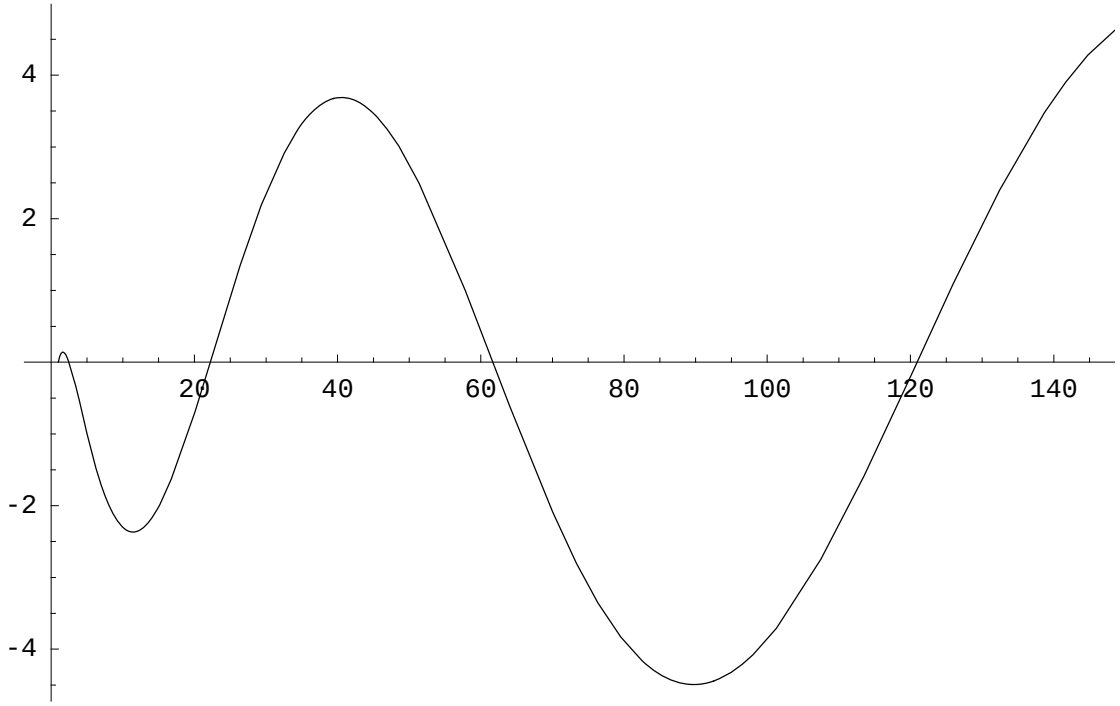
```
In[22]:= NMaximize[{eq, regione}, {x, y}]
Out[22]= {0.580755, {x → 0.672134, y → 1.24428}}
```

Nel caso che non sia possibile trovare direttamente il minimo globale, o se si è interessati a trovarne uno locale, dobbiamo andare ad utilizzare le altro funzioni che abbiamo visto sopra:

```
In[23]:= FindMinimum[Log[x] Cos[Sqrt[x]], {x, 10}]
Out[23]= {-2.36686, {x → 11.4238}}
```

```
In[24]:= FindMinimum[Log[x] Cos[Sqrt[x]], {x, 130}]
Out[24]= {-4.49167, {x → 89.7131}}
```

Possiamo vedere come la stessa funzione, dati valori iniziali diversi, abbia risultati diversi, segno che in questo caso ci troviamo di fronte ad una funzione con più minimi, e ne scegliamo alcuni a seconda del punto iniziale.



CVD... Vedete come siano presenti più minimi (e anche più massimi), e come la funzione diverge e non sia effettivamente presente un minimo (massimo) globale.

Trovare il minimo (o massimo) assoluto numericamente può diventare difficile... Il problema principale risiede sempre nel fatto che numericamente si eseguono varie iterazioni che, a seconda della forma della funzione, possono non trovare esattamente il minimo od il massimo globale, ma soltanto uno locale. Questo dipende sempre dal metodo di ricerca, e da come viene scelto il punto

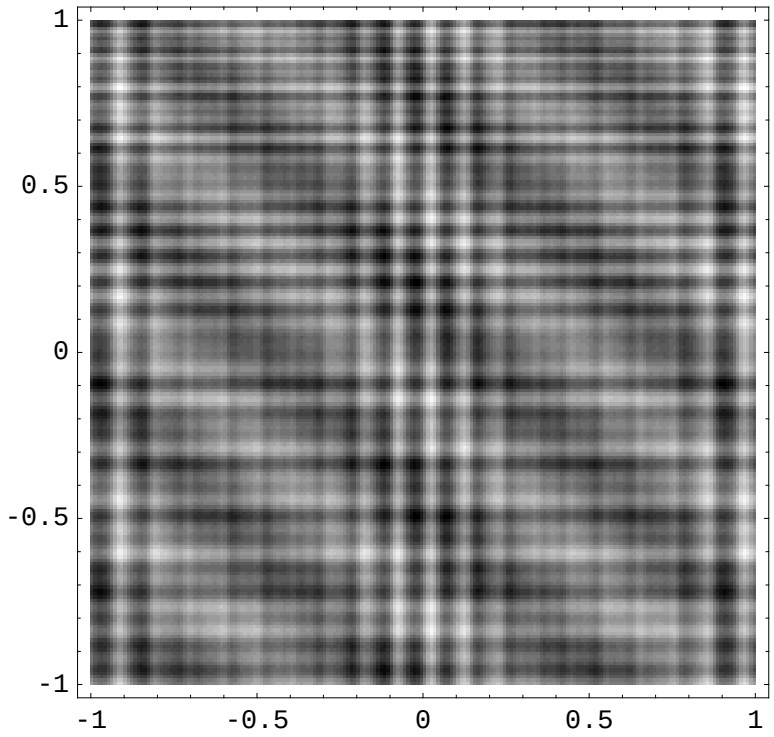
iniziale.

I metodi di ricerca del minimo globale si basano quasi tutti sulla ricerca di una direzione verso cui andare a cercare il minimo, ripetendo il procedimento iterativamente fino a quando il risultato non rientra nella tolleranza cercata: ovviamente, per funzioni particolarmente 'tortuose', la ricerca del giusto cammino può diventare qualcosa di veramente difficile per qualsiasi algoritmo.

Supponiamo di avere una funzione di questo tipo:

In[25]:=
```
f[x_, y_] := Exp[Sin[60 x]] + Sin[60 ℯ^y] +
  Sin[70 Sin[x]] + Sin[Sin[80 y]] - Sin[10 (x + y)] + 1/4 (x^2 + y^2)
```

Il grafico di questa funzione è il seguente, riportato in maniera bidimenisionale:



Come potete vedere, di massimi e di minimi ce n'è a iosa, e non è facile andare a trovare il minimo. Proviamo con Minimize:

In[26]:=
```
Minimize[f[x, y], {x, y}]
```

Out[26]=
```
Minimize[ℯ^Sin[60 x] + 1/4 (x^2 + y^2) + Sin[60 ℯ^y] -
  Sin[10 (x + y)] + Sin[70 Sin[x]] + Sin[Sin[80 y]], {x, y}]
```

In questo caso, non abbiamo ottenuto un bel niente... Proviamo a risolverlo numericamente:

```
In[27]:= NMinimize[f[x, y], {x, y}]
Out[27]= {-2.01986, {x → -1.48893, y → 0.291833}}
```

Abbiamo sicuramento trovato un minimo, ma non abbiamo la certezza, in questo caso, di aver trovato proprio quello globale, per il semplice fatto che la funzione è fortemente non quadratica e lineare, e quindi i metodi numerici hanno notevoli difficoltà a trovarlo. Effettivamente, ci sono punti che hanno un valore inferiore, segno che le coordinate trovate non corrispondono ad un minimo:

```
In[28]:= f[-0.024, 0.21]
Out[28]= -3.32725
```

I metodi numerici devono quindi essere sempre esaminati con attenzione, come d'altronde capita in tutti gli algoritmi: utilizzarli alla cieca non serve a niente, e crea solo danno. Bisogna saper bene quello che si sta facendo, e conoscere i limiti ed i vantaggi degli algoritmi che si utilizzano, perchè possono portare, come in questo caso, a risultati errati che possono essere considerati giusti se non ci si pone la giusta attenzione. Bisogna anche considerare che funzioni realmente complicate come questa, nella pratica non saranno incontrate molto spesso; per le funzioni che possiamo definire 'normali' (per esempio senza 4000 minimi nell'intervallo (0,1)), il comando funziona egregiamente. Soprattutto se abbiamo comunque un'idea di dove si dovrebbe trovare il minimo globale. Tutti i metodi di ricerca del minimo, infatti, lavorano meglio se il punto iniziale della ricerca si trova nell'intorno di quest'ultimo. Comunque vedremo meglio questo problema in una delle appendici di questo tutorial.

- **Dati numerici**

Abbiamo visto fino ad adesso come trattare numericamente le funzioni. Tuttavia, ci sono casi in cui ciò non è possibile. Si pensi, per esempio, ai dati sperimentali di qualche esperienza: in questo caso non abbiamo delle funzioni, ma solamente una lista di valori, un campionamento, la variazione di un parametro del transistor a variare della temperatura e così via: dobbiamo poter trattare anche questo tipo di dati, elaborarli ed in genere fare con loro quello che vogliamo.

Un'operazione che possiamo effetutare su questi valori, è trovare la funzione che li approssima meglio; per poter ottenere questo risultato *Mathematica* ci viene in aiuto con i suoi comandi di fitting:

`Fit[{`y_1`, `y_2`, … }, {`f_1`, `f_2`, … }, `x`]`	esegue il fitting dei valori y_n con una combinazione lineare delle funzioni f_i
`Fit[{{`x_1`, `y_1`}, {`x_2`, `y_2`}, … }, {`f_1`, `f_2`, … }, `x`]`	lo stesso di prima, ma per funzioni e valori multidimensionali.

Adesso, creeremo con Table una lista di valori, e poi cerceremo di effettuare il fitting, usando il metodo dei minimi quadrati:

```
In[29]:= dati = Table[Exp[x / 5.] + Random[], {x, 10}];
```

Vediamo adesso di sfruttare il comando appena appreso per scrivere il fitting di questi dati usando, per esempio, una funzione quadratica, quindi combinazione lineare delle funzioni 1, x, x^2:

```
In[30]:= Fit[dati, {1, x, x^2}, x]

Out[30]= 2.21615 - 0.265065 x + 0.08228 x^2
```

Si vede come abbiano effettuato il fitting dei dati. Naturalmente potevamo avere anche 10000000 punti, e *Mathematica* avrebbe effettuato il fitting con la stessa facilità.

Oltre a questo, possiamo scrivere direttamente un'equazione parametrizzata, specificare i parametri e il programma, invece di effettuare una combinazione lineare, aggiusta i parametri in modo di effettuare il fitting dei dati:

`FindFit[`*data*`,` *form*`, {`p_1`,` p_2`,` … `},` x`]`	trova il fitting *form* con i parametri p_i

Possiamo fare lo stesso esempio di sopra, scrivendo l'equazione quadratica parametrizzata e verificando se il risultato è uguale a quello che abbiamo trovato in precedenza:

```
In[31]:= FindFit[dati, a x^2 + b x + c, {a, b, c}, x]

Out[31]= {a → 0.08228, b → -0.265065, c → 2.21615}
```

```
In[32]:= a x^2 + b x + c /. %

Out[32]= 2.21615 - 0.265065 x + 0.08228 x^2
```

Possiamo vedere come l'equazione ottenuta sia effettivamente la stessa della precedente. Tuttavia, dato che qua si parla di parametri e non di combinazione lineare, possiamo effettuare anche fitting non lineari:

```
In[33]:= FindFit[dati, a + b x + Sin[Sqrt[c x]], {a, b, c}, x]

Out[33]= {a → 0.131899, b → 0.757938, c → 3.73583}
```

Abbiamo quindi visto come è possibile non solo elaborare funzioni con *Mathematica*, ma anche dati. Ovviamente non abbiamo neanche scalfito le potenzialità del programma. Le funzionalità vanno ben oltre, come calcolare in pochi secondi trasformate numeriche di Fourier di milioni di elementi, oppure effettuare calcoli statistici sui dati, come il valor medio con la funizione Mean, la mediana con Median, la deviazione standard con (indovinate?) StandardDeviation e così via.

Non ho avuto molto a che fare, sinceramente, con le funzioni riguardanti la statistica... Probabilmente, appena comincerò la tesi e dovrò elaborare le misure degli esperimenti, vedrò meglio questo concetto. Comunque posso vedere che le funzioni di *Mathematica* dedicate alla statistica non sono numerose come negli altri campi. Per questo, negli Extra packages dell'installazione standard ve ne sono numerosi riguardanti la statistica. Se vi serve qualcosa di avanzato, è probabile che prima dobbiate caricare quello che vi interessa. Andate a spulicarli, se vi serve, e decidete quale usare di volta in volta. Se li usate spesso, però, nelle opzioni del programma potete decidere di caricarli all'avvio, per cui vedete un po' voi.

- **Precisione**

Abbiamo visto come è facile risolvere problemi numerici con funzioni così potenti. Gli esempi che vi ho fatto vedere non devono trarre in inganno. Ho usato sempre funzioni semplici, ma questo per farvi capire il contetto, e perchè funzioni troppo complicate potrebbero distrarre da quello che è il funzionamento del programma. Però vi posso assicurare che *Mathematica* digerisce veramente di tutto.

Quello che ancora non avete visto, però, è come possiamo usare la precisione arbitraria. Le funzioni che abbiamo utilizzato finora, infatti, usano la precisione di macchina. Se vogliamo invece ottenere un numero superiore di cifre significative, dobbiamo specificare al programma che ne abbiamo bisogno. In questo modo dobbiamo un pochino giocare con le opzioni delle funzioni, per specificare come giostrarci la situazione.

`WorkingPrecision`	numero di cifre da usare nel calcolo
`PrecisionGoal`	numero di cifre della precisione che si cerca di raggiungere
`AccuracyGoal`	numero di cifre dell'accuratezza che si cerca di raggiungere

Una delle opzioni che vengono usate in questo caso è WorkingPrecision. Precisamente, se si pone uguale ad n, si forza il comando ad utilizzare n cifre significative per l'algoritmo:

```
In[34]:= NIntegrate[Sin[Log[x]], {x, 5, 9}, WorkingPrecision → 70]
Out[34]= 3.689003334597008878366246602388423393856254806708080513846498696
```

Come possiamo vedere in questo caso, abbiamo ottenuto la precisione richiesta. Per default, il parametro è fissato alla costante MachinePrecision, che dice in pratica di utilizzare la precisione macchina su cui si lavora. Tuttavia, sappiamo che se effettuiamo un calcolo numerico con una precisione di n cifre, le ultime di solito non sono significative, per gli errori di troncamento che si hanno, e che abbiamo sicuramente tutti quanti studiato a calcolo numerico. Per vedere la precisione,

quindi il numero di cifre significative, del risultato che abbiamo ottenuto, possiamo usare il comando Accuracy:

```
In[35]:= Accuracy[%]

Out[35]= 63.0291
```

Tuttavia, una volta specificato questo parametro, possiamo giocare definendo all'interno di questo range il numero di cifre significative ed il numero di cifre di accuratezza, perchè WorkingPrecision definisce solamente un limite superiore. Questo perchè, per ragioni teoriche sulla convergenza, anche poche cifre significative in più possono aumentare in modo notevole il tempo di calcolo. Tuttavia, in molti casi è necessario sapere con precisione l'accuratezza con cui troviamo la soluzione numerica

```
In[36]:= NIntegrate[Sin[Log[x]], {x, 5, 9},
          WorkingPrecision → 70, PrecisionGoal → 50]

Out[36]= 3.689003334597008878366246602388423393856256480670808

In[37]:= Accuracy[%]

Out[37]= 50.5546
```

Vediamo che adesso, anche se abbiamo utilizzato una precisione di 70 cifre, la precisione richiesta è soltanto di 50 cifre, e l'accuratezza è rimasta tale:

```
In[38]:= NIntegrate[Sin[Log[x]], {x, 5, 9},
          WorkingPrecision → 70, AccuracyGoal → 68, PrecisionGoal → 30]

Out[38]= 3.689003334597008878366246602388423393856

In[39]:= Accuracy[%]

Out[39]= 39.2759
```

Possiamo vedere che, settando entrambi i valori, il calcolo si ferma una volta raggiunta la condizione che restituisce il numero inferiore di cifre, perchè si è raggiunta una condizione. Tuttavia dobbiamo notare come tutte le considerazioni vadano considerate sempre all'interno di WorkingPrecision:

```
In[40]:= NIntegrate[Sin[Log[x]], {x, 5, 9},
          WorkingPrecision → 30, PrecisionGoal → 50]

— NIntegrate::tmap :
   NIntegrate is unable to achieve the tolerances specified by the
     PrecisionGoal and AccuracyGoal options because the working precision is
     insufficient.  Try increasing the setting of the WorkingPrecision option.

Out[40]= 3.689003334597008878366246602 39

In[41]:= Accuracy[%]

Out[41]= 29.1321
```

In questo caso si vede che abbiamo cercato di raggiungere una precisione con un numero di cifre superiore rispetto a quelle usate per il calcolo, cosa che ovviamente è senza senso, e *Mathematica* si ferma al numero di cifre utilizzate per il calcolo, segnando questa incongruenza con un warning. In teoria non possiamo dire niente riguardo il risltato ottenuto, perchè si è interrotto prima di verificare la convergenza, ma possiamo vedere come il risultato ottenuto abbia la massima precisione possibile per il numero di cifre che abbiamo preso in esame. Tuttavia questa non è una regola, e il consiglio è di variare le opzioni in modo che nel risultato non compaia nessun warning. AccuracyGoal e PrecisionGoal sembrano fare la stessa cosa, ma in realtà hanno significato leggermente diverso a seconda del tipo di comando che si utilizza, cone ad esempio NIntegral piuttosto che NDSolve.

```
In[42]:= Options[NDSolve]

Out[42]= {AccuracyGoal → Automatic, Compiled → True,
          DependentVariables → Automatic, EvaluationMonitor → None,
          MaxStepFraction → 1/10, MaxSteps → Automatic,
          MaxStepSize → Automatic, Method → Automatic,
          NormFunction → Automatic, PrecisionGoal → Automatic,
          SolveDelayed → False, StartingStepSize → Automatic,
          StepMonitor → None, WorkingPrecision → MachinePrecision}

In[43]:= Options[NIntegrate]

Out[43]= {AccuracyGoal → ∞, Compiled → True, EvaluationMonitor → None,
          GaussPoints → Automatic, MaxPoints → Automatic, MaxRecursion → 6,
          Method → Automatic, MinRecursion → 0, PrecisionGoal → Automatic,
          SingularityDepth → 4, WorkingPrecision → MachinePrecision}
```

Options è una funzione che serve per vedere tutte le opzioni di una funzione, in modo da sapere quali sono. Possiamo vedere come nelle due funzioni prese in considerazione, di default si sceglie WorkingPrecision proprio uguale a MachinePrecision, ma come, per esempio, abbiano un valore diverso di AccuracyGoal, perchè la natura interna delle funzioni preferisce lavorare su una opzione piùttosto che su un altra. In particolare, il valore ∞ indica di considerare solo se sono soddisfatte le

condizioni dell'altra opzione: in NIntegrate accuracyGoal è settata su ∞, per cui le condizioni di terminazione dell'algoritmo sono date esclusivamente da PrecisionGoal. Tuttavia, nei casi normali, un semplice cambiamenti di WorkingPrecision può bastare, dato che in questo modo le altre due sono risettate automaticamente, dato che Automatic significa il loro valore si modifica automaticamente al variare di WorkingPrecision.