Packages

Vediamo adesso alcuni packages, predefiniti in *Mathematica*, che potrebbero essere particolarmente utili per risolvere particolari problemi (per esempio, disegnare diseguaglianze, o tracciare diagrammi di Bode, che hanno scale logaritmiche).

Di solito i packages vengono trascurati, e ci si concentra solamente sulle funzioni predefinite in *Mathematica*. Questo di certo non è male, per chi ci mette mano per la prima volta, tuttavia trascurare i package secondo me rappresenta un grave errore; si pensa che si tratti solamente di comandi aggiuntivi di poca importanza, e che la sostanza è in *Mathematica* stessa. Questo è vero e falso allo stesso tempo: vero perchè i packages sono scritti nel linguaggio standard di *Mathematica*, e quello che fanno lo potete fare anche voi definendo le vostre funzioni: falso perchè per implementare una funzione di quelle definite nei packages occorrerebbero pagine e pagine di codice, che fortunatamente qualcun altro ha già scritto per noi. Non si tratta solamente di orpelli, ma ci sono molti comandi che possono veramente semplificare la vita di chi utilizza *Mathematica*. Sono certo che fra i vari package ce ne saranno almeno un paio che troverete invitanti, e che non abbandonerete. Capisco che la prima cosa che si deve fare è imparare ad usare il programma ma, una volta imparato, imparare i comandi dei package (mica tutti: l'help serve anche per trovare quello che cercate in un dato momento), vi permette di fare cose che altrimenti, diciamocela tutta, non sapreste fare.

Non li elencherò tutti, perchè sono troppi, ma quelli che reputo utili (o che almeno mi sono serviti) sono tutti qua; considero anche il fatto che, pur essendo solamente alcuni, sono già parecchi, per cui avrete di che sperimentare ed imparare fino alla laurea ed oltre... Ovviamente appena finito di leggere queste pagine dovete correre a leggervi l'help per scoprire gli altri!!!!

Comunque, cominciamo:

■ Algebra`AlgebraicInequalities`

In[1]:=
```
<< Algebra`InequalitySolve`
```

Questo package contiene una funzione utilizzata per risolvere una disequazione (oppure un sistema di disequazioni):

`InequalitySolve[`*expr*`, `*x*`]`	trova i valori di *x* che soddisfano l'espressione contenente operatori logici e disuguaglianze algebriche
`InequalitySolve[`*expr*`, {`x_1`, `x_2`, … }]`	trova i valori di x_i che soddisfano le diseguaglianze

Supponiamo di avere la seguente espressione:

```
In[2]:= diseguaglianza = x^2 / (1 + x) < 5
```

$$\text{Out[2]=} \quad \frac{x^2}{1+x} < 5$$

```
In[3]:= InequalitySolve[diseguaglianza, x]
```

$$\text{Out[3]=} \quad x < -1 \;||\; \frac{1}{2}\left(5 - 3\sqrt{5}\right) < x < \frac{1}{2}\left(5 + 3\sqrt{5}\right)$$

Come potete vedere, abbiamo ottenuto la soluzione che volevamo.

Un sistema di diseguaglianze viene scritto come un'unica equazione, dove le varie parti del sistema sono legate fra di loro dall'operatore AND:

```
In[4]:= sistema =
         y / x > (4 x^3 + y) / (x - y) && x < y^2
```

$$\text{Out[4]=} \quad \frac{y}{x} > \frac{4x^3 + y}{x - y} \;\&\&\; x < y^2$$

```
In[5]:= InequalitySolve[sistema, {x, y}]
```

$$\text{Out[5]=} \quad (x < 0 \;\&\&\; y < x) \;||\; (0 < x \leq 1 \;\&\&\; y > \sqrt{x}) \;||\; (x > 1 \;\&\&\; y > x)$$

Abbiamo utilizzato le soluzioni desiderate, sotto forma di condizioni per le incognite.

Come potete vedere, questo comando è molto potente, anche se funziona soltanto con equazioni polinomiali:

```
In[6]:= InequalitySolve[Exp[x] < 1 + x, x]

  - InequalitySolve::npi : A nonpolynomial equation
       or inequality encountered. The solution set may be incorrect.

Out[6]= False
```

Come potete vedere, non possiamo ottenere gli intervalli, in questo caso.

■ Algebra`ReIm`

Questo package permette di lavorare meglio con funzioni complesse, estraendone parte reale ed immaginaria senza dover utilizzare ComplexExpand.

Vediamo, per esempio, questa espressione:

```
In[7]:= Re[x / y]
```

$$\text{Out[7]= } \text{Re}\left[\frac{x}{y}\right]$$

Vediamo che rimane non valutata. Andiamo, adesso, a caricare il package:

```
In[8]:= << Algebra`ReIm`
```

```
In[9]:= Re[x / y]
```

$$\text{Out[9]= } \frac{\text{Im}[x]\,\text{Im}[y]}{\text{Im}[y]^2+\text{Re}[y]^2}+\frac{\text{Re}[x]\,\text{Re}[y]}{\text{Im}[y]^2+\text{Re}[y]^2}$$

Come si vede, questa volta *Mathematica* esplicita direttamente la parte reale ed immaginaria dei simboli, evitando comandi come ComplexExpand. Tuttavia, se specifichiamo un simbolo come reale o come immaginario, senza dover specificare il simbolo esplicitamente:

```
In[10]:= y /: Re[y] = 0;
```

```
In[11]:= Re[x / y]
```

$$\text{Out[11]= } \frac{\text{Im}[x]}{\text{Im}[y]}$$

Come vedere, con il package abbiamo calcolato il valore considerando l'assunzione che avevamo fatto.

■ Algebra`RootIsolation`

Con questo package possiamo avere funzioni per gestire meglio il conteggio e l'isolamento di soluzioni di equazioni polinomiali:

`CountRoots[`*poly*`, {`*x*`,` m_1`,` m_2`}]`	restituisce il numero delle soluzioni di *poly* nell'intervallo complesso $\{m_1, m_2\}$
`RealRootIntervals[`*poly*`]`	restituisce gli intervalli disgiunti che isolano le radici reali di *poly*
`RealRootIntervals[{`$poly_1$`,` $poly_2$`, … }]`	restituisce gli intervalli disgiunti che isolani le radici reali delle equazioni $poly_1$, $poly_2$, …
`ComplexRootIntervals[`*poly*`]`	restituisce gli intervalli disgiunti che isolano le soluzioni complesse di *poly*
`ComplexRootIntervals[{`$poly_1$`,` $poly_2$`, … }]`	indovinate cosa fa??? Per i polinomi $poly_1$, $poly_2$, …
`ContractInterval[`*a*`,` *n*`]`	restituisce un intervallo isolante che circonda il valore del numero algebrico *a* fino alla precisione di almeno *n* cifre significative

Si possono eseguire delle operazioni sui polinomi e scoprire proprietà sulle loro radici senza bisogno di andare a trovarle esplicitamente. Per esempio:

In[12]:=
```
<< Algebra`RootIsolation`
```

In[13]:=
```
CountRoots[(x - 4) (x^4 + 3 x - 6), {x, -5, 5}]
```

Out[13]= 3

Abbiamo visto che nell'intervallo specificato si trovano tre soluzioni, senza averle calcolate:

In[14]:=
```
CountRoots[x^9 + 1, {x, 0, 1 + I}]
```

Out[14]= 2

Come vedete, funziona anche per intervalli complessi.

Consideriamo ancora la prima espressione, e fediamo gli intervalli delle soluzioni reali:

In[15]:=
```
RealRootIntervals[(-4 + x) (-6 + 3 x + x^4)]
```

Out[15]=
```
{{-3, 0}, {1, 2}, {2, 6}}
```

Questi sono, per l'appunto, intervalli disgiunti, ognuno contentente una radice reale del polinomio. Analogamente possiamo realizzarli per le soluzioni immaginarie:

In[16]:=
```
ComplexRootIntervals[x^9 + 1]
```

Out[16]=
$$\left\{\{-2, 0\}, \left\{-\frac{3}{2} - \frac{3\,i}{2}, -\frac{3}{4}\right\}, \left\{-\frac{3}{2}, -\frac{3}{4} + \frac{3\,i}{2}\right\}, \left\{-\frac{3}{4} - \frac{3\,i}{2}, 0\right\}, \left\{-\frac{3}{4}, \frac{3\,i}{2}\right\}, \left\{-\frac{3\,i}{2}, \frac{3}{4}\right\}, \left\{0, \frac{3}{4} + \frac{3\,i}{2}\right\}, \left\{\frac{3}{4} - \frac{3\,i}{2}, \frac{3}{2}\right\}, \left\{\frac{3}{4}, \frac{3}{2} + \frac{3\,i}{2}\right\}\right\}$$

Possiamo anche specificare un intervallo piccolo a piacere per una soluzione:

In[17]:=
```
ContractInterval[Root[x^9 + 1, 2], 20]
```

Out[17]=
$$\left\{-\frac{56524103165212567109}{73786976294838206464} - \frac{189717416474225563087\,i}{295147905179352825856}, -\frac{45219282532170053687 1}{59029581035870565 1712} - \frac{37943483294845112 6173\,i}{59029581035870565 1712}\right\}$$

In[18]:=
```
N[%, 20]
```

Out[18]=
$$\{-0.76604444311897803520 - 0.64278760968653932632\,i, -0.76604444311897803520 - 0.64278760968653932632\,i\}$$

Come potete vedere, possiamo ottenere l'approssimazione che vogliamo.

■ Calculus`FourierTransform`

Questo package contiene alcune funzioni che sono l'equivalente numerico dei comandi per la trasformata di Fourier predefinite in *Mathematica*:

`NFourierTransform[`*expr*`, t, ω]`	restituisce un approssimazione numerica della trasformata di Fourier di *expr* valutata nel valore numerico *ω*, mentr *expr* è considerata funzione di *t*
`NInverseFourierTransform[`*expr*`, ω, t]`	restituisce un approssimazione numerica della trasformata inversa di Fourier di *expr* valutata nel valore numerico *t*, mentr *expr* è considerata funzione di *ω*
`NFourierSinTransform[`*expr*`, t, ω]`	restituisce un approssimazione numerica della trasformata in seno di Fourier di *expr* valutata nel valore numerico *ω*, mentr *expr* è considerata funzione di *t*
`NInverseFourierSinTransform[`*expr*`, ω, t]`	frestituisce un approssimazione numerica della trasformata inversa in seno di Fourier di *expr* valutata nel valore numerico *t*, mentr *expr* è considerata funzione di
`NFourierCosTransform[`*expr*`, t, ω]`	restituisce un approssimazione numerica della trasformata in coseno di Fourier di *expr* valutata nel valore numerico *ω*, mentr *expr* è considerata funzione di
`NInverseFourierCosTransform[`*expr*`, ω, t]`	restituisce un approssimazione numerica della trasformata inversa in cosenodi Fourier di *expr* valutata nel valore numerico *t*, mentr *expr* è considerata funzione

Supponiamo di avere la seguente espressione:

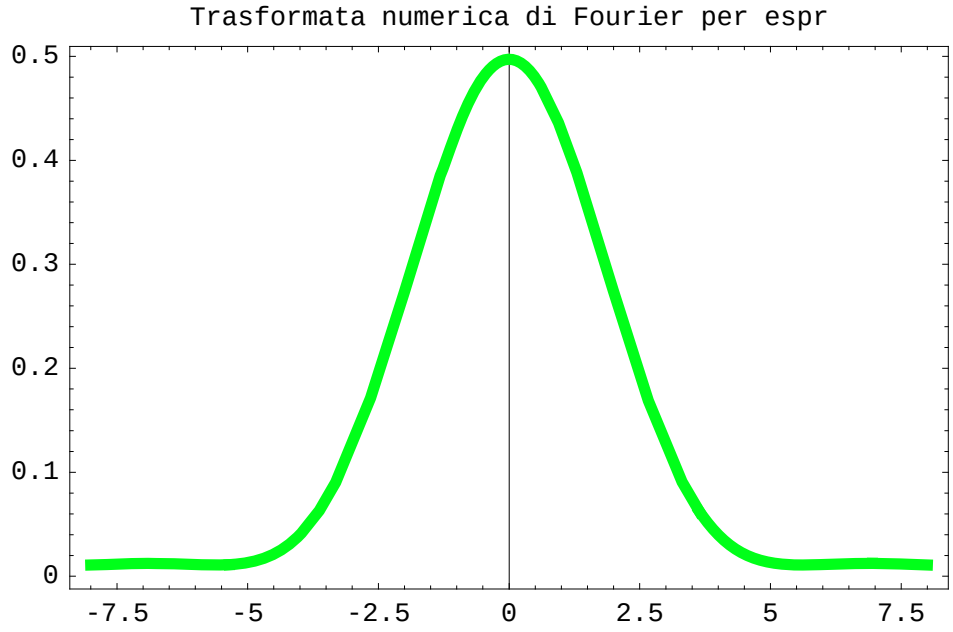
```
In[19]:= espr = Exp[-Abs[t^3]] / (1 + Abs[t]);

In[20]:= Timing[FourierTransform[espr, t, w]]

Out[20]= {21.422 Second, FourierTransform[e^-Abs[t]^3 / (1 + Abs[t]), t, w]}
```

Come potete vedere, in questo caso non è in grado di calcolare la trasformata di questa funzione, fra l'altro dopo un tempo neanche breve per il programma. Allora, se ci interessa comunque la trasformata, per esempio in $\omega = 10$, possiamo utilizzare la funzione del packge:

```
In[21]:= << Calculus`FourierTransform`

In[22]:= Timing[NFourierTransform[espr, t, 10]]

Out[22]= {0.016 Second, 0.00716034 + 0. i}
```

Come potete vedere, non solo abbiamo trovato la soluzione che ci interessava, ma abbiamo anche risparmiato un sacco di tempo...

In[23]:=
```
Plot[
 Evaluate[Abs[NFourierTransform[espr, t, ω]]], {ω, -8, 8},
 PlotStyle → {Hue[0.35], Thickness[0.013]},
 Frame → True,
 PlotLabel → "Trasformata numerica di Fourier per espr"
]
```



Out[23]= `- Graphics -`

Come potete vedere, non è poi molto importante il non poter calcolare analiticamente la funzione, a meno che non dobbiate scriverlo sul quaderno, ovvio...

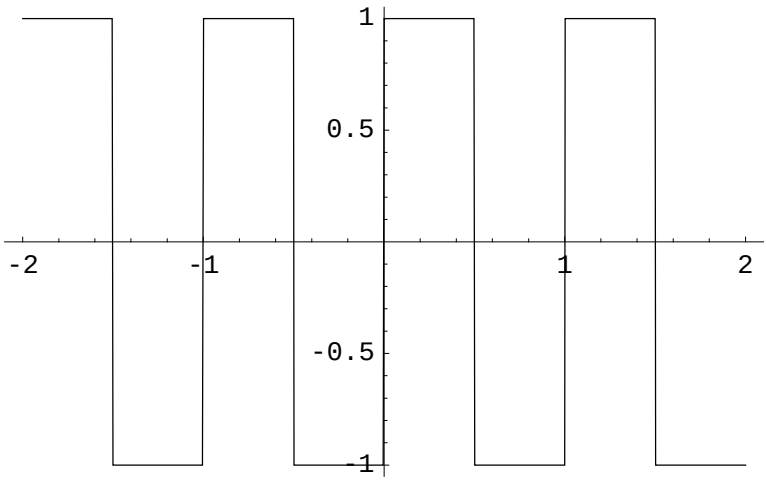
Il vantagigo di questo comando, rispetto alla trasformata numerica predefinita in *Mathematica*, è che, mentre quest'ultima lavora solamente con liste di dati, quella definira nel package invece lavora su funzioni, il che specialmente in calcoli teorici è un bel vantaggio.

Oltre che per la trasformata generale, il pacchetto offre anche funzioni per lavorare con le funzioni periodiche e con la serie di Fourier:

`FourierCoefficient[`*expr*`, `*t*`, `*n*`]`	restituisce l' *n–simo* coefficiente nell'espansione in serie esponenziale della funzione periodica in *t* uguale ad *expr* nell'intervallo $t=-\frac{1}{2}$, $t=\frac{1}{2}$
`FourierSinCoefficient[`*expr*`, `*t*`, `*n*`]`	restituisce l' *n–simo* coefficiente nell'espansione in serie del seno
`FourierCosCoefficient[`*expr*`, `*t*`, `*n*`]`	restituisce l' *n–simo* coefficiente nell'espansione in serie del coseno
`FourierSeries[`*expr*`, `*t*`, `*k*`]`	restituisce la serie esponenziale fino all'ordine *k* della funzione periodica in *t* uguale ad *expr* nell'intervallo $t=-\frac{1}{2}$, $t=\frac{1}{2}$
`FourierTrigSeries[`*expr*`, `*t*`, `*k*`]`	restituisce l'espansione in serie trigonometrica fino all'ordine *k*

Supponiamo di avere la classica onda quadra:

In[24]:=
```
p1 = Plot[(x - Round[x]) / Abs[x - Round[x]], {x, -2, 2}]
```



Out[24]=
```
- Graphics -
```

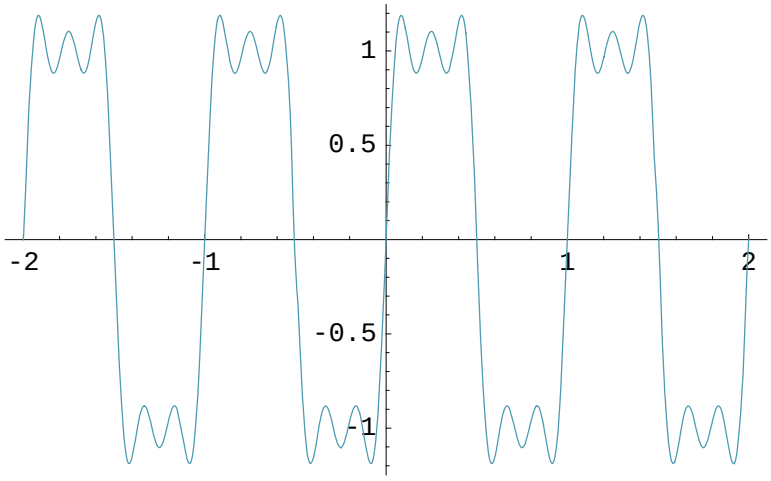
Sviluppiamo adesso i primi quattro termini della serie di Fourier: dato che l'intervallo preso in considerazione dal comando è compreso in $(-\frac{1}{2}, \frac{1}{2})$, non abbiamo bisogno di specificare il Round, e basta scrivere:

In[25]:=
```
FourierTrigSeries[Abs[t] / t, t, 5]
```

Out[25]=
$$\frac{4\,\text{Sin}[2\,\pi\,t]}{\pi} + \frac{4\,\text{Sin}[6\,\pi\,t]}{3\,\pi} + \frac{4\,\text{Sin}[10\,\pi\,t]}{5\,\pi}$$

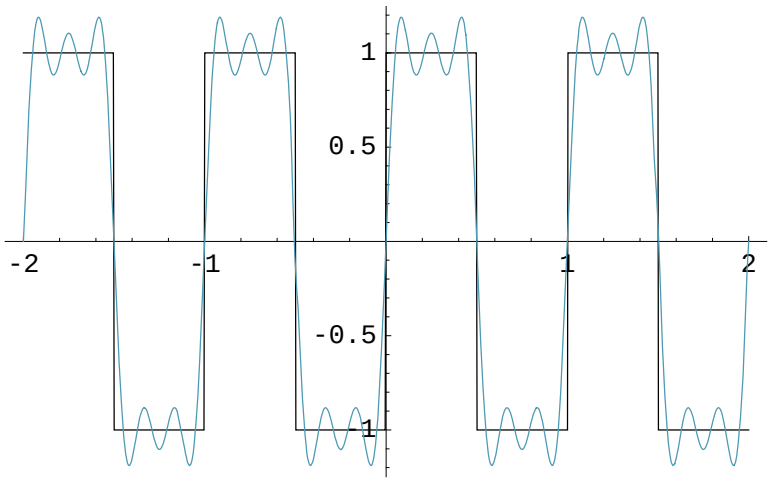
Vediamo che effettivamente rappresenta quello che volevamo:

In[26]:=
```
Plot[%, {t, -2, 2}, PlotStyle → {RGBColor[0.3, 0.6, 0.7]}]
```



Out[26]=
```
- Graphics -
```

In[27]:=
```
Show[p1, %]
```



Out[27]=
```
- Graphics -
```

Come potete vedere, l'espansione trigonometrica (troncata) rappresenta effettivamente l'espansione in serie di Fourier. Possiamo anche trovarci la serie esponenziale, se vogliamo:

In[28]:=
```
FourierSeries[Abs[t] / t, t, 5]
```

Out[28]=
$$\frac{2\,i\,e^{-2\,i\,\pi\,t}}{\pi} - \frac{2\,i\,e^{2\,i\,\pi\,t}}{\pi} + \frac{2\,i\,e^{-6\,i\,\pi\,t}}{3\,\pi} - \frac{2\,i\,e^{6\,i\,\pi\,t}}{3\,\pi} + \frac{2\,i\,e^{-10\,i\,\pi\,t}}{5\,\pi} - \frac{2\,i\,e^{10\,i\,\pi\,t}}{5\,\pi}$$

Naturalmente, possono capitare casi in cui non è possibile risolvere simbolicamente l'espansione in serie:

```
In[29]:= Clear[espr]

In[30]:= espr[t_] := Cos[Cos[t]]

In[31]:= FourierTrigSeries[espr[t], t, 3]
```

$$\text{Out[31]= } \int_{-\frac{1}{2}}^{\frac{1}{2}} \text{Cos}[\text{Cos}[t]]\, dt + 2\,\text{Cos}[2\pi t] \int_{-\frac{1}{2}}^{\frac{1}{2}} \text{Cos}[2\pi t]\,\text{Cos}[\text{Cos}[t]]\, dt + 2\,\text{Cos}[4\pi t] \int_{-\frac{1}{2}}^{\frac{1}{2}} \text{Cos}[4\pi t]\,\text{Cos}[\text{Cos}[t]]\, dt + 2\,\text{Cos}[6\pi t] \int_{-\frac{1}{2}}^{\frac{1}{2}} \text{Cos}[6\pi t]\,\text{Cos}[\text{Cos}[t]]\, dt$$

Possiamo comunque ottenere la serie con coefficienti numerici, se non riusciamo a trovare i sombolici:

`NFourierCoefficient[`*expr*`, `*t*`, `*n*`]`	restituisce l'approssimazione numerica dell' *n–simo* coefficiente nell'espansione in serie
`NFourierSinCoefficient[`*expr*`, `*t*`, `*n*`]`	restituisce l'approssimazione numerica dell' *n–simo* coefficiente nell'espansione in serie seno
`NFourierCosCoefficient[`*expr*`, `*t*`, `*n*`]`	restituisce *l'* approssimazione numerica dell' *n–simo* coefficiente nell' espansione in serie coseno
`NFourierSeries[`*expr*`, `*t*`, `*k*`]`	trova l'espansione in serie esponenziale, di ordine *k*, con coefficienti numerici
`NFourierTrigSeries[`*expr*`, `*t*`, `*k*`]`	trova l'espansione in serie trigonometrica, di ordine *k*, con coefficienti numerici

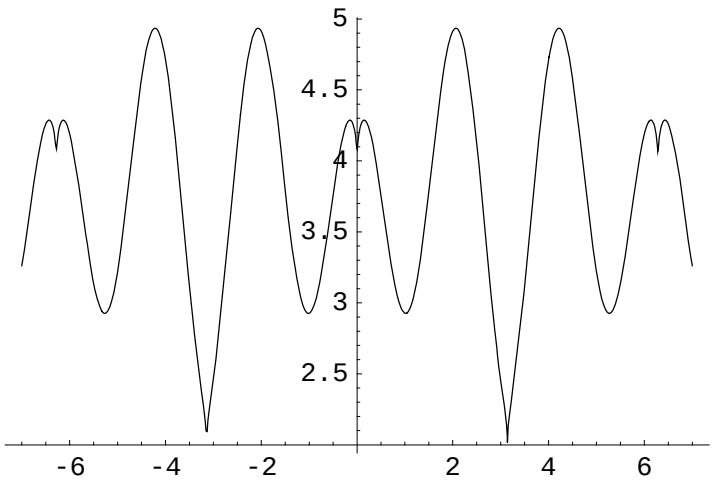
Vediamo di risolvere il problema di poco fa in termini numerici:

```
In[32]:= NFourierTrigSeries[espr[t], t, 3]

Out[32]= 0.574072 - 0.0405754 Cos[2 π t] + 0.00954455 Cos[4 π t] -
          0.00419166 Cos[6 π t] + 0. Sin[2 π t] + 0. Sin[4 π t] + 0. Sin[6 π t]
```

Notate anche che, se avete espressioni comunque convertibili, nella stragrande maggioranza dei casi la soluzione numerica si trova MOLTO più velocemente di quella simbolica, per cui, per espressioni complicate, usatela solo se vi serve veramente.

Inoltre, potremmo avere funzioni periodiche, ma con un periodo diverso da 1, per cui il periodo non sarà più contenuto in $(-\frac{1}{2}, \frac{1}{2})$. Per ovviare a ciò, dobbiamo modificare i parametri di Fourier. Per esempio, se ho:

In[33]:=
```
(f = 3 + Sqrt[Abs[Sin[x]]] + Cos[3 x];
 Plot[f, {x, -7, 7}])
```



Out[33]= - Graphics -

In[34]:=
```
FindMinimum[f, {x, -4}]
```

```
— FindMinimum::lstol :
   The line search decreased the step size to within tolerance specified by
     AccuracyGoal and PrecisionGoal but was unable to find a sufficient
     decrease in the function.  You may need more than MachinePrecision
     digits of working precision to meet these tolerances. More…
```

Out[34]= {2.00009, {x → -3.14159}}

Possiamo vedere (nonostante il warning, dovuto al fatto che abbiamo un piccolo problema di convergenza, perchè il minimo è una cuspide), che il periodo è fra $-\pi$ e π, per cui dobbiamo far sapere al comando che questi sono i nuovi limiti; per far questo si utilizza l'opzione FourierParameters, modificando i parametri necessari:

common convention	setting	discrete Fourier	inverse discrete Fourier transform
Mathematica default	{0, 1}	$\frac{1}{n^{1/2}} \sum_{r=1}^{n} u_r \, e^{2\pi i (r-1)(s-1)/n}$	$\frac{1}{n^{1/2}} \sum_{s=1}^{n} v_s \, e^{-2\pi i (r-1)(s-1)/n}$
data analysis	{-1, 1}	$\frac{1}{n} \sum_{r=1}^{n} u_r \, e^{2\pi i (r-1)(s-1)/n}$	$\sum_{s=1}^{n} v_s \, e^{-2\pi i (r-1)(s-1)/n}$
signal processing	{1, -1}	$\sum_{r=1}^{n} u_r \, e^{-2\pi i (r-1)(s-1)/n}$	$\frac{1}{n} \sum_{s=1}^{n} v_s \, e^{2\pi i (r-1)(s-1)/n}$
general case	{a, b}	$\frac{1}{n^{(1-a)/2}} \sum_{r=1}^{n} u_r \, e^{2\pi i b (r-1)(s-1)/n}$	$\frac{1}{n^{(1+a)/2}} \sum_{s=1}^{n} v_s \, e^{-2\pi i b (r-1)(s-1)/n}$

setting	Fourier coefficient	inverse Fourier coefficient
{0, 1}	$\int_{-1/2}^{1/2} f(t) \, e^{2\pi i n t} \, dt$	$\sum_{n=-\infty}^{\infty} F_n \, e^{-2\pi i n t}$
{a, b}	$\lvert b\rvert^{(1-a)/2} \int_{-1/(2\lvert b\rvert)}^{1/(2\lvert b\rvert)} f(t) \, e^{2\pi i b n t} \, dt$	$\lvert b\rvert^{(1+a)/2} \sum_{n=-\infty}^{\infty} F_n \, e^{-2\pi i b n t}$

Nel nostro caso abbiamo:

In[35]:=

```
Chop[
 NFourierTrigSeries[f, x, 5,
  FourierParameters → {0, 2 Pi},
  AccuracyGoal → 8, PrecisionGoal → 8
 ]
]
```

Out[35]=

$$\sqrt{2\pi}\,(1.667 - 0.0315571\cos[4\pi^2 x] - 0.00730143\cos[8\pi^2 x] - 0.00562131\cos[12\pi^2 x] - 0.0027441\cos[16\pi^2 x] - 0.0025366\cos[20\pi^2 x])$$

Come potete vedere, abbiamo usato lo stesso comando, specificando però il periodo della funzione trigonometrica aggiustando i parametri tramite l'opzione apposita.

Inoltre, in questo package sono definite anche funzioni per la trasformata di Fourier di sequenze numeriche, quindi funzioni definite negli interi, per variabile discretizzata:

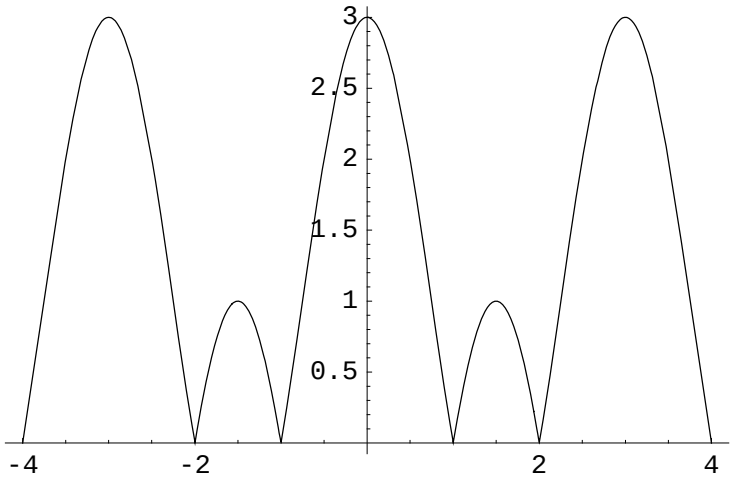
`DTFourierTransform[`*expr*`, `*n*`, `*omega*`]`	restituisce $F(\omega)$, una funzione periodica in ω, uguale alla sommatoria di Fourier di *expr*, considerata funzione degli interi *n*
`InverseDTFourierTransform[`*expr*`, `*omega*`, `*n*`]`	restituisce f_n, una funzione di interi *n* definita come inversa della sommatoria di Fourier $F(\omega)$, dove $F(\omega)$ è uguale ad *expr* nell'intervallo $\omega=\frac{-1}{2}$ to $\omega=\frac{1}{2}$
`NDTFourierTransform[`*expr*`, `*n*`, `*omega*`]`	trova un'approssimazione numerica per $F(\omega)$
`NInverseDTFourierTransform[`*expr*`, `*omega*`, `*n*`]`	trova un'approssimazione numerica per f_n

qua otteniamo una trasformata periodica di una sequenza

```
In[36]:= dtft = DTFourierTransform[Sum[DiscreteDelta[3 n + j], {j, -1, 1}], n, ω]
```

$$\text{Out[36]= } 1 + e^{-\frac{2}{3} i \pi \omega} + e^{\frac{2 i \pi \omega}{3}}$$

```
In[37]:= Plot[Abs[dtft], {ω, -4, 4}]
```




```
Out[37]= - Graphics -
```

Possiamo effettiare la trasformata inversa numerica:

```
In[38]:= Chop[
          Table[
           NInverseDTFourierTransform[
            dtft, ω, n, AccuracyGoal → 8, PrecisionGoal → 8
           ],
           {n, -5, 5}
          ]
         ]

Out[38]= {0.00738387, -0.0115663, 0.0206748, -0.0472568, 0.206748, 2.65399,
          0.206748, -0.0472568, 0.0206748, -0.0115663, 0.00738387}
```

Anche in questo caso, possiamo aggiustare i parametri:

setting	*discrete-time Fourier transform*	*inverse discrete-time Fourier coefficient*
{0, 1}	$\sum_{n=-\infty}^{\infty} f_n e^{2\pi i n \omega}$	$\int_{-1/2}^{1/2} F(\omega) e^{-2\pi i n \omega} d\omega$
{a, b}	$\lvert b\rvert^{(1-a)/2} \sum_{n=-\infty}^{\infty} f_n e^{2\pi i b n \omega}$	$\lvert b\rvert^{(1+a)/2} \int_{-1/(2\lvert b\rvert)}^{1/(2\lvert b\rvert)} F(\omega) e^{-2\pi i b n \omega} d\omega$

■ Calculus`VectorAnalysis`

Questo package è destinato a studiare sistemi e funzioni vettoriali, specialmente nel caso tridimensionale, permettendo di eseguire complicate analisi con semplici comandi.

Prima di tutto, dobbiamo considerare le funzioni che permettono di definire il tipo di sistemi di coordinate: oltre a quelle cartesiane, infatti, sono presenti altri tipi di coordinate, come quelle sferiche, che rappresentano meglio un determinato campo vettoriale perchè, data la sua simmetria, le equaizioni in quel sistema di coordinate appaiono semplici. Possiamo specificare sia il tipo di coordinate, sia il nome delle variabili che rappresentano le coordinate nelle tre direzioni (in senso generale, dato che non si parla più di assi a seconda del sistema di coordinate):

`CoordinateSystem`	restituisce il nome del sistema di coordinate corrente
`Coordinates[ ]`	restitusce le variabili di default che rappresentano l'attuale sistema di coordinate
`Coordinates[`*coordsys*`]`	restituisce i simboli per le coordinate di default del sistema di coordinate *coordsys*
`SetCoordinates[`*coordsys*`]`	pone come sistema di coordinate di default *coordsys* con le sue variabili di default
`SetCoordinates[`*coordsys*`[`*vars*`]]`	pone come sistema di coordinate di default *coordsys* con le variabili poste pari a *vars*

Per poter vedere come funziona questo package, ovviamente dobbiamo prima caricarlo...

In[39]:=
```
<< Calculus`VectorAnalysis`
```

A questo punto, vediamo il sistema di coordinate di default:

In[40]:=
```
CoordinateSystem
```

Out[40]=
```
Cartesian
```

In[41]:=
```
Coordinates[]
```

Out[41]=
```
{Xx, Yy, Zz}
```

Come potete vedere, di default il package pone il sistema di coordinate cartesiano. Possiamo assegnare vari tipi di coordinate:

Bipolar (bipolari)	Bispherical (bisferiche)
Cartesian (cartesiane)	ConfocalEllipsoidal (ellissoidali confocali)
ConfocalParaboloidal (paraboloidi confocali)	Conical (coniche)
Cylindrical (cilindriche)	EllipticCylindrical (cilindriche ellittiche)
OblateSpheroidal (sferiche schiacciate)	ParabolicCylindrical (cilindriche paraboloidi)
Paraboloidal (paraboloidi)	ProlateSpheroidal (sferiche allungate)
Spherical (sferiche)	Toroidal (toroidali)

Come potete vedere, ce n'è veramente per tutti i gusti... vediamo un poco di spiegare cosa rappresentano:

In[42]:=

- ⋆ **Cartesian[*x*, *y*, *z*]** rappresenta il sistema di coordinate standard rettangolare tridimensionale.
- ⋆ **Cylindrical[*r*, *theta*, *z*]** rappresenta il sistema di coordinate cilindrico, con coordinate polari r, θ per rappresentare il piano $x\,y$, e l'asse z a rappresentare l'altezza.
- ⋆ **Spherical[*r*, *theta*, *phi*]** rappresenta il sistema di coordinate sferico, dove r rappresenta la distanza dall'origine, θ rappresenta l'angolo misurato a partire dall'asse z positivo, e ϕ rappresenta l'angolo misurato nel piano $x\,y$, in senso antiorario a partire dall'asse x positivo.
- ⋆ **ParabolicCylindrical[*u*, *v*, *z*]** è un sistema di coordinate dove il piano $x\,y$ è rappresentato da u, v: mantenendo costante un parametro e variando l'altro si ottengono due parabole opposte, dove nel fuoco comune passa l'asse z

★ **Paraboloidal[*u*, *v*, *phi*]** rappresenta il sistema di coordinate paraboloico: come prima, variando uno fra u, v si ottengono due parabole opposte con fuoco comune. Inoltre, ϕ rappresenta l'angolo di rotazione di questo piano rispetto alla bisettrice delle due parabole.

★ **EllipticCylindrical[*u*, *v*, *z*, *a*]** è un sistema di coordinate tridimensionale, parametrizzato da a, e rappresenta un sistema di coordinate simile a quello cilindrico, dove però, mantenendo costante u, nel piano al variare di v si producono delle ellissi, mentre variando u si ottengono delle iperboli. In ambeue i casi il fuochi sono distanziati da $2\,a$, mentre z definisce la distanza dall'asse dei fuochi comuni. a di default assume valore pari ad 1.

★ **ProlateSpheroidal[*xi*, *eta*, *phi*, *a*]** è simile al caso precedente, ma il piano delle ellissi ruota, rispetto all'asse che connette i due fuochi, di un angolo ϕ. Anche in questo caso il valore di default per a è pari ad 1.

★ **OblateSpheroidal[*xi*, *eta*, *phi*, *a*]** è quasi analogo a prima, soltanto che la rotazione avviene rispetto all'asse perpendicolare a quello che connette i due fuochi.

★ **Bipolar[*u*, *v*, *z*, *a*]** è sempre un sistema di coordinate parametrizzato da a, ed è costruito attorno a due fuochi distanti $2\,a$. Con u costante si ottiene un fascio di circonferenze che passa per i due punti, mentre fissando v si ottengono delle famiglie di ellissi che degenerano in uno dei due fuochi. z rappresenta la distanza da questo piano. Indovinate qual'è il valore di default per a?

★ **Bispherical[*u*, *v*, *phi*, *a*]** differisce da quello precedente soltanto dal fatto che ϕ rappresenta l'angolo azimutale.

★ **Toroidal[*u*, *v*, *phi*, *a*]** rappresenta il sistema di coordinate toroidale, che si ottiene facendo ruotare le coordinate bipolari attraverso l'asse perpendicolare a quello che unisce i due fuochi, con ϕ che specifica la rotazione.

★ **Conical[*lambda*, *mu*, *nu*, *a*, *b*]** è un sistema parametrizzato sia da a che da b. Le superfici che si ottengono per un valore fissato di λ sono sfere centrate nell'origine, mentre fissando μ si ottengono dei coni che divergono dall'origine e si prolungano lungo l'asse z, e fissando invece v si ottengono coni che si aprono lungo l'asse y. I valori di default per i parametri sono $a = 1$, $b = 2$.

★ **ConfocalEllipsoidal[*lambda*, *mu*, *nu*, *a*, *b*, *c*]** contiene tre parametri a, b, c. Le superfici che si ottengono fissando λ sono ellissoidi; fissando μ si ottengono degli iperboloidi ad un foglio, mentre fissando v iperboloidi a due fogli. I valori di default per i paramentri sono $a = 3$, $b = 2$, $c = 1$.

★ **ConfocalParaboloidal[*lambda*, *mu*, *nu*, *a*, *b*]** è un sistema di coordinate che, tanto per cambiare, ha a e b come parametri. Fissando la variabile λ solo paraboloidi ellittici che si estendono nella direzione negativa dell'asse z. Fissando la coordinata μ si disegnano paraboloidi

iperbolici, ed infine, fissando la coordinata *v* si ottengono paraboloidi ellittici che si estendono nella direzione positiva dell'asse *z*. I valori di default sono $a = 2$, $b = 1$.

In[43]:=

Una volta fissato un sistema di coordinate, sono presenti alcune funzioni per poter vedere qualche informazioni sul range delle variabili e dei parametri, ovvero di quanto possono variare:

`CoordinateRanges[ ]`	restituisce gli intervalli entro cui ognuna delle coordinate del sistema di coordinate di default può
`Parameters[ ]`	restituisce una lista dei valori dei parametri di default per il sistema di coordinate corrente
`ParameterRanges[ ]`	restituisce gli intervalli entro i quali possono variare i parametri del sistema di coordinate corrente
`CoordinateRanges[`*coordsys*`]`, `Parameters[`*coordsys*`]`, `ParameterRanges[`*coordsys*`]`	analogo ai tre precedenti, ma invece che usare il sistema di coordinate di default spiega variabili e parametri del sistema di coordinate *coordsys*
`SetCoordinates[`*coordsys*`[`*vars*`, `*param*`]]`	pone come sistema di coordinate di default *coordsys* con le variabili *vars* e valori per i parametri *param*

Supponiamo di voler cambiare il nostro sistema di coordinate di default in sferico:

In[44]:=
```
SetCoordinates[Spherical]
```

Out[44]=
```
Spherical[Rr, Ttheta, Pphi]
```

Vediamo gli intervalli in cui si trovano le coordinate;

In[45]:=
```
CoordinateRanges[]
```

Out[45]=
```
{0 ≤ Rr < ∞, 0 ≤ Ttheta ≤ π, -π < Pphi ≤ π}
```

Vediamo che il raggio, per esempio, può essere solo positivo ed, in genere, che ci sono le restrizioni di questo sistema di coordinate.

Vediamo, adesso, il range entro cui possono variare i parametri del sistema ellissoidi confocali, ed i valori di default:

In[46]:=
```
ParameterRanges[ConfocalEllipsoidal]
```

Out[46]=
```
0 < #3 < #2 < #1 < ∞
```

In[47]:=
```
Parameters[ConfocalEllipsoidal]
```

Out[47]=
```
{3, 2, 1}
```

Come potete vedere, abbiamo una restrizione per quanto riguarda i parametri, che non possono essere scelti a caso. Tuttavia questo sono sicuro che non capiterà perchè, quando si decide di studiare un problema, si scegli accuratamente il sistema di coordinate, per cui sapreste sicuramente quello che state per fare, vero?

Per poter semplificarci la vita, ci sono comandi che permettono di passare da coordinate cartesiane a quelle del sistema di coordinate scelto, e viceversa, sia se quest'ultimo rappresenta il sistema di default, sia se è un altro:

`CoordinatesToCartesian[`*pt*`]`	restituisce le coordinate cartesiane di *pt*, dato nelle coordinate del sistema di coordinate di
`CoordinatesToCartesian[`*pt*`, `*coordsys*`]`	restituisce le coordinate cartesiane di *pt*, dato nelle coordinate del sistema di coordinate
`CoordinatesFromCartesian[`*pt*`]`	restituisce le coordinate nel sistema di coordinate di default di *pt*, con coordinate date in forma cartesiana
`CoordinatesFromCartesian[`*pt*`, `*coordsys*`]`	restituisce le coordinate nel sistema di coordinate *coordsys* di *pt*, con coordinate date in forma cartesiana

Vediamo un esempio di facile comprensione:

In[48]:=
```
CoordinatesToCartesian[{1, Pi / 2, Pi / 4}, Spherical]
```

Out[48]=
$$\left\{\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}, 0\right\}$$

Tuttavia, questa funzione vale pure per coordinate simboliche, per cui restituisce le formule di conversione:

In[49]:=
```
CoordinatesToCartesian[{λ, μ, ν}, ConfocalEllipsoidal]
```

Out[49]=
$$\left\{\frac{\sqrt{(9-\lambda)\,(9-\mu)\,(9-\nu)}}{2\sqrt{10}}, \frac{\sqrt{-(4-\lambda)\,(4-\mu)\,(4-\nu)}}{\sqrt{15}}, \frac{\sqrt{(1-\lambda)\,(1-\mu)\,(1-\nu)}}{2\sqrt{6}}\right\}$$

Sono comandi molto utili, che utlilizzerete abbastanza spesso, se utilizzerete questo package.

Inoltre, ci sono comandi per effettuare operazioni standard sui vettori tridimensionali anche per gli altri sistemi di coordinate:

`DotProduct[`v_1`, `v_2`]`	calcola il prodotto scalare di v_1 e v_2 nel sistema di coordinate di default
`CrossProduct[`v_1`, `v_2`]`	calcola il prodotto vettoriale nel sistema di coordinate corrente
`ScalarTripleProduct[`v_1`, `v_2`, `v_3`]`	calcola il triplo prodotto scalare
`DotProduct[`v_1`, `v_2`, `*coordsys*`]`, `CrossProduct[`v_1`, `v_2`, `*coordsys*`]`, etc.	restituiscono i risultati con i vettori scritti nel sistema di coordinate *coordsys*

Vediamo il prodotto scalare nel caso normale:

```
In[50]:= Clear[a, b, c, d, e, f]

In[51]:= DotProduct[{a, b, c}, {d, e, f}, Cartesian]

Out[51]= a d + b e + c f
```

Vediamo il prodotto con coordinate date in un altro sistema di coordinate:

```
In[52]:= DotProduct[{a, b, c}, {d, e, f}, ProlateSpheroidal]

Out[52]= Cos[b] Cos[e] Cosh[a] Cosh[d] +
          Cos[c] Cos[f] Sin[b] Sin[e] Sinh[a] Sinh[d] +
          Sin[b] Sin[c] Sin[e] Sin[f] Sinh[a] Sinh[d]
```

Come vedete, le cose si complicano un pochetto, ma perchè raramente vediamo questo tipo di cose (almeno io le ho usate davvero poco...)

```
In[53]:= CrossProduct[{a, b, c}, {d, e, f}, Bipolar] // FullSimplify // Timing

Out[53]= {454.656 Second, {-2 Im[ArcCoth[f Sin[a]/(-Cos[a] + Cosh[b]) +
                 c Sin[d]/(Cos[d] - Cosh[e]) + i (f Sinh[b]/(Cos[a] - Cosh[b]) +
                 c Sinh[e]/(-Cos[d] + Cosh[e]))]],
           2 Re[ArcCoth[f Sin[a]/(-Cos[a] + Cosh[b]) + c Sin[d]/(Cos[d] - Cosh[e]) +
                 i (f Sinh[b]/(Cos[a] - Cosh[b]) + c Sinh[e]/(-Cos[d] + Cosh[e]))]],
           (Sin[d] Sinh[b] - Sin[a] Sinh[e])/((Cos[a] - Cosh[b]) (Cos[d] - Cosh[e]))}}
```

Come vedete, il risultato è tutt'altro che semplice, ma *Mathematica* digerisce con facilità anche queste bestialita abnormi...

Un'altro aspetto da considerare, molto importante quando si devono effettuare integrazioni e calcoli differenziali su campi tridimensionale, è il differenziale che si usa dV. In coordinate cartesiane

questo è dato da $(d\,x^2 + d\,y^2 + d\,z^2)^{1/2}$, che rappresenta la lunghezza d'arco differenziale; ma ovviamente questa formula cambia al variare del tipo di coordinate che usiamo nel nostro problema:

`ArcLengthFactor[{`f_x`,` f_y`,` f_z`}, t]`	restituisce la derivata della lunghezza d'arco lungo la curva parametrizzata da *t* nel sistema di coordinate di default.
`ArcLengthFactor[{`f_x`,` f_y`,` f_z`}, t,` *coordsys*`]`	restituisce la stessa cosa di prima, ma nelle coordinate *coordsys*

consideriamo la seguente funzione parametrica:

```
In[54]:= param = {Cos[t], Sin[t], t}
Out[54]= {Cos[t], Sin[t], t}
```

Come avrete capito, in coordinate cartesiane, rappresenta un'elica. La lunghezza d'arco rappresenta la velocità con cui il punto, al variare di t, descrive la figura. In questo caso abbiamo:

```
In[55]:= ArcLengthFactor[param, t, Cartesian]
Out[55]= Sqrt[1 + Cos[t]^2 + Sin[t]^2]
```

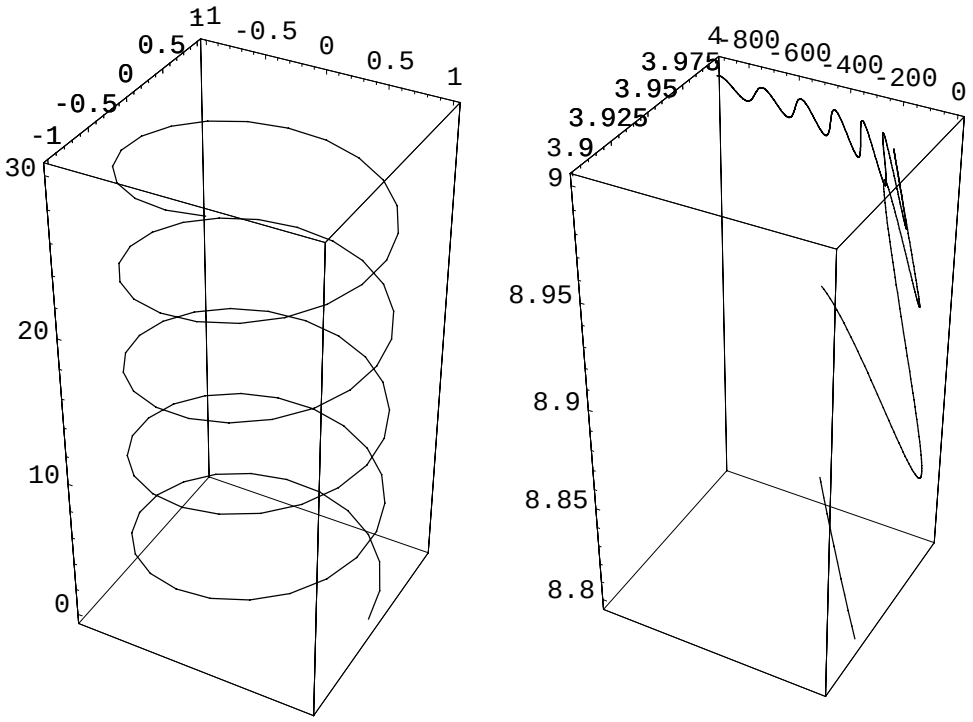
```
In[56]:= % // Simplify
Out[56]= Sqrt[2]
```

Come potete vedere, questa rappresenta la 'velocità' del punto che descrive la curva al variare di t. Tuttavia, se le stesse equazioni le utilizziamo per un altro sistema di coordinate, non rappresentano più un'elica, e di conseguenza questo valore varia:

```
In[57]:= ArcLengthFactor[param, t, Spherical] // FullSimplify
Out[57]= Sqrt[Cos[t]^4 + Sin[t]^2 + Cos[t]^2 Sin[Sin[t]]^2]
```

Come potete vedere, in questo caso la velocità varia al variare della posizione, cioè del parametro t.

In[58]:=
```
p1 = ParametricPlot3D[param, {t, 0, 30},
   BoxRatios → {1, 1, 2},
   PlotPoints → 100
  ];

SetCoordinates[ConfocalEllipsoidal];

p2 = ParametricPlot3D[
   CoordinatesFromCartesian[param], {t, 0, 30},
   BoxRatios → {1, 1, 2},
   PlotPoints → 1000
  ];

Show[GraphicsArray[{p1, p2}]]
```



Out[61]= - GraphicsArray -

Come potete vedere, la stessa rappresentazione in coordinate diverse cambia di parecchio... Di conseguenza cambiano anche i valori che abbiamo calcolato prima...

Quello che abbiamo calcolato, in pratica, ci serve per poter calcolare l'integrale di una funzione, definita nelle tre coordinate, con l'integrale che si estende lungo la curva che abbiamo definito. Supponiamo di definire la seguente funzione:

In[62]:=
```
f[{x_, y_, z_}] := 3 x^2 Sin[y] z
```

Notate come abbia scritto come argomento una lista. Questo ci serve per scrivere in modo semplificato la funzione nelle nostre coordinate: infatti scriveremo:

```
In[63]:= f[param]

Out[63]= 3 t Cos[t]^2 Sin[Sin[t]]
```

invece di

```
In[64]:= f[param[[1]], param[[2]], param[[3]]]

Out[64]= f[Cos[t], Sin[t], t]
```

Oppure un altro dei modi con Map e così via...

Per poter ottenere l'integrale lungo la curva, occorre integrare la funzione, avente come argomento le coordinate della curva parametrica lungo cui vogliamo calcolare l'integrale, per la lunghezza d'arco definita dal sistema di coordinate che abbiano scelto:

```
In[65]:= Integrate[
          f[param] ArcLengthFactor[param, t, Cartesian],
          {t, 0, 2 Pi}]

Out[65]= -3 Sqrt[2] π^2 StruveH[1, 1]
```

MMMMmmmm.... la soluzione è più complicata di quanto sembrasse a prima vista. Comunque siamo riusciti a calcolarla. Se volessimo avere un'approssimazione numerica basterebbe fare:

```
In[66]:= N[%]

Out[66]= -8.31004
```

All'opposto dell'integrale, c'è la derivata, che viene rappresentata nelle sue derivate parziali dalla matrice Jacobiana, che serve ad aggiustare il volumetto elementare nel cambio di coordinate:

`JacobianMatrix[ ]`	restitusce la matrice delle derivate della trasformazione dal sistema di coordinate di default a quello cartesiano, usando le veriabili di default
`JacobianMatrix[`*pt*`]`	matrice Jacobiana calcolata nel punto specificato, dato nelle coordinate del sistema di coordinate di
`JacobianMatrix[`*coordsys*`]`	restituisce la matrice delle derivate per passare dalle coordinate *coordsys* a quelle cartesiane
`JacobianMatrix[`*pt*`, `*coordsys*`]`	come sopra, ma nel punto specificato...
`JacobianDeterminant[ ]`, `JacobianDeterminant[`*pt*`]`, etc.	determinante della matrice Jacobiana
`ScaleFactors[ ]`, `ScaleFactors[`*pt*`]`, etc.	lista dei fattori di scala

Vediamo la matrice Jacobiana per passare dalle coordinate cartesiani a quelle cartesiani:

In[67]:=
```
JacobianMatrix[Cartesian] // MatrixForm
```

Out[67]//MatrixForm=

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

Alquanto banale.... Vediamo invece di vedere qualcosa di più serio, per esempio usando il sistema di coordinate bipolare:

In[68]:=
```
JacobianMatrix[Bipolar] // MatrixForm
```

Out[68]//MatrixForm=

$$\begin{pmatrix} -\frac{\text{Sin}[Uu]\,\text{Sinh}[Vv]}{(-\text{Cos}[Uu]+\text{Cosh}[Vv])^2} & \frac{\text{Cosh}[Vv]}{-\text{Cos}[Uu]+\text{Cosh}[Vv]} - \frac{\text{Sinh}[Vv]^2}{(-\text{Cos}[Uu]+\text{Cosh}[Vv])^2} & 0 \\ \frac{\text{Cos}[Uu]}{-\text{Cos}[Uu]+\text{Cosh}[Vv]} - \frac{\text{Sin}[Uu]^2}{(-\text{Cos}[Uu]+\text{Cosh}[Vv])^2} & -\frac{\text{Sin}[Uu]\,\text{Sinh}[Vv]}{(-\text{Cos}[Uu]+\text{Cosh}[Vv])^2} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

L'integrale su una regione tridimensionale si calcola usando lo Jacobiano. per esempio, se cerco di integrare una funzione descritta in un sistema di coordinate che non sia quello cartesiano, bisogna utilizzare la trasformazione opportuna: ovviamente, ci serve il determinante dello Jacobiano... Supponiamo di avere la seguente funzione espressa in coordinate cilindriche:

In[69]:=
```
g[r_, θ_, z_] := Sin[θ]^2 z r
```

L'integrale, calcolato nel cilindro di raggio 3 e di altezza 2 si scrive:

```
In[70]:= Integrate[
          g[r, θ, z] JacobianDeterminant[Cylindrical[r, θ, z]],
          {r, 0, 3}, {θ, 0, 2 Pi}, {z, 0, 2}
         ]
```

```
Out[70]= 18 π
```

Infine, vediamo le funzioni che voi tutti sicuramente aspettate, che sono quelle più comuni quando si usano funzioni tridimensionali:

`Div[`*f*`]`	divergenza del campo vettoriale *f* nel sistema di coordinate di default
`Curl[`*f*`]`	rotore del campo vettoriale *f* nel sistema di coordinate di default
`Grad[`*f*`]`	gradiente della funzione scalare *f* nel sistema di coordinate di default
`Laplacian[`*f*`]`	laplaciano della funzione scalare *f* nel sistema di coordinate di default
`Biharmonic[`*f*`]`	laplaciano del laplaciano della funzione scalare *f* nel sistema di coordinate di default
`Div[`*f*`,` *coordsys*`]`, `Curl[`*f*`,` *coordsys*`]`, etc.	operatori differenziali che operano nel sistema di coordinate *coordsys*

Il campo vettoriale è dato come lista delle tre funzioni che rappresentano le componenti del campo vettoriale. Ogni funzione, in genere, dipenderà da tutte e tre le coordinate. Per esempio, in forma simbolica possiamo avere:

```
In[71]:= Clear[f, g, h]
```

```
In[72]:= Div[{f[r, θ, ϕ], g[r, θ, ϕ], h[r, θ, ϕ]}, Spherical[r, θ, ϕ]]
```

```
Out[72]= 1/r^2 (Csc[θ] (r Cos[θ] g[r, θ, ϕ] + 2 r f[r, θ, ϕ] Sin[θ] + r h^(0,0,1)[r, θ, ϕ] +
           r Sin[θ] g^(0,1,0)[r, θ, ϕ] + r^2 Sin[θ] f^(1,0,0)[r, θ, ϕ]))
```

Come vedete in questo caso viene dato in forma puramente simbolica. Possiamo fare un caso più pratico, per farvi vedere meglio:

```
In[73]:= Grad[r^2 + y
             θ^3 + (r θ ϕ)^4, Spherical[r, θ, ϕ]] // Simplify
```

```
Out[73]= {2 (r + 2 r^3 θ^4 ϕ^4), θ^2 (3 y + 4 r^4 θ ϕ^4)/r, 4 r^3 θ^4 ϕ^3 Csc[θ]}
```

■ DiscreteMath`Combinatorica`

Questo è uno dei packages più corposi di *Mathematica*, con oltre 450 funzioni avanzate per permutazioni e combinazioni, sia per disegnare e calcolare con i grafi, orientati e non. Sono funzioni molto specifiche, con cui personalmente non ho mai avuto a che fare, ma che riporto perchè reputo comunque il pacchetto importante per chiunque abbia a che fare con calcolo combinatorio.

Le funzioni per le permutazioni sono:

BinarySearch	DerangementQ
Derangements	DistinctPermutations
EncroachingListSet	FromCycles
FromInversionVector	HeapSort
Heapify	HideCycles
IdentityPermutation	Index
InversePermutation	InversionPoset
Inversions	InvolutionQ
Involutions	Josephus
LexicographicPermutations	LongestIncreasingSubsequence
MinimumChangePermutations	NextPermutation
PermutationQ	PermutationType
PermutationWithCycle	Permute
RandomHeap	RandomPermutation
RankPermutation	RevealCycles
Runs	SelectionSort
SignaturePermutation	ToCycles
ToInversionVector	UnrankPermutation

I comandi per i subset sono:

BinarySubsets	DeBruijnSequence
GrayCodeKSubsets	GrayCodeSubsets
GrayGraph	KSubsets
LexicographicSubsets	NextBinarySubset
NextGrayCodeSubset	NextKSubset
NextLexicographicSubset	NextSubset
NthSubset	RandomKSubset
RandomSubset	RankBinarySubset
RankGrayCodeSubset	RankKSubset
RankSubset	Strings
Subsets	UnrankBinarySubset
UnrankGrayCodeSubset	UnrankKSubset
UnrankSubset	

E i comandi per la teoria dei gruppi sono:

AlternatingGroup	AlternatingGroupIndex
CycleIndex	CycleStructure
Cycles	Cyclic
CyclicGroup	CyclicGroupIndex
Dihedral	DihedralGroup
DihedralGroupIndex	EquivalenceClasses
KSubsetGroup	KSubsetGroupIndex
ListNecklaces	MultiplicationTable
NecklacePolynomial	OrbitInventory
OrbitRepresentatives	Orbits
Ordered	PairGroup
PairGroupIndex	PermutationGroupQ
SamenessRelation	SymmetricGroup
SymmetricGroupIndex	

Per le partizioni di interi abbiamo:

Compositions	DominatingIntegerPartitionQ
DominationLattice	DurfeeSquare
FerrersDiagram	NextComposition
NextPartition	PartitionQ
ReverseLexicographicPartitions	RandomComposition
RandomPartition	TransposePartition

Per i set di partizioni abbiamo:

CoarserSetPartitionQ	FindSet
InitializeUnionFind	KSetPartitions
PartitionLattice	RGFQ
RGFToSetPartition	RGFs
RandomKSetPartition	RandomRGF
RandomSetPartition	RankKSetPartition
RankRGF	RankSetPartition
SetPartitionListViaRGF	SetPartitionQ
SetPartitionToRGF	SetPartitions
ToCanonicalSetPartition	UnionSet
UnrankKSetPartition	UnrankRGF
UnrankSetPartition	

Per le tavole di Young abbiamo le seguenti funzioni:

ConstructTableau	DeleteFromTableau
FirstLexicographicTableau	InsertIntoTableau
LastLexicographicTableau	NextTableau
PermutationToTableaux	RandomTableau
TableauClasses	TableauQ
Tableaux	TableauxToPermutation
TransposeTableau	

Per conteggi vari abbiamo:

Backtrack	BellB
Cofactor	Distribution
Element	Eulerian
NumberOf2Paths	NumberOfCompositions
NumberOfDerangements	NumberOfDirectedGraphs
NumberOfGraphs	NumberOfInvolutions
NumberOfKPaths	NumberOfNecklaces
NumberOfPartitions	NumberOfPermutationsByCycles
NumberOfPermutationsByInversions	NumberOfPermutationsByType
NumberOfSpanningTrees	NumberOfTableaux
StirlingFirst	StirlingSecond

Per esempio, se vogliamo vedere tutti i possibili modi per scrivere un numero come somma di un determinato numero di addendi, possiamo utilizzare il seguente comando:

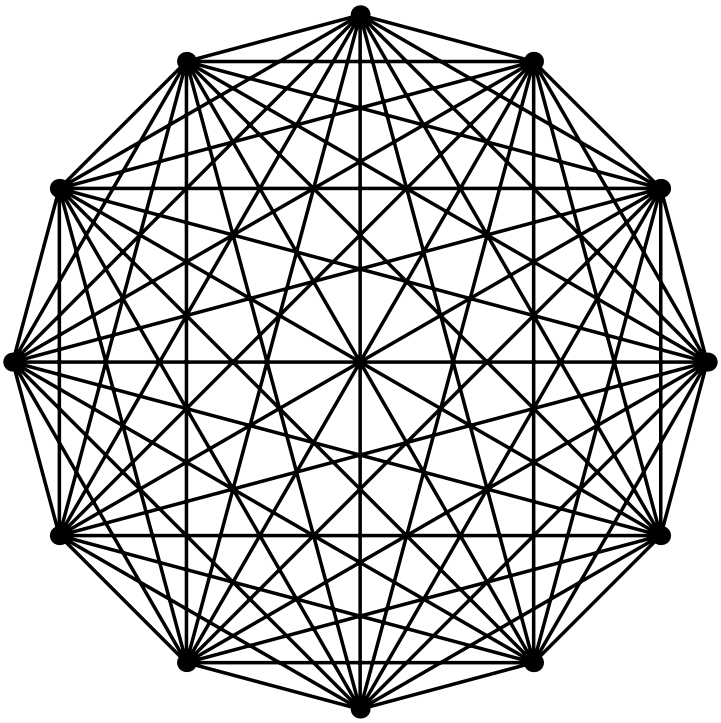
```
In[74]:= << DiscreteMath`Combinatorica`

In[75]:= Compositions[9, 3]

Out[75]= {{0, 0, 9}, {0, 1, 8}, {0, 2, 7}, {0, 3, 6}, {0, 4, 5},
          {0, 5, 4}, {0, 6, 3}, {0, 7, 2}, {0, 8, 1}, {0, 9, 0},
          {1, 0, 8}, {1, 1, 7}, {1, 2, 6}, {1, 3, 5}, {1, 4, 4},
          {1, 5, 3}, {1, 6, 2}, {1, 7, 1}, {1, 8, 0}, {2, 0, 7},
          {2, 1, 6}, {2, 2, 5}, {2, 3, 4}, {2, 4, 3}, {2, 5, 2},
          {2, 6, 1}, {2, 7, 0}, {3, 0, 6}, {3, 1, 5}, {3, 2, 4}, {3, 3, 3},
          {3, 4, 2}, {3, 5, 1}, {3, 6, 0}, {4, 0, 5}, {4, 1, 4}, {4, 2, 3},
          {4, 3, 2}, {4, 4, 1}, {4, 5, 0}, {5, 0, 4}, {5, 1, 3}, {5, 2, 2},
          {5, 3, 1}, {5, 4, 0}, {6, 0, 3}, {6, 1, 2}, {6, 2, 1}, {6, 3, 0},
          {7, 0, 2}, {7, 1, 1}, {7, 2, 0}, {8, 0, 1}, {8, 1, 0}, {9, 0, 0}}
```

Notate come siano dati le parti ordinate, per cui {0, 0, 9} ≠ {0, 9, 0}, per esempio. Questo perchè la funzione dice come dividere in un numero determinato di gruppi, un determinato numero di elementi, per cui i gruppi sono comunque diversi.

Comunque, sono funzioni avanzate, che conosceranno chi deve utilizzarle. Se le provate a caso (come me) vi perderete subito. Se cercate qualcosa di particolare perchè sapete che vi serve, qua siete nel posto giusto.

Invece, trovo più interessante la parte di questo package riguardante la gestione dei grafi.

In[76]:=
```
ShowGraph[CompleteGraph[12]]
```



Out[76]=
```
- Graphics -
```

Come potete vedere, abbiamo realizzato un grafo completo di 12 elementi in men che non si dica...

Se vogliamo mostrare la matrice di adiacenza di un grafo, basta semplicemente utilizzare il comando designato:

In[77]:=
```
TableForm[ToAdjacencyMatrix[CompleteGraph[12]]]
```

Out[77]//TableForm=

0	1	1	1	1	1	1	1	1	1	1	1
1	0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	1	1	1	1	1	1
1	1	1	0	1	1	1	1	1	1	1	1
1	1	1	1	0	1	1	1	1	1	1	1
1	1	1	1	1	0	1	1	1	1	1	1
1	1	1	1	1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1	0	1	1	1
1	1	1	1	1	1	1	1	1	0	1	1
1	1	1	1	1	1	1	1	1	1	0	1
1	1	1	1	1	1	1	1	1	1	1	0

Possiamo anche vedere il numero di vertici di un grafo:

In[78]:=
```
V[CompleteGraph[12]]
```

Out[78]=
```
12
```

Ed il numero di lati:

In[79]:=
```
M[CompleteGraph[12]]
```

Out[79]=
```
66
```

Per chi ancora non l'avesse capito, CompleteGraph crea un grafo in cui ogni elemento è legato all'altro... Qua sotto sono rappresentate le funzioni usate per creare i particolari tipi di grafi:

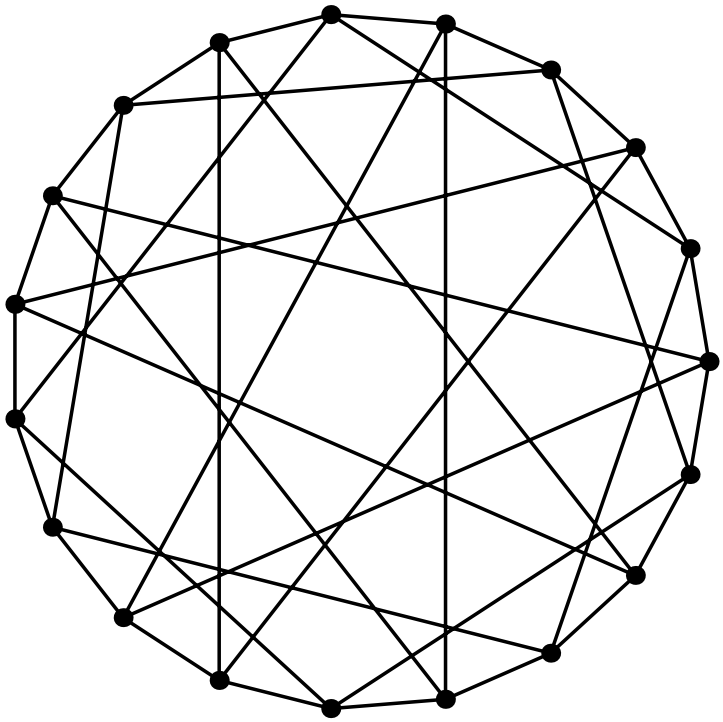
BooleanAlgebra	ButterflyGraph
CageGraph	CartesianProduct
ChvatalGraph	CirculantGraph
CodeToLabeledTree	CompleteBinaryTree
CompleteGraph	CompleteKPartiteGraph
CompleteKaryTree	CoxeterGraph
CubeConnectedCycle	CubicalGraph
Cycle	DeBruijnGraph
DodecahedralGraph	EmptyGraph
ExactRandomGraph	ExpandGraph
FiniteGraphs	FolkmanGraph
FranklinGraph	FruchtGraph
FunctionalGraph	GeneralizedPetersenGraph
GraphComplement	GraphDifference
GraphIntersection	GraphJoin
GraphPower	GraphProduct
GraphSum	GraphUnion
GridGraph	GrotztschGraph
Harary	HasseDiagram
HeawoodGraph	HerschelGraph
Hypercube	IcosahedralGraph
IntervalGraph	KnightsTourGraph
LabeledTreeToCode	LeviGraph
LineGraph	ListGraphs
MakeGraph	McGeeGraph
MeredithGraph	MycielskiGraph
NoPerfectMatchingGraph	NonLineGraphs
OctahedralGraph	OddGraph
OrientGraph	Path
PermutationGraph	PetersenGraph
RandomGraph	RandomTree
RealizeDegreeSequence	RegularGraph
RobertsonGraph	ShuffleExchangeGraph
SmallestCyclicGroupGraph	Star
TetrahedralGraph	ThomassenGraph
TransitiveClosure	TransitiveReduction
Turan	TutteGraph
Uniquely3ColorableGraph	UnitransitiveGraph
VertexConnectivityGraph	WaltherGraph
Wheel	

Inoltre, possiamo anche creare dei grafi a partire da altre rappresentazioni: per esempio possiamo creare un grafo se siamo in possesso della matrice di adiacenza, oppure della lista di coppie ordinate delle connessioni, oppure di ottenere una determinata rappresentazione di un grafo:

Edges	FromAdjacencyLists
FromAdjacencyMatrix	FromOrderedPairs
FromUnorderedPairs	IncidenceMatrix
ToAdjacencyLists	ToAdjacencyMatrix
ToOrderedPairs	ToUnorderedPairs

Per esempio:

In[80]:=
```
ShowGraph[RobertsonGraph]
```



Out[80]= `- Graphics -`

Se cercate un tipo di grafo in particolare, di certo lo troverete nell'elenco di sopra...

Possiamo anche effettuare delle operazioni sui grafi per modificarli:

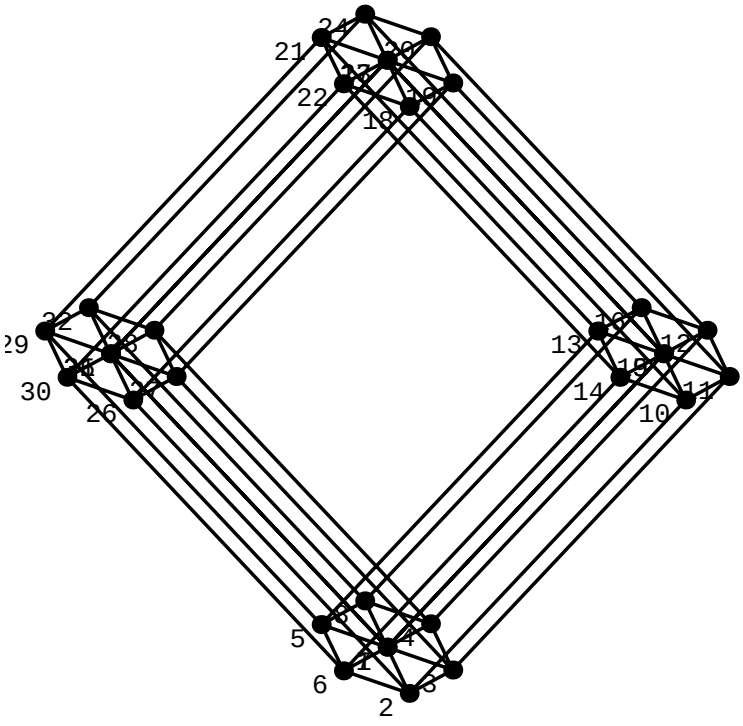
AddEdge	AddEdges
AddVertex	AddVertices
ChangeEdges	ChangeVertices
Contract	DeleteCycle
DeleteEdge	DeleteEdges
DeleteVertex	DeleteVertices
InduceSubgraph	MakeDirected
MakeSimple	MakeUndirected
PermuteSubgraph	RemoveMultipleEdges
RemoveSelfLoops	ReverseEdges

Per esempio, definiamo un grafo:

In[81]:=
```
grafo = Hypercube[5]
```

Out[81]=
```
-Graph:<80, 32, Undirected>-
```

In[82]:=
```
ShowGraph[grafo, VertexNumber → True]
```



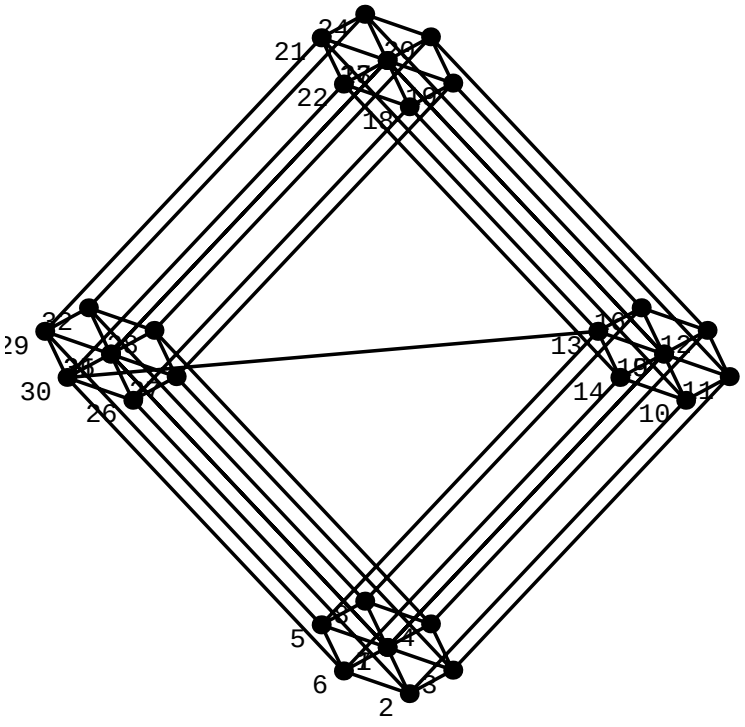
Out[82]=
```
- Graphics -
```

Supponiamo di dover aggiungere una connessione fra il vertice 13 ed il 30:

In[83]:= **grafo = AddEdge[grafo, {13, 30}]**

Out[83]= **-Graph:<81, 32, Undirected>-**

In[84]:= **ShowGraph[grafo, VertexNumber → True]**



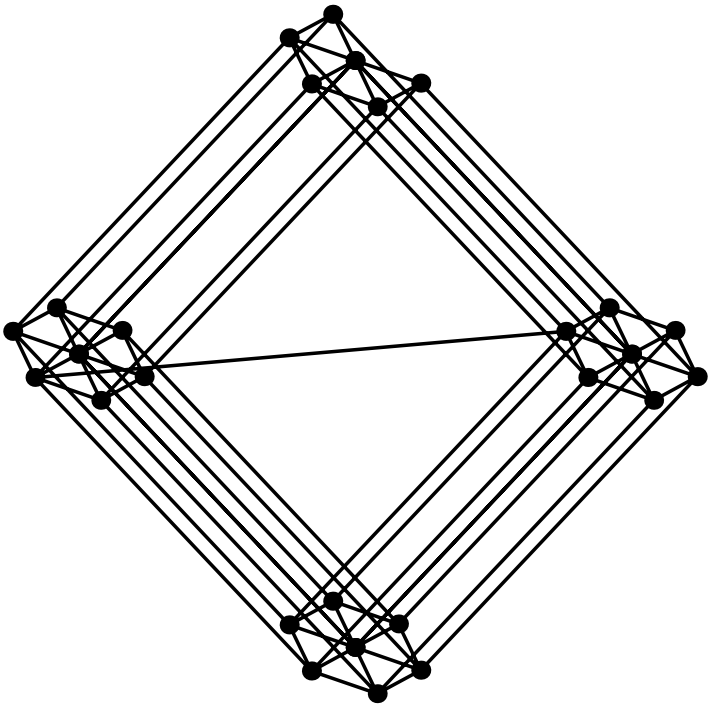
Out[84]= **- Graphics -**

Se adesso voglio cancellare il vertice numero 20 basta fare:

In[85]:= **grafo = DeleteVertex[grafo, 20]**

Out[85]= **-Graph:<76, 31, Undirected>-**

In[86]:=
```
ShowGraph[grafo]
```



Out[86]=
```
- Graphics -
```

Come potete vedere, sono scomparse anche le connessioni di quel vertice particolare. Se adesso mi serve la lista di coppie non ordinate basta fare:

In[87]:=
```
ToUnorderedPairs[grafo]
```

Out[87]=
```
{{1, 2}, {2, 3}, {3, 4}, {1, 4}, {5, 6}, {6, 7}, {7, 8}, {5, 8},
 {9, 10}, {10, 11}, {11, 12}, {9, 12}, {13, 14}, {14, 15}, {15, 16},
 {13, 16}, {17, 18}, {18, 19}, {20, 21}, {21, 22}, {22, 23},
 {20, 23}, {24, 25}, {25, 26}, {26, 27}, {24, 27}, {28, 29},
 {29, 30}, {30, 31}, {28, 31}, {1, 5}, {2, 6}, {3, 7}, {4, 8},
 {9, 13}, {10, 14}, {11, 15}, {12, 16}, {17, 20}, {18, 21}, {19, 22},
 {24, 28}, {25, 29}, {26, 30}, {27, 31}, {1, 9}, {2, 10}, {3, 11},
 {4, 12}, {5, 13}, {6, 14}, {7, 15}, {8, 16}, {9, 17}, {10, 18},
 {11, 19}, {13, 20}, {14, 21}, {15, 22}, {16, 23}, {17, 24}, {18, 25},
 {19, 26}, {20, 28}, {21, 29}, {22, 30}, {23, 31}, {1, 24}, {2, 25},
 {3, 26}, {4, 27}, {5, 28}, {6, 29}, {7, 30}, {8, 31}, {13, 29}}
```

Le coppie le ho definite non ordinate perchè non lo era il grafo.

Inoltre, possiamo anche avere delle opzioni che modificano l'aspetto del grafo:

Algorithm	Box
Brelaz	Center
Circle	Directed
Disk	EdgeColor
EdgeDirection	EdgeLabel
EdgeLabelColor	EdgeLabelPosition
EdgeStyle	EdgeWeight
Euclidean	HighlightedEdgeColors
HighlightedEdgeStyle	HighlightedVertexColors
HighlightedVertexStyle	Invariants
LNorm	Large
LoopPosition	LowerLeft
LowerRight	NoMultipleEdges
NoSelfLoops	Normal
NormalDashed	NormalizeVertices
One	Optimum
Parent	PlotRange
RandomInteger	Simple
Small	Strong
Thick	ThickDashed
Thin	ThinDashed
Type	Undirected
UpperLeft	UpperRight
VertexColor	VertexLabel
VertexLabelColor	VertexLabelPosition
VertexNumber	VertexNumberColor
VertexNumberPosition	VertexStyle
VertexWeight	Weak
WeightRange	WeightingFunction
Zoom	

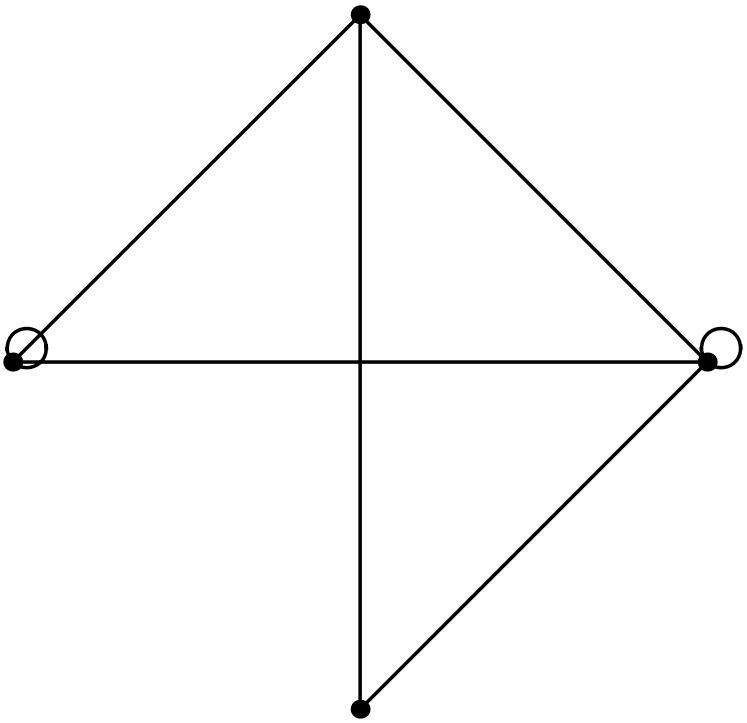
Per esempio, supponiamo di avere questo grafo, che parte dalla seguente matrice di adiacenza:

In[88]:=
```
adiacenza = {
    {0, 1, 1, 1},
    {0, 1, 0, 1},
    {1, 0, 0, 1},
    {0, 1, 1, 1}
   };
```

In[89]:=
```
grafo = FromAdjacencyMatrix[adiacenza]
```

Out[89]=
```
-Graph:<7, 4, Undirected>-
```

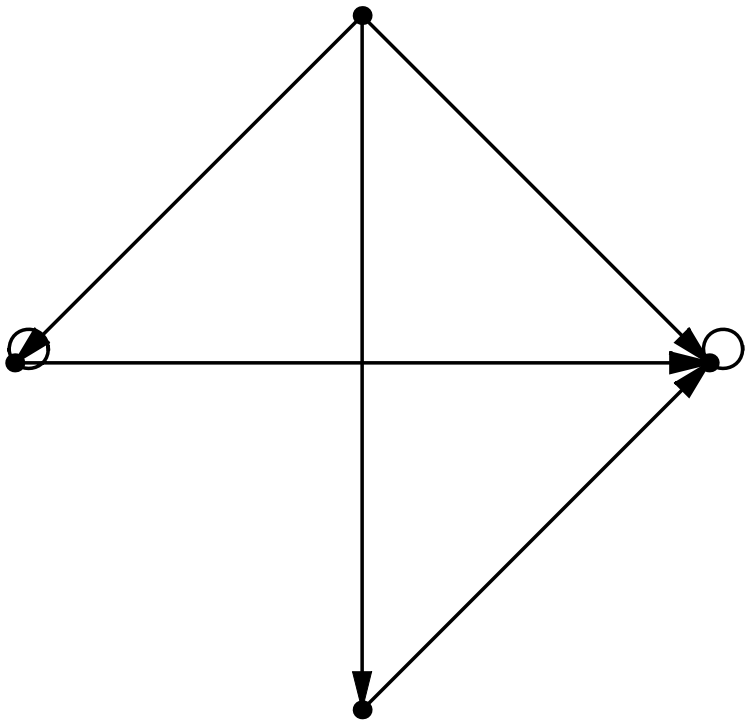
In[90]:=
```
ShowGraph[grafo]
```



Out[90]=
```
- Graphics -
```

Se vogliamo mostrare le direzioni delle connessioni:

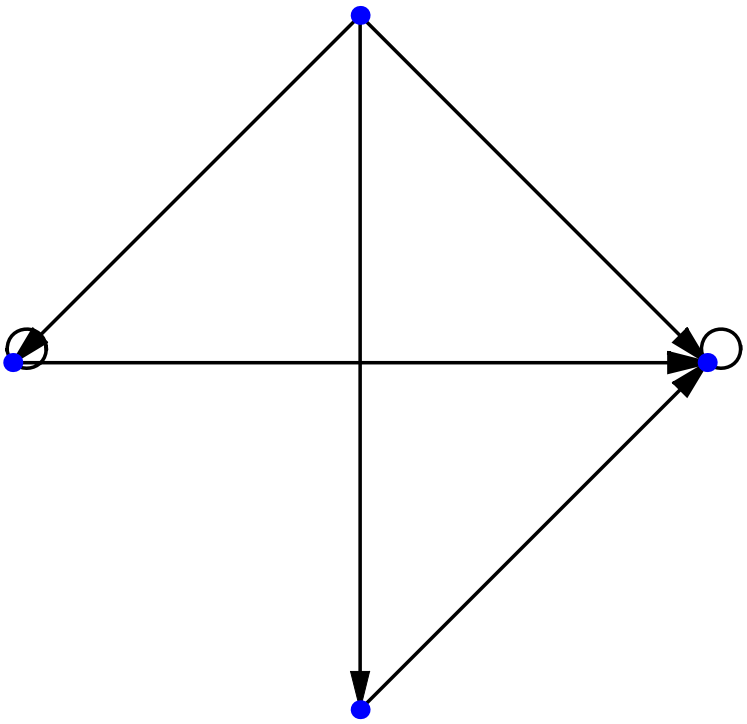
In[91]:=
```
ShowGraph[
 SetGraphOptions[
  grafo,
  EdgeDirection → True
 ]
]
```



Out[91]= - Graphics -

Se vogliamo anche modificare il punto che rappresenta un vertice del grafo:

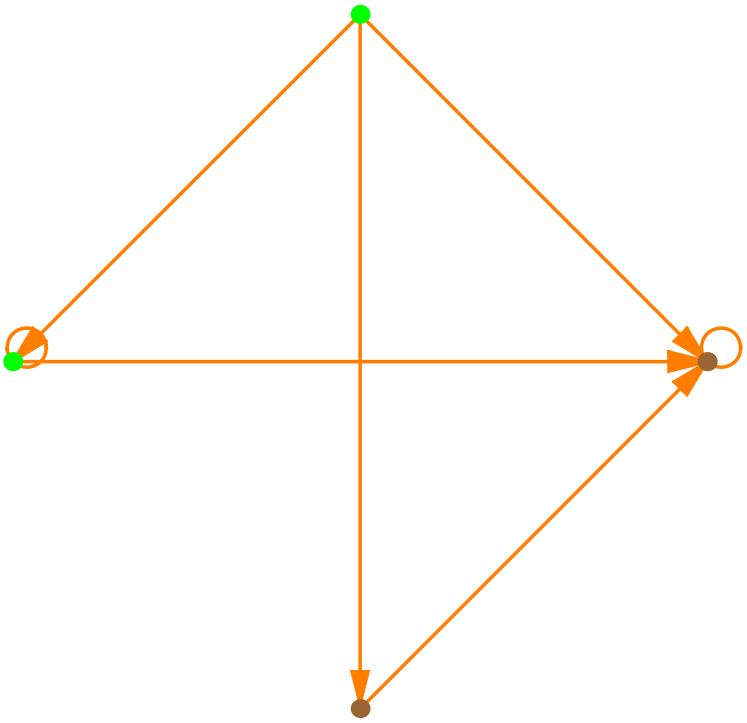
In[92]:=
```
ShowGraph[
 SetGraphOptions[
  grafo,
  EdgeDirection → True,
  VertexColor → Blue
 ]
]
```



Out[92]= `- Graphics -`

Posso anche definire lo stile per particolari parti del grafo, usando le liste:

In[93]:=
```
ShowGraph[
 SetGraphOptions[
  grafo,
  {
   {1, 2, VertexColor → Green},
   {3, 4, VertexColor → Brown}
  },
  EdgeColor → Orange,
  EdgeDirection → True
 ]
]
```



Out[93]= `- Graphics -`

Come potete vedere, potete creare grafi dall'aspetto che volete, alla faccia, se mi permettete, di Metapost e PSTricks, che mi hanno fatto impazzire per disegnare solamente una freccia...

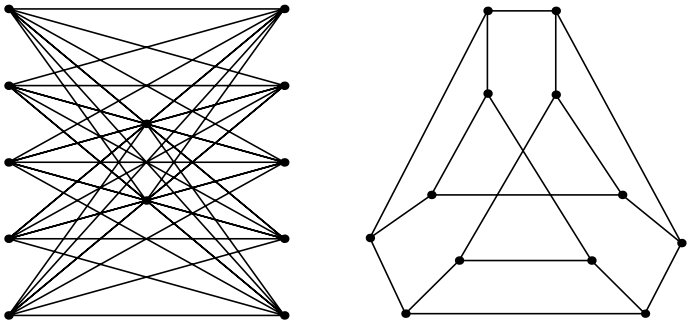
Possiamo anche definire pesi ed etichette per i grafi:

GetEdgeLabels	GetEdgeWeights
GetVertexLabels	GetVertexWeights
SetEdgeLabels	SetEdgeWeights
SetGraphOptions	SetVertexLabels
SetVertexWeights	

Ed altre funzioni per generare i disegni dei grafi:

AnimateGraph	CircularEmbedding
DilateVertices	GraphOptions
Highlight	RadialEmbedding
RandomVertices	RankGraph
RankedEmbedding	RootedEmbedding
RotateVertices	ShakeGraph
ShowGraph	ShowGraphArray
ShowLabeledGraph	SpringEmbedding
TranslateVertices	Vertices

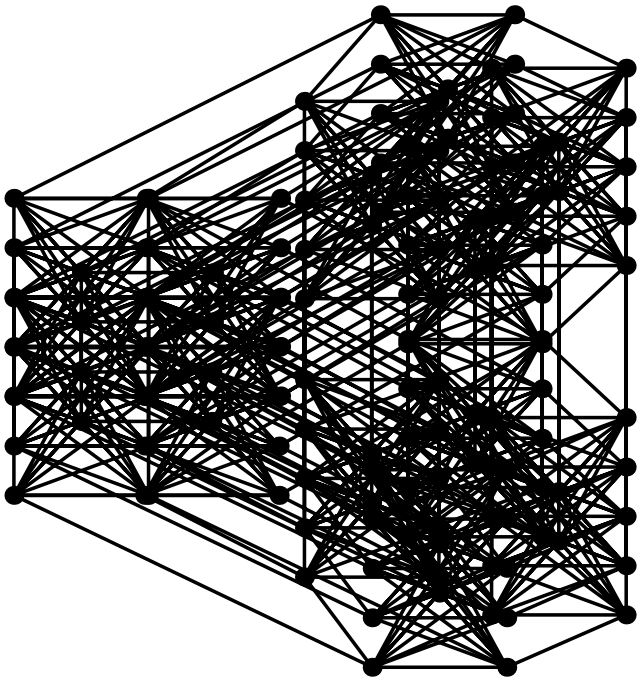
In[94]:=
```
ShowGraphArray[{CompleteGraph[5, 2, 5], FranklinGraph}]
```



Out[94]=
```
- GraphicsArray -
```

Possiamo fare il prodotto nel senso dei grafi:

In[95]:=
```
ShowGraph[GraphProduct[CompleteGraph[5, 2, 5], FranklinGraph]]
```



Out[95]=
```
- Graphics -
```

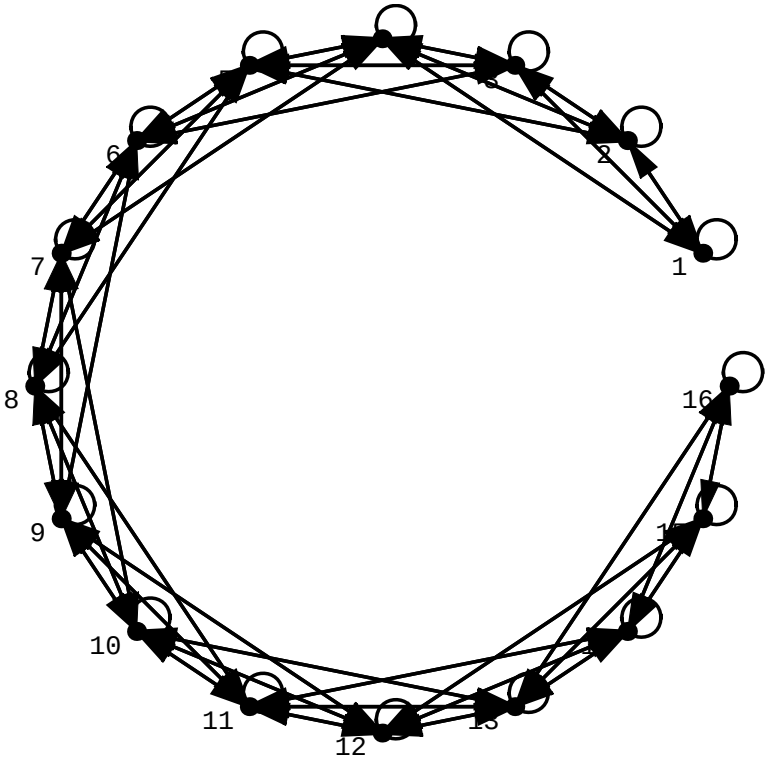
Non è che sia effettivamente molto chiara... Abbiamo preso il primo grafo, e l'abbiamo sostituito ad ogni vertice del secondo grafo, connettendo inoltre i corrispondenti punti del vari sotto-grafi.

Possiamo anche creare grafi personalizzati, dove i vertici sono connessi se soddisfano espressioni specificate da noi: bisogna specificare il numero di vertici del grafo, e la relazione booleana che dice se due vertici sono connessi. Per esempio, questa rappresentazione connette due vertici se la loro somma è minore del doppio del più piccolo dei due sommato a 4:

In[96]:=
```
grafo = MakeGraph[
  Range[16],
  (#1 + #2 < 2 Min[{#1, #2}] + 4) &
 ]
```

Out[96]=
```
-Graph:<100, 16, Directed>-
```

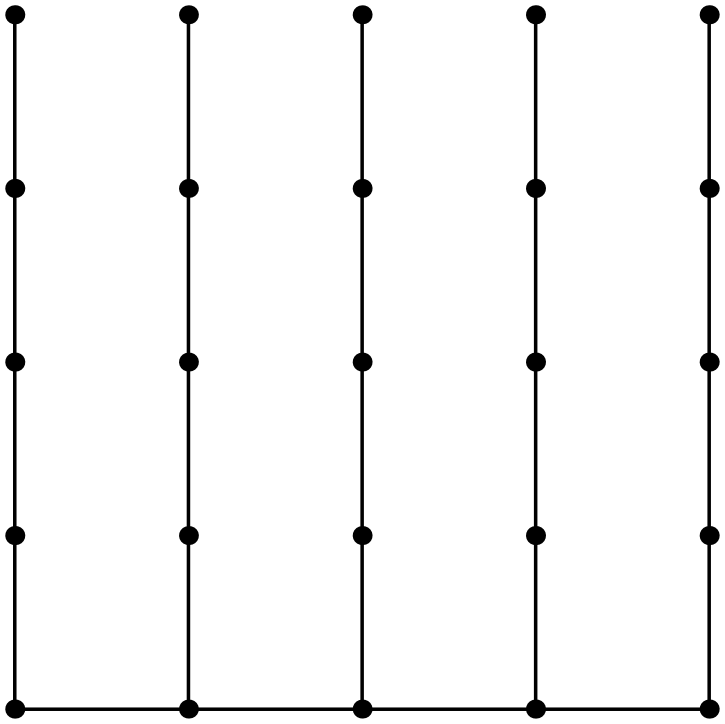
In[97]:= **ShowGraph[grafo, VertexNumber → True]**



Out[97]= **-** Graphics **-**

Come potete vedere, quello che possiamo con i grafi è veramente un casino. Ci sono libri interi che sono dedicati soltanto a questo package!!!!! Quindi come posso andare oltre la punta della punta della punta dell'iceberg??? Sperimentate e scoprirete sempre cose nuove. Vi posso assicurare che imparare questo package è come imparare un programma a parte!!!

In[98]:=
```
ShowGraph[ShortestPathSpanningTree[GridGraph[5, 5], 1]]
```



Out[98]= `- Graphics -`

■ DiscreteMath`GraphPlot

Questo package contiene altre funzioni specifiche per il plottaggio di grafi. Supporta i grafi di *Combinatoria* e le sue funzioni sono ottimizzate per funzionare con grandi grafi. Il comando principale è quello che permette di disegnare i grafici sullo schermo:

`GraphPlot[`*g*, *options*`]`	calcola un layout bidimensionale del grafo usando delle regole predefinite
`GraphPlot3D[`*g*, *options*`]`	calcola il layout di un grafo tridimensionale

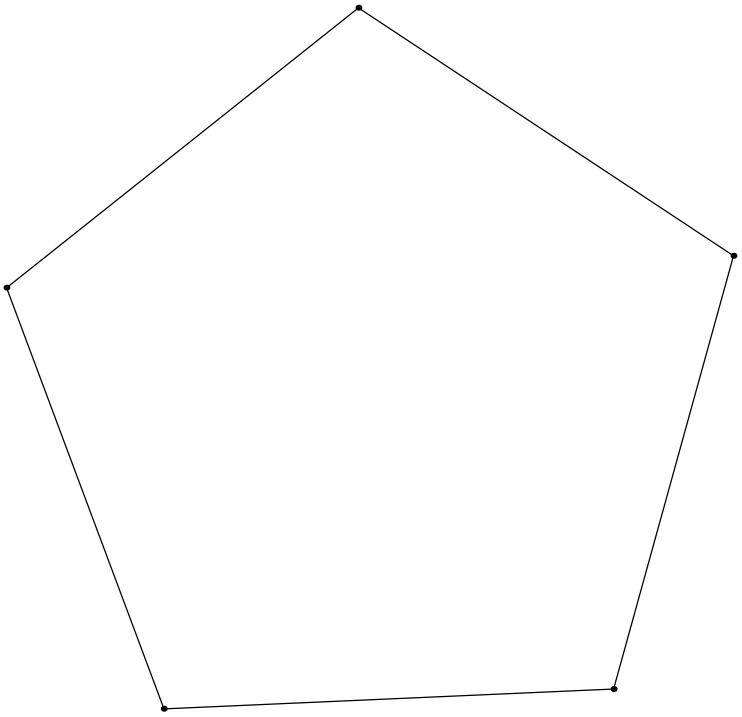
A differenza del packege di prima, in questo il disegno viene automaticamente riarrangiato: mentre prima i vertici avevano posizioni definite, qua vengono automaticamente spostati in modo da dare il layout migliore predefinito.

Vediamo un esempio, disegnando un grafo definito da una lista di regole:

In[99]:=
```
<< DiscreteMath`GraphPlot`;
```

In[100]:=
```
lista = {1 → 2, 2 → 3, 3 → 4, 4 → 5, 5 → 1};
```

In[101]:=
```
GraphPlot[lista]
```

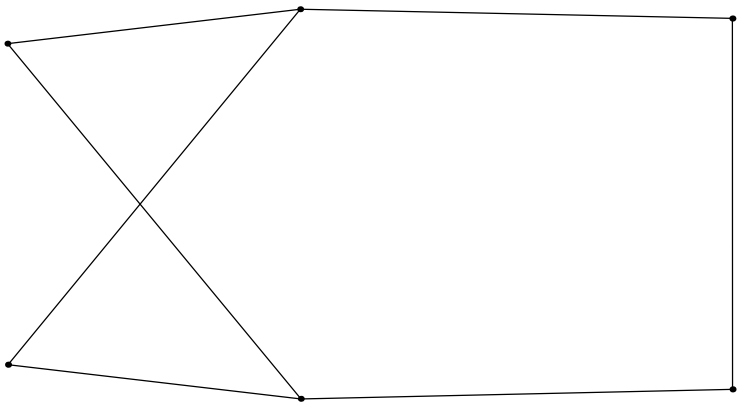


Out[101]=
```
- Graphics -
```

Aggiungiamo un altro vertice:

In[102]:=
```
lista = Flatten[Append[lista, {6 → 1, 6 → 3}]];
```

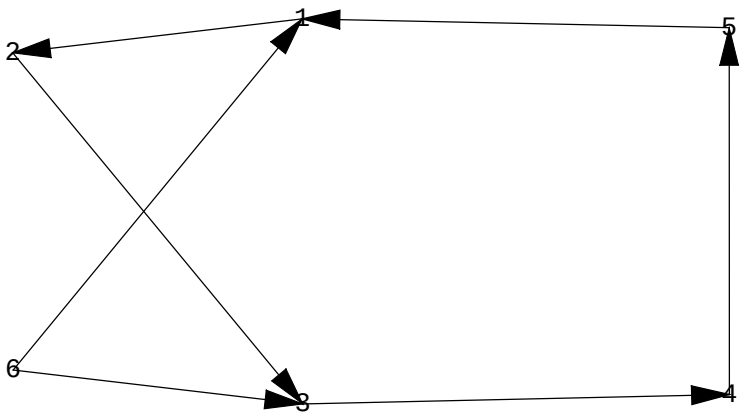
In[103]:=
```
GraphPlot[lista]
```



Out[103]=
```
- Graphics -
```

Come potete vedere, il layout dello stesos grafo è cambiato aggiungendo un vertice:

In[104]:=
```
GraphPlot[lista, EdgeStyleFunction → (Arrow[{#1, #2}] &),
 VertexStyleFunction → Automatic]
```



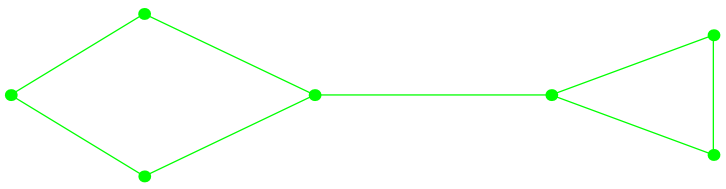
Out[104]= `- Graphics -`

option name	*default value*	
`Method`	`Automatic`	metodo usato per disegnare il grafo
`"VertexStyleFunction"`	`None`	definisce come sono disegnati i vertici
`"EdgeStyleFunction"`	`None`	descrive come disegnare i rami
`PlotStyle`	`Automatic`	definisce lo stile di disegno del grafo
`RandomSeed`	`Automatic`	definisce il posizionamento iniziale dei vertici per la funzione di disegno del grafo
`"VertexCoordinates"`	`None`	coordinate di vertici definite

Definiamo quest'altro grafo:

In[105]:=
```
grafo2 = {1 → 2, 2 → 3, 3 → 4, 4 → 5, 5 → 6, 6 → 7, 7 → 5, 1 → 4};
```

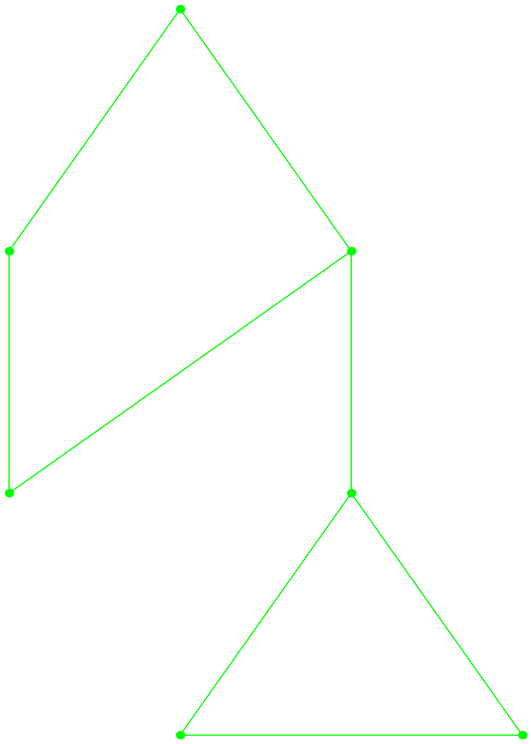
In[106]:=
```
GraphPlot[grafo2,
 PlotStyle → {Green, PointSize[0.016]}
]
```



Out[106]= `- Graphics -`

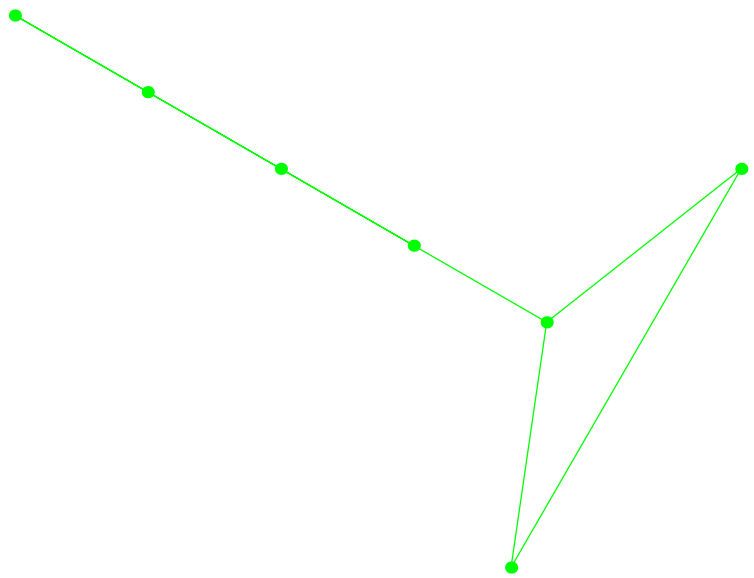
Come vedete, ho disegnato il grafo. Possiamo selezionare il metodo di disegno:

In[107]:=
```
GraphPlot[grafo2,
 PlotStyle → {Green, PointSize[0.016]},
 Method -> "LayeredDrawing"
]
```



Out[107]= - Graphics -

In[108]:=
```
GraphPlot[grafo2,
 PlotStyle → {Green, PointSize[0.016]},
 Method -> "RadialDrawing"
]
```



Out[108]= `- Graphics -`

E così via...

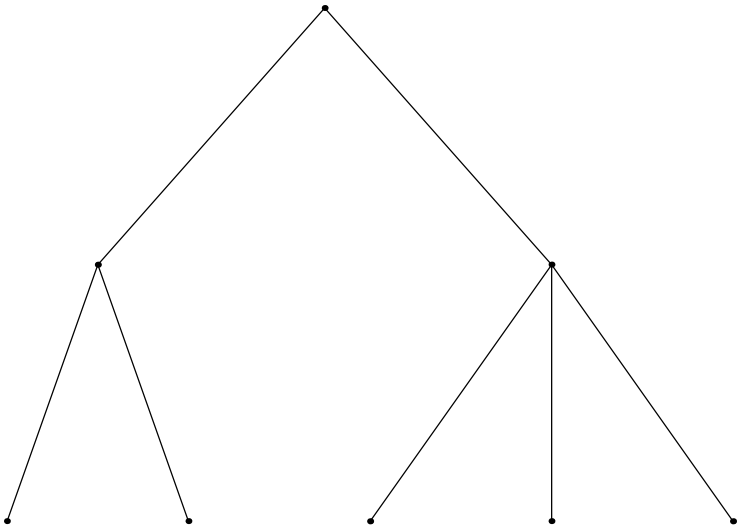
Possiamo anche disegnare facilemte degli alberi:

`TreePlot[`*g*, *options*`]`	genera un grafo ad albero
`TreePlot[`*g*, *r*, *options*`]`	genera un grafo ad albero, con radice *r*

Vediamo:

In[109]:=
```
grafo3 = {1 → 2, 2 → 3, 2 → 8, 1 → 4, 4 → 5, 4 → 6, 4 → 7};
```

In[110]:= **TreePlot[grafo3]**

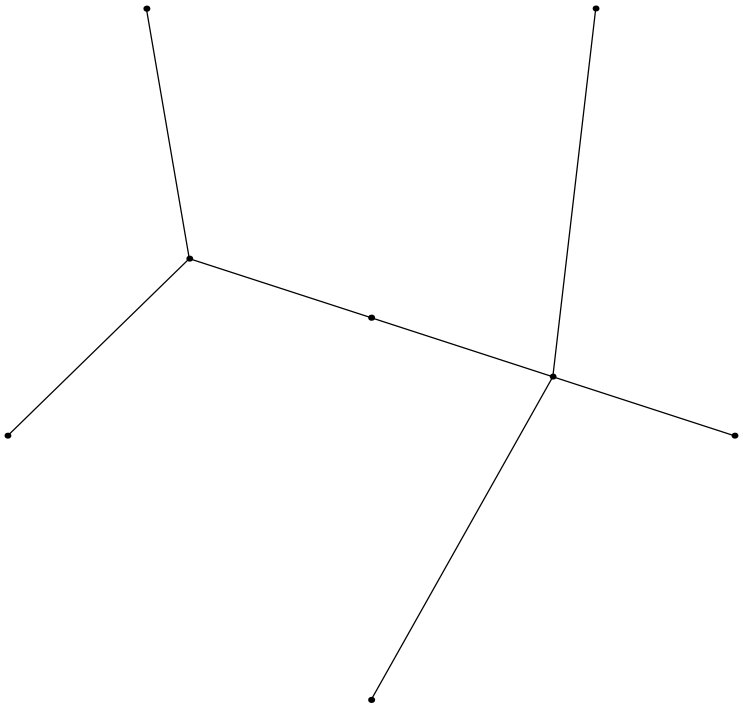


Out[110]= **-** Graphics **-**

option name	*default value*	
AspectRatio	Automatic	rapporto larghezza–altezza
"RootPosition"	Top	posizione della radice
"TreeSizeFunction"	(1&)	altezza di ogni livello dell' albero

Possiamo mettere, per esempio, la radice al centro:

In[111]:=
```
TreePlot[grafo3, "RootPosition" → Center]
```



Out[111]=
```
- Graphics -
```

Questo package aggiunge anche due utili funzioni:

`GraphDistance[g, i, j]`	calcola la distanza del grafo fra i due vertici *i* e *j*
`PseudoDiameter[g]`	restituisce lo pseudodiametro del grafo non direzionale

Considerando il penultimo grafo, possiamo vedere la distanza fra due suoi punti:

In[112]:=
```
GraphDistance[grafo2, 1, 5]
```

Out[112]=
```
2
```

Questo package contiene altri comandi minori, come il conteggio dei numeri di vertici oppure la loro lista, quindi andate a vedervi i comandi che vi servono. In fondo, questa è solo un appendice!!!

■ Graphics`Animation`

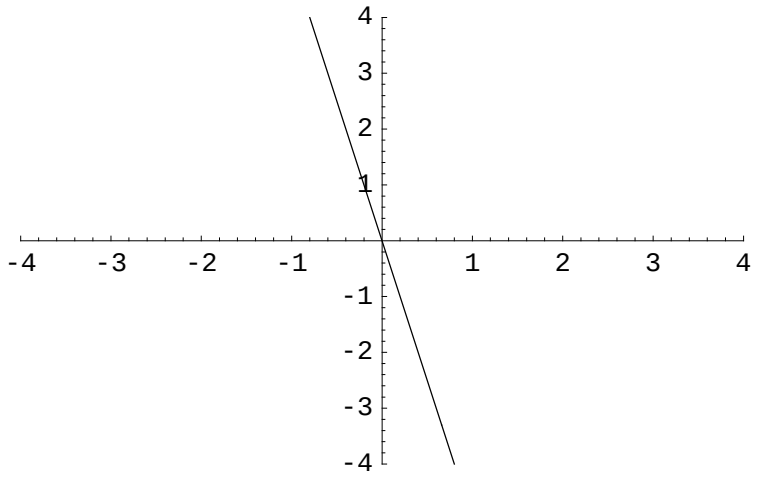
Questo package definisce comandi specifici per la creazione delle animazioni in Mathematica: definiscono in pratica un insieme di immagini che poi vengono usati come frame per l'animazione, che poi può essere esportata con il comando Export, come ben sappiamo:

`Animate[` *grcom*`,` `{`*t*`,` *tmin*`,` *tmax*`,` *dt*`}]`	esegue il comandio grafico *grcom* per il range specificato di *t*
`ShowAnimation[` `{`p_1`,` p_2`,` p_3`, … }]`	anima la sequenza di immagini "\!\(*Cell[BoxData[FormBox[SubscriptBox[StyleBox[\"\\<\\\"p\\\"\\>\", \"TI\"], StyleBox[\"\\<\\\"i\\\"\\>\", \"TI\"]], TraditionalForm]], \"InlineFormula\"]\)

Questi due comandi sono abbastanza semplici ed esplicativi. Vediamo un esempio del primo comando:

In[113]:=
```
<< Graphics`Animation`
```

In[114]:=
```
Animate[
 Plot[n x, {x, -4, 4}, PlotRange → {{-4, 4}, {-4, 4}}], {n, -5, 5}]
```

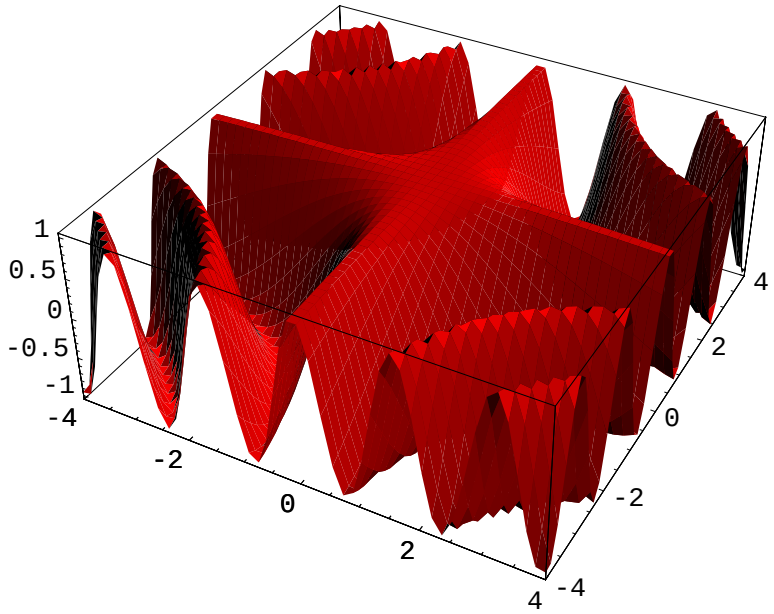


Qua vi mostro solamente i due frame per risparmiare spazio. Se adesso volete creare l'animazione, basta fare doppio clic su di un frame, e partirà l'animazione ciclica; cliccando di nuovo l'animazione si fermerà. Notate anche come in basso a sinistra nel notebook compaiano pulsanti per il controllo dell'animazione.

A proposito, dovete farlo voi, naturalmente: cliccare sul PDF non funziona...

La potenza di questo comando, però, consiste nel fatto che la parametrizzazione non è limitata alla funzione, ma anche ai parametri della funzione Plot (o di qualsiasi altra funzione che restituisca un grafico utilizzata nell'argomento del comando): possiamo per esempio creare un animazione dove cambiano i colori:

In[115]:=
```
Animate[
 Plot3D[
  Cos[x y], {x, -4, 4}, {y, -4, 4},
  LightSources → {{{1, 1, 1}, Hue[t]}}, PlotPoints → 50, Mesh → False
 ],
 {t, 0, 1, .1}
]
```



Notate come questa volta siamo stati in grado di lavorare sui parametri, oltre al fatto che abbiamo utilizzato un altra funzione grafica diversa da Plot.

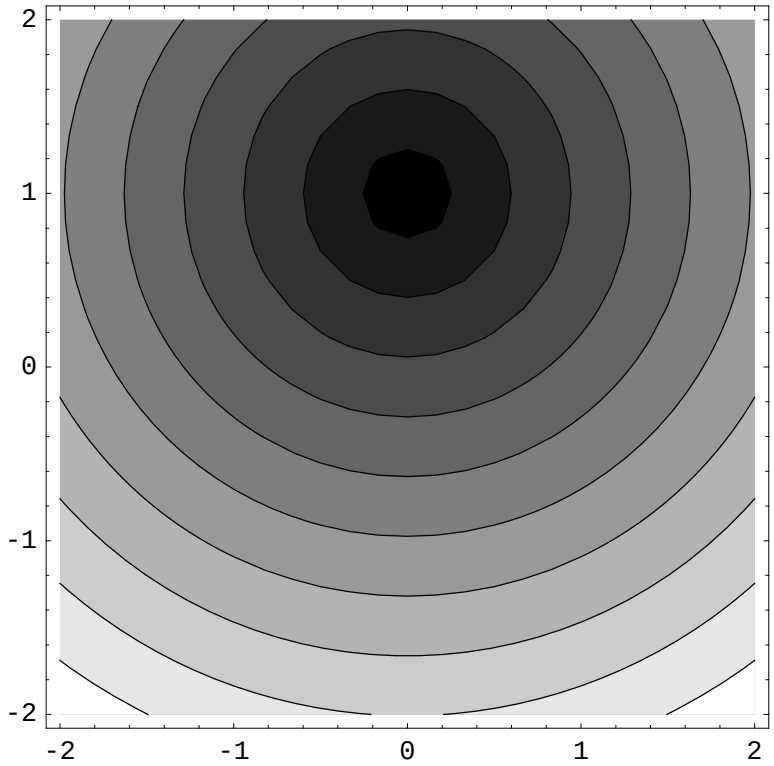
Il secondo comando, ShowAnimation, è simile al primo, con la differenza che questa volta, invece di creare i frame con un parametro, richiediamo esplicitamente la lista dei frame creati precedentemente. Quindi possiamo per esempio creare varie animazioni e fonderle in una soltanto, eseguire stacchi etc, con un minimo di pre-processing. Tenete sempre conto, però, che le animazioni (a meno che non sia un requisito del vostro studio) devono restare semplici, per far focalizzare l'attenzione sulla funzione e sul concetto, invece che sulla magnifica combinazione di colori che siete riusciti a fare.

Inoltre, ci sono comandi specifici per i vari tipi di grafico presenti in Mathematica:

`MoviePlot[ f [ x, t ], {x, xmin, xmax}, {t, tmin, tmax} ]`	anima `Plot[ f [ x, t ], {x, xmin, xmax} ]` per il range definito del parametro *t*
`MoviePlot3D[ f [ x, y, t ], {x, xmin, xmax}, {y, ymin, ymax}, {t, tmin, tmax} ]`	anima grafici tridimensionali
`MovieDensityPlot[ f [ x, y, t ], {x, xmin, xmax}, {y, ymin, ymax}, {t, tmin, tmax} ]`	anima grafici di densità
`MovieContourPlot[ f [ x, y, t ], {x, xmin, xmax}, {y, ymin, ymax}, {t, tmin, tmax} ]`	anima grafici di curve di livello
`MovieParametricPlot[ { f [ s, t ], g [ s, t ] }, {s, smin, smax}, {t, tmin, tmax} ]`	anima grafici parametrici
`SpinShow[graphics]`	ruota oggetti tridimensionali

Il funzionamento è analogo per tutte le funzioni specificate qua sopra (a parte l'ultima, che vedremo subito dopo).

In[116]:=
```
MovieContourPlot[Sqrt[(x - Sin[t])^2 + (y - Cos[3 t])^2],
 {x, -2, 2}, {y, -2, 2}, {t, 0, 2 Pi - Pi / 7, Pi / 7}]
```



Come vedete, è abbastanza semplice creare animazioni in questo modo. Inoltre, queste funzioni accettano anche gli argomenti corrispondenti alle funzioni che animano.

SpinShow, invece, crea una rotazione dell'oggetto tridimensionale; in realtà fa ruotare il punto di

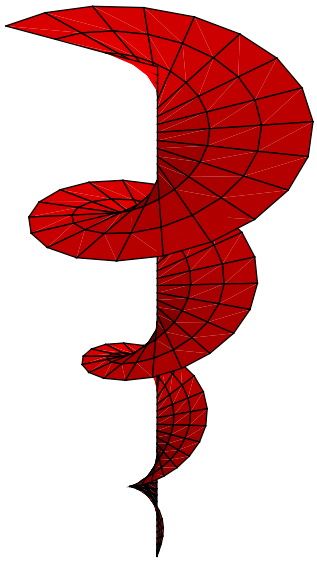
vista. Le opzioni sono le seguenti:

option name	*default value*	
`SpinOrigin`	`{0, 0, 1.5}`	usato con `SpinDistance` per determinare `ViewPoint`
`SpinDistance`	`2`	usato con `SpinOrigin` per determinare `ViewPoint`
`SpinTilt`	`{0,0}`	specifica gli angoli di Eulero α *e* γ
`SpinRange`	`{0 Degree, 360 Degree}`	specifica il raggio degli angoli entro cui varia α
`RotateLights`	`False`	specifica se le luci debbano o meno ruotare assieme all'oggetto

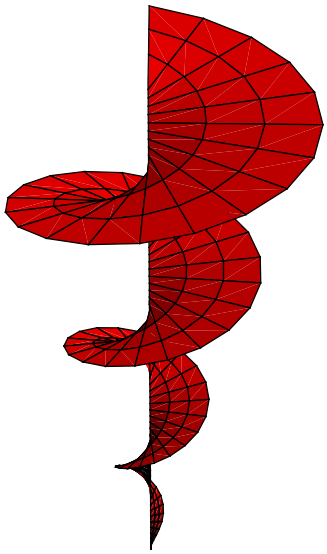
Il comando crea l'animazione creando un punto di vista che ruota attorno ad una sfera di raggio SpinDistance, e traslando il risultato di SpinOrigin. La rotazione usa il primo angolo di Eulero α con le condizione di SpinRange. Inoltre possiamo specificare anche gli angoli β, γ per ottenere altri effetti di rotazione:

Supponiamo di avere la seguente superfice:

In[117]:=
```
superfice = ParametricPlot3D[
   {a b Cos[b], a b Sin[b], b}, {a, 0, 2}, {b, 0, 7 Pi},
   Axes → False, Boxed → False, BoxRatios → {1, 1, 2},
   PlotPoints → {4, 70}, LightSources → {{{3, 3, 3}, Hue[5]}}
  ];
```



In[118]:=
```
SpinShow[superfice, Frames → 20, SpinRange → {0, 4 Pi}]
```



Toh, una vite che si avvita da sola... Certo che sono proprio un simpaticone ed un mostro di simpatia....

■ Graphics`ComplexMap`

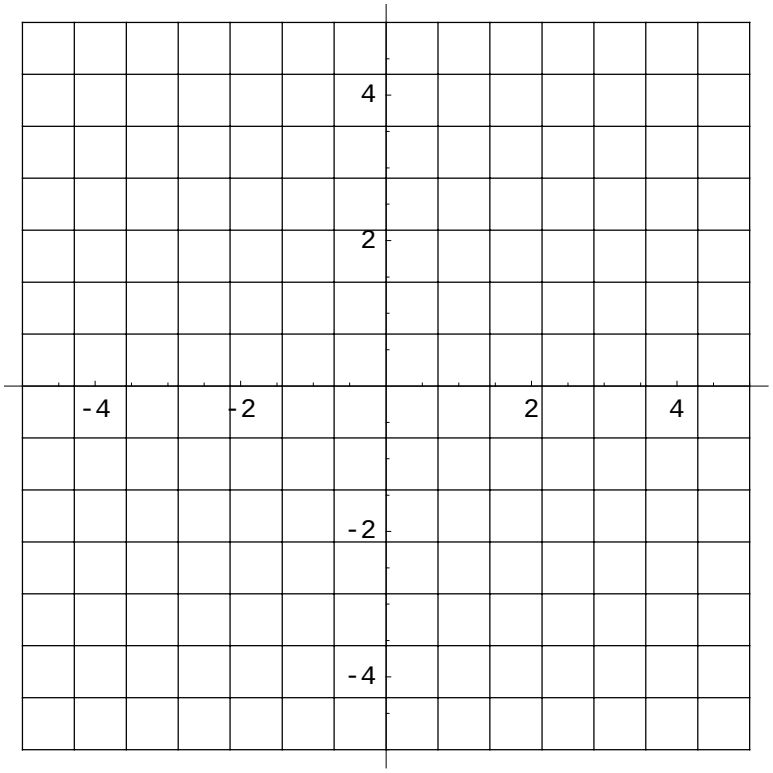
Questo package permette di rappresentare in maniera classica le funzioni complesse di variabile complessa, vedendo come viene trasformata una griglia di linee nel piano bidimensionale dal dominio al codominio.

`CartesianMap[`*f*`, {`*xmin*`, `*xmax*`}, {`*ymin*`, `*ymax*`}, {`*rmin*`, `*rmax*`}, {`*thetamin*`,`	disegna l'immagine in coordinate cartesiane, nel range specificato, della funzione *f*
`PolarMap[`*f*`,`	restituisce l'immagine con le linee in coordinate polari *f*

Vediamo le griglie formate da questa funzione, senza deformazione:

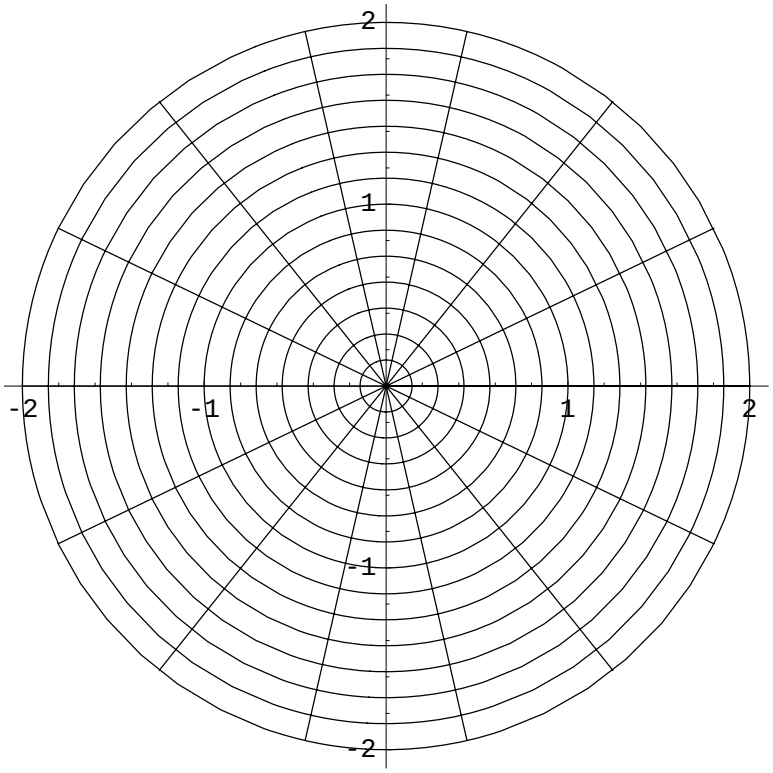
In[119]:=
```
<< Graphics`ComplexMap`
```

In[120]:=
```
CartesianMap[Identity, {-5, 5}, {-5, 5}]
```



Out[120]= - Graphics -

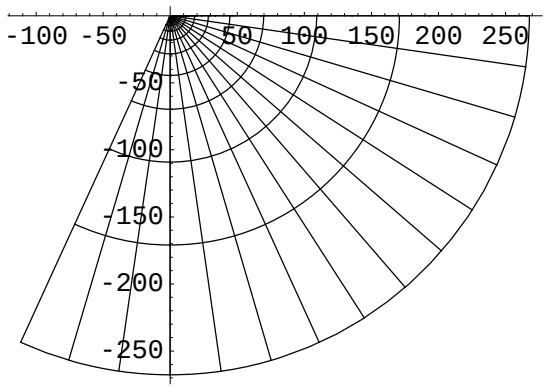
In[121]:=
```
PolarMap[Identity, {0, 2}, {0, 2 Pi}]
```



Out[121]= - Graphics -

Se vogliamo adesso applicare la funzione, sostituiamo ad Identity la funzione di nostro interesse:

In[122]:=
```
CartesianMap[Cos, {0, 2}, {0, 2 Pi}]
```

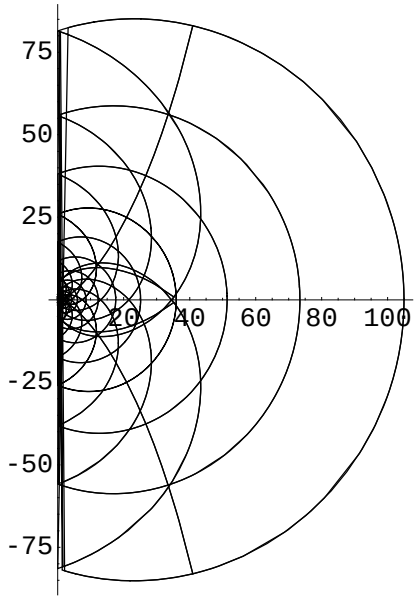


Out[122]= - Graphics -

Notate come sia necessario solamente l'head per creare la funzione. Questo, da un lato, permette di scrivere di meno. Tuttavia se dobbiamo inserire delle funzioni personalizzate dobbiamo o definirle prima, oppure andare ad usare le funzioni pure. Per esempio, potrei voler mappare qualcosa del tipo:

In[123]:=
```
mappa[x_] := Sqrt[ Cos[x]]
```

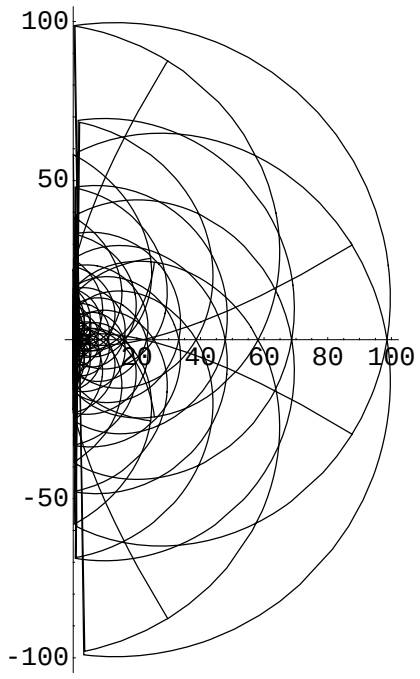
In[124]:=
```
PolarMap[mappa, {0, 10}, {0, 2 Pi}]
```



Out[124]=
```
- Graphics -
```

Potevamo anche utilizzare, naturalmente, una funzione pura, per ottenere lo stesso risultato, evitando di andare a definire prima la funzione; vediamolo con il seno, stavolta:

In[125]:=
```
PolarMap[Sqrt[Sin[#]] &, {0, 10}, {0, 2 Pi}]
```



Out[125]=
```
- Graphics -
```

Come potete vedere la rappresentazione è semplice, invece di andare a creare una lista di funzioni come sarebbe stato necessario se avessimo utilizzato solamente il comando Plot.

■ Graphics`FilledPlot`

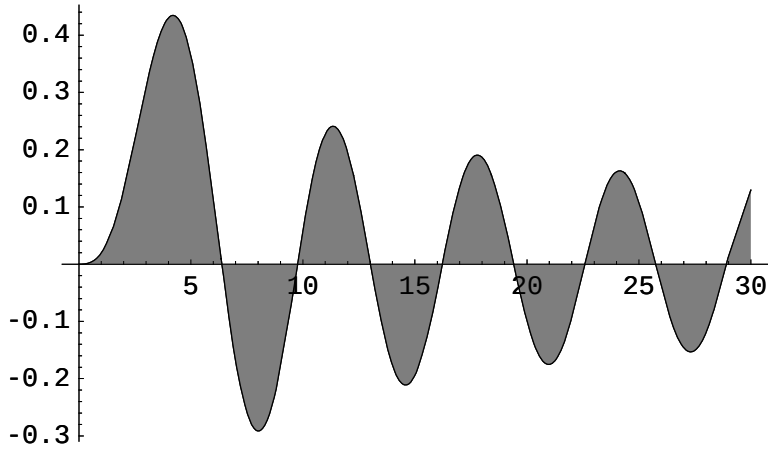
Con questo package non si crea della matematica od altre cose importanti, ma a mio avviso permette di creare grafici bidimensionali con un aspetto più accattivante senza dover andare a giocherellare con le opzioni, riempendendoli (cosa che le opzioni di Plot non fanno):

`FilledPlot[`*f*`, {`*x*`, `*xmin*`, `*xmax*`}]`	disegna *f* nella variabile *x*, riempendo lo spazio fra la funzione e l'asse delle ascisse
`FilledPlot[{`f_1`, `f_2`, ... }, {`*x*`, `*xmin*`, `*xmax*`}]`	disegna f_i, riempendo lo spazio fra due successive coppie di curve con colori differenti

Il funzionamento è molto semplice:

In[126]:=
```
<< Graphics`FilledPlot`
```

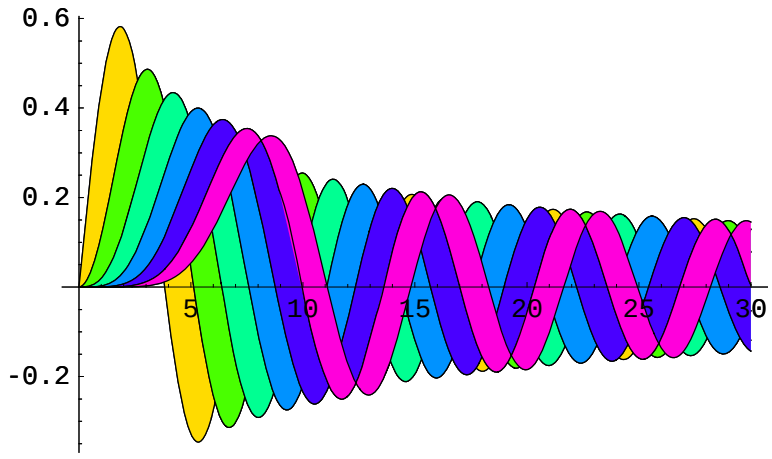
In[127]:=
```
FilledPlot[BesselJ[3, x], {x, 0, 30}]
```



Out[127]= - Graphics -

Possiamo anche riempire gli spazi di funzioni successive:

In[128]:=
```
FilledPlot[
 Evaluate[
  Table[
   BesselJ[n, x], {n, 7}
  ]
 ],
 {x, 0, 30}
]
```



Out[128]= - Graphics -

I colori, di default, sono scelti automaticamente. Tuttavia ci sono opzioni che permettono di modificare il tipo di colori utilizzati per il riempimento dello spazio fr ale funzioni:

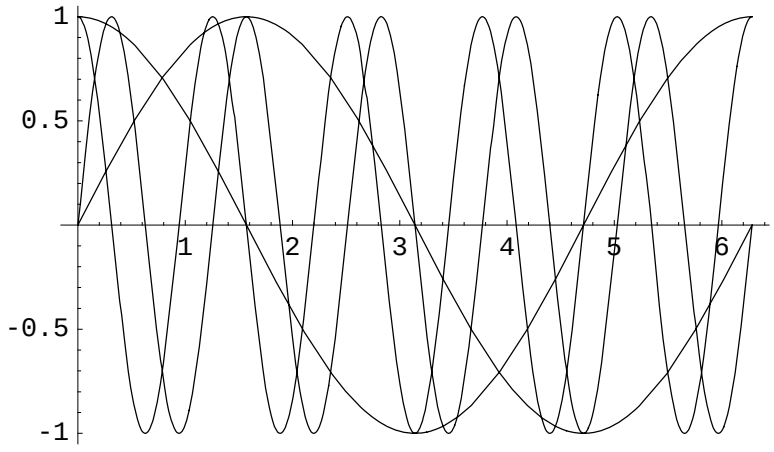
option name	*default value*	
`Fills`	`Automatic`	curve e stili usate per il riempimento
`Curves`	`Back`	il modo in cui vengono visualizzate le curve

Fills lavora in diversi modi: possiamo specificare direttamente i colori utilizzati, in maniera che fra la prima funzione e la seconda ci sia un colore, fra la seconda e la terza un altro e così via, scrivendo qualcosa come `Fills -> {`$color_1$`, `$color_2$`, … }`. Se, invece, vogliamo specificare il colore che riempe lo spazio fra due curve specifiche della lista possiamo utilizzare `Fills -> {{{`$curve_{11}$`, `$curve_{21}$`}, `$color_1$`}, … }`, dove `{`$curve_{11}$`, `$curve_{21}$`}` sono due numeri che rappresentano le posizioni, nella lista, delle funzioni che prendiamo in considerazione, e riempiamo lo spazio fra di loro con il colore specificato. Se vogliamo indicare l'asse delle ascisse utilizziamo, al posto del numero, la parola Axis. Invece, il parametro Curve determina se dobbiamo disegnare le curve delle funzioni dietro i colori, davanti oppure non dobbiamo disegnarle.

Consideriamo la seguente lista di funzioni:

In[129]:=
```
lista = {Cos[x], Sin[x], Cos[5 x], Sin[5 x]};
```

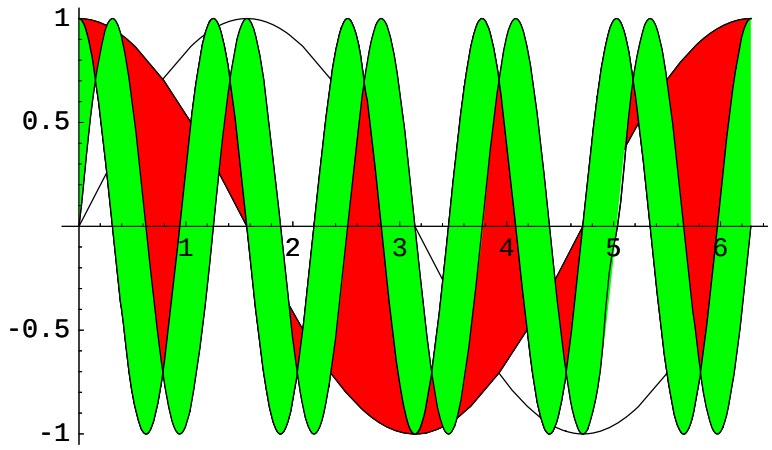
In[130]:=
```
Plot[Evaluate[lista], {x, 0, 2 Pi}]
```



Out[130]= `- Graphics -`

Supponiamo di voler riempire lo spazio fra la prima e la terza funzione della lista, ed anche fra la terza e la quarta, rispettivamente con il rosso e con il verde:

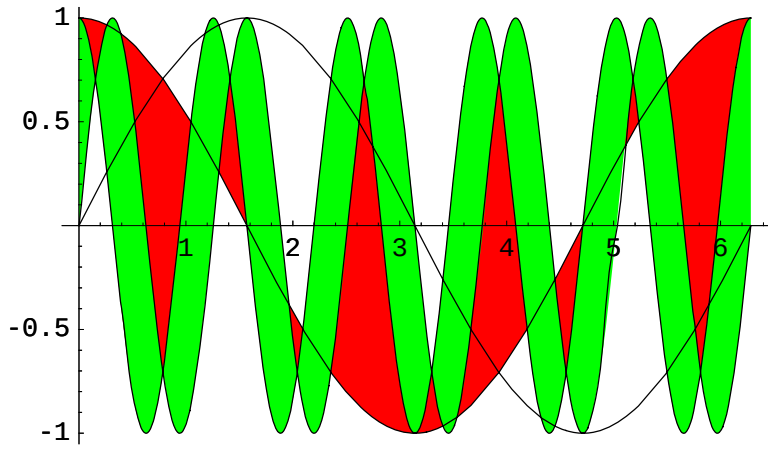
In[131]:=
```
FilledPlot[Evaluate[lista], {x, 0, 2 Pi},
 Fills → {{{1, 3}, Red}, {{3, 4}, Green}}]
```



Out[131]= `- Graphics -`

Abbiamo riempito solamente quello che ci serviva. Vedete anche come le linee dei grafici siano nascosti dal colore. Se vogliamo mostrarli dobbiamo specificarlo con l'altra opzione:

In[132]:=
```
FilledPlot[Evaluate[lista], {x, 0, 2 Pi},
 Fills → {{{1, 3}, Red}, {{3, 4}, Green}}, Curves → Front]
```



Out[132]= `- Graphics -`

Ed in questo modo abbiamo specificato di voler vedere le linee dei grafici.

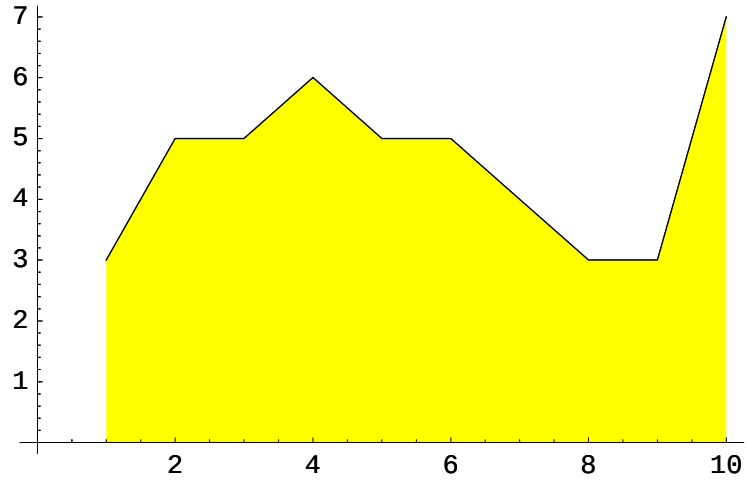
Abbiamo anche analoghi comando per le liste di dati, non solo per le funzioni:

`FilledListPlot[{`y_1`,` y_2`,` … `}]`	genera dei grafici di dati dati da `{1,` y_1`}`, {2, y_2}, … colorando lo spazio fra il grafico e l'asse delle
`FilledListPlot[{{`x_1`,` y_1`}, {`x_2`,` y_2`}, … }]`	genera un grafico, riempendo lo spazio fra la curve dei dati e l'asse *x* delle ascisse
`FilledListPlot[`$data_1$`,` $data_2$`, … ]`	genera un grafico con colori fra le curve specificate dai dati specificati $data_i$

Le opzioni in questo caso sono analoghe a quelle viste prima:

In[133]:=
```
lista = {3, 5, 5, 6, 5, 5, 4, 3, 3, 7};
```

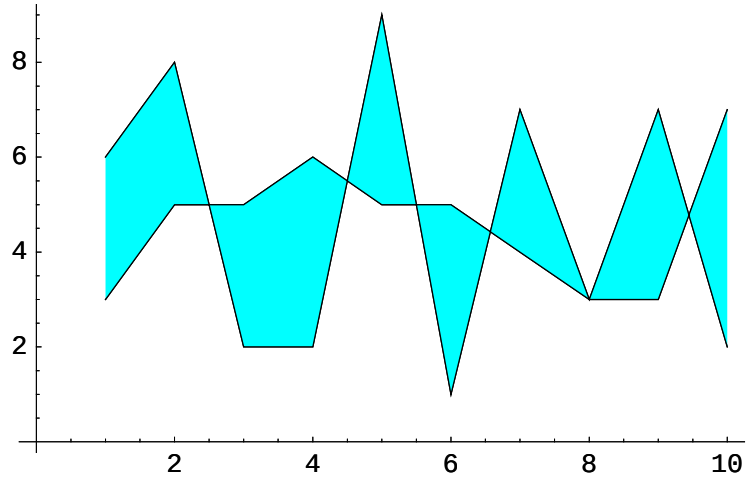
In[134]:=
```
FilledListPlot[lista, Fills → Yellow]
```



Out[134]=
```
- Graphics -
```

In[135]:=
```
lista2 = {6, 8, 2, 2, 9, 1, 7, 3, 7, 2};
```

In[136]:=
```
FilledListPlot[lista, lista2]
```



Out[136]=
```
- Graphics -
```

Un package piccolo ma simpatico, non trovate?

■ Graphics`Graphics`

Questo package aggiunge nuovi comandi che permettono di disegnare particolari tipi di grafici, che sarebbe difficoltoso da fare con i comandi predefiniti di Mathematica: per esempio, i grafici logaritmici, oppure quelli polari:

`LogPlot[`*f*`, {`*x*`, `*xmin*`, `*xmax*`}]`	genera un grafico lineare-logaritmico di *f* in funzione di *x* da *xmin* a *xmax*
`LogLinearPlot[`*f*`, {`*x*`, `*xmin*`, `*xmax*`}]`	genera un grafico logaritmico-lineare di *f*
`LogLogPlot[`*f*`, {`*x*`, `*xmin*`, `*xmax*`}]`	genera un grafico logaritmico-logaritmico di *f*
`LogListPlot[{{`x_1`, `y_1`}, {`x_2`, `y_2`}, … }]`	genera un grafico lineare logaritmico dei dati (x_1,y_1), …
`LogLinearListPlot[{{`x_1`, `y_1`}, {`x_2`, `y_2`}, … }]`	genera un grafico logaritmico-lineare dei dati (x_1,y_1), …
`LogLogListPlot[{{`x_1`, `y_1`}, {`x_2`, `y_2`}, … }]`	genera un grafico logaritmico-logaritmico dei dati (x_1,y_1), …
`LogListPlot[{`y_1`, `y_2`, … }]`, `LogLinearListPlot[{`y_1`, `y_2`, … }]`, `LogLogListPlot[{`y_1`, `y_2`, … }]`	grafici dei punti y_1, y_2, … con valori di *x* 1, 2, …

Questo package è utile, per esempio, per il tracciamento dei diagrammi di Bode oppure di Nyquist, dato che permette di lavorare con grafici logaritmici e polari:

In[137]:=
```
<< Graphics`Graphics`
```

In[138]:=
```
tf[ω] := 3 (I ω - 2) (I ω - 15) / ((ω^2 + 4 I ω + 2) (I ω - 10)^5)
```

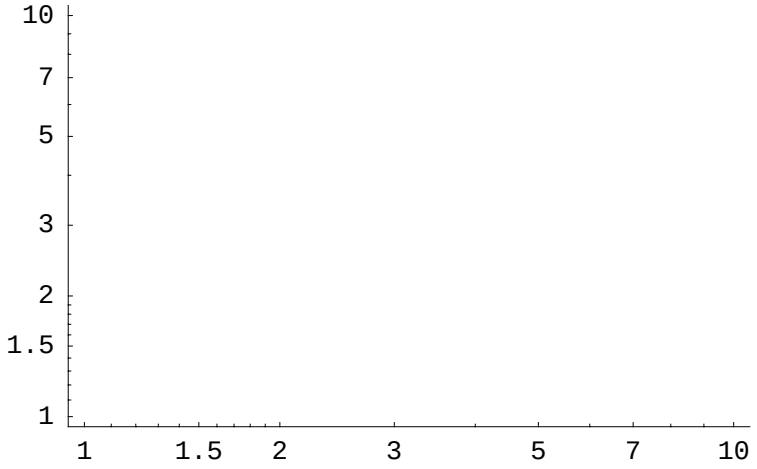
In[139]:=
```
LogLogPlot[Abs[tf[ω]], {ω, 1, 100}]
```

— *ParametricPlot::pptr :* $\left\{\frac{Log[\omega]}{Log[10]}, \frac{Log[Abs[tf[\omega]]]}{Log[10]}\right\}$ *does not evaluate to a pair of real numbers at* $\omega = 1.000004125`$. More…

— *ParametricPlot::pptr :* $\left\{\frac{Log[\omega]}{Log[10]}, \frac{Log[Abs[tf[\omega]]]}{Log[10]}\right\}$ *does not evaluate to a pair of real numbers at* $\omega = 5.0161321657186635`$. More…

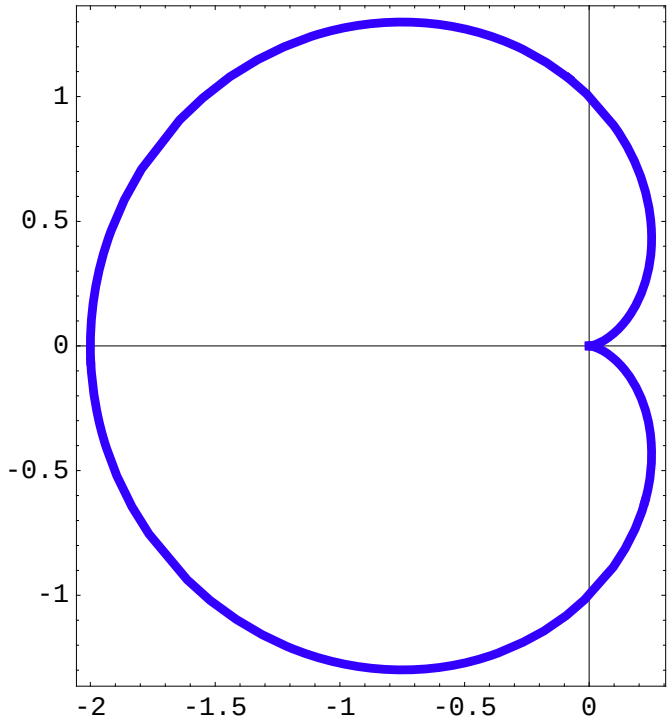
— *ParametricPlot::pptr :* $\left\{\frac{Log[\omega]}{Log[10]}, \frac{Log[Abs[tf[\omega]]]}{Log[10]}\right\}$ *does not evaluate to a pair of real numbers at* $\omega = 9.396071186077995`$. More…

— *General::stop : Further output of ParametricPlot::pptr will be suppressed during this calculation.* More…



Out[139]= - Graphics -

In[140]:=
```
PolarPlot[1 - Cos[ω], {ω, 0, 2 π}, Frame → True,
 PlotStyle → {{Hue[0.7], Thickness[0.015]}}]
```



Out[140]= `- Graphics -`

Come potete vedere, le opzioni sono simili a quelle che si hanno per i comandi standard.

Inoltre, *Mathematica* con questo package è anche in grado di eseguire diversi tipi di grafico utili per la rappresentazione dei dati, come quelli che si vedono, per esempio, in Excel: grafici a torta, a barre e così via.

`BarChart[`*datalist*$_1$`,` *datalist*$_2$`,` … `]`	genera un grafico a barre a partire dai set di dati
`GeneralizedBarChart[`*datalist*$_1$`,` *datalist*$_2$`,` … `]`	genera un grafico a barre dei seti di dati, dove *i* dati specificano la posizione, *l'* altezza *e* la larghezza delle barre
`StackedBarChart[`*datalist*$_1$`,` *datalist*$_2$`,` … `]`	genera un grafico a barre impilate
`PercentileBarChart[`*datalist*$_1$`,` *datalist*$_2$`,` … `]`	genera un grafico a barre impilate dei dati, dove i dati di ogni grupo di barre sono scalati in modo che il valore assoluto della somma si auguale ad 1

Supponiamo di avere le tre seguenti liste di dati:

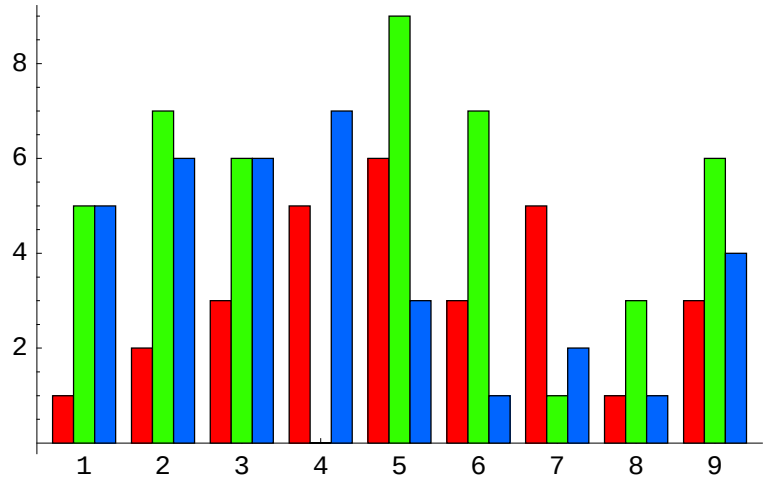
In[141]:=
```
dati1 = {1, 2, 3, 5, 6, 3, 5, 1, 3};
```

In[142]:=
```
dati2 = {5, 7, 6, 0, 9, 7, 1, 3, 6};
```

In[143]:=
```
dati3 = {5, 6, 6, 7, 3, 1, 2, 1, 4};
```

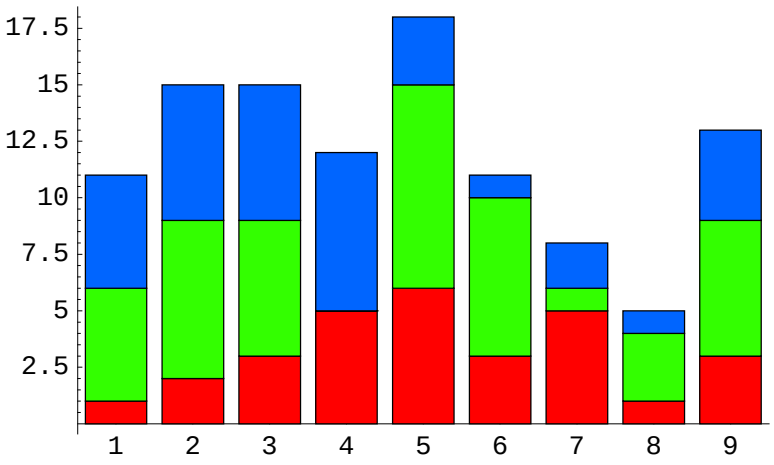
Vediamo la differenza fra i tipi di grafico ottenibili:

In[144]:=
```
BarChart[dati1, dati2, dati3]
```



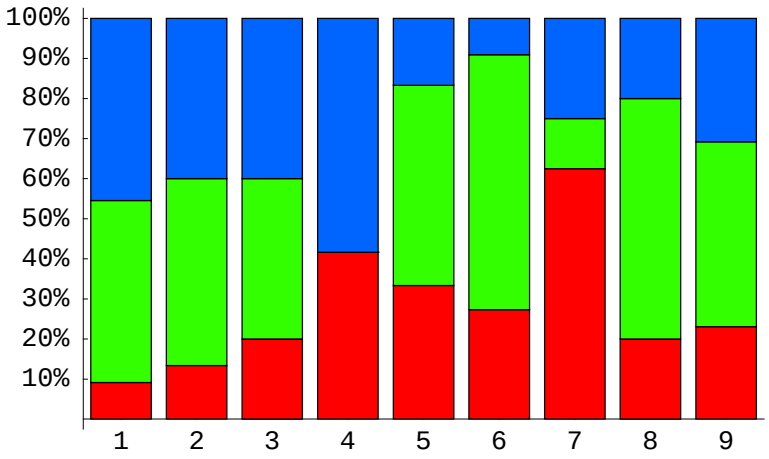
Out[144]= `- Graphics -`

In[145]:=
```
StackedBarChart[dati1, dati2, dati3]
```



Out[145]= `- Graphics -`

In[146]:=
```
PercentileBarChart[dati1, dati2, dati3]
```

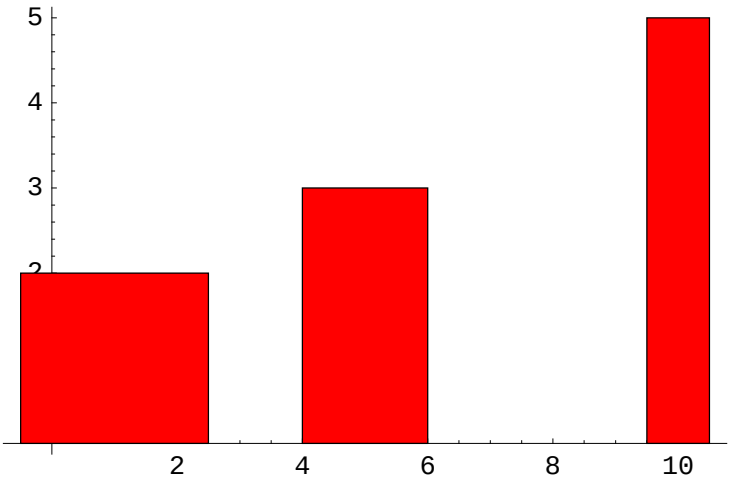


Out[146]= - Graphics -

Nel caso del comando GeneralizedBarChart, ogni elemento della lista dei dati è a sua volta una lista di tre elementi, dove il primo elemento rappresenta la posizione della barra nell'asse delle ascisse, il secondo elemento rappresenta l'altezza della barra, ed il terzo la larghezza:

In[147]:=
```
dati = {{1, 2, 3}, {10, 5, 1}, {5, 3, 2}};
```

In[148]:=
```
GeneralizedBarChart[dati]
```



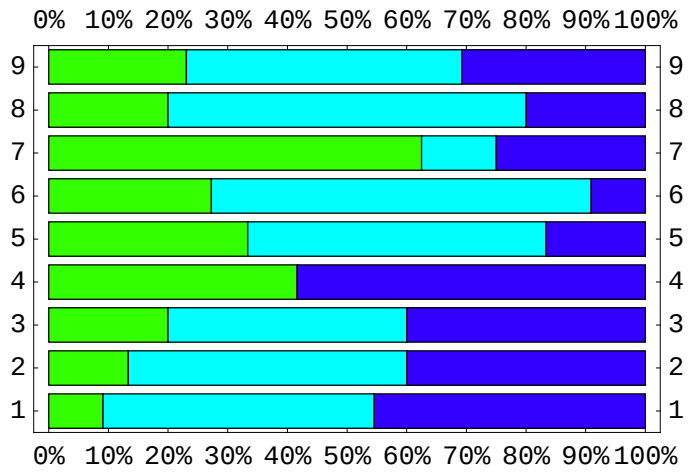
Out[148]= - Graphics -

Naturalmente abbiamo delle opzioni che ci permettono di poter formattare i grafici a barre:

option name	*default value*	
`BarOrientation`	`Vertical`	determina l'orientamento delle barre: `Vertical` oppure
`BarStyle`	`Automatic`	determina lo stile delle barre; per un singolo set di dati determina l'aspetto di ogni barra. Per più set di dati determina lo stile di ogni singolo set
`BarLabels`	`Automatic`	determina le etichette da applicare nei marker della posizione
`BarEdges`	`True`	specifica se bisogna disegnare il contorno delle barre
`BarEdgeStyle`	`Automatic`	detrermina lo stile per il contorno delle barre

Oltre a queste opzioni, i comandi accettano anche la maggior parte delle opzioni utilizzate dai comandi di grafica standard. Vediamo di ridisegnare, per esempio, il secondo grafico che abbiamo fatto, con qualche opzione in più...

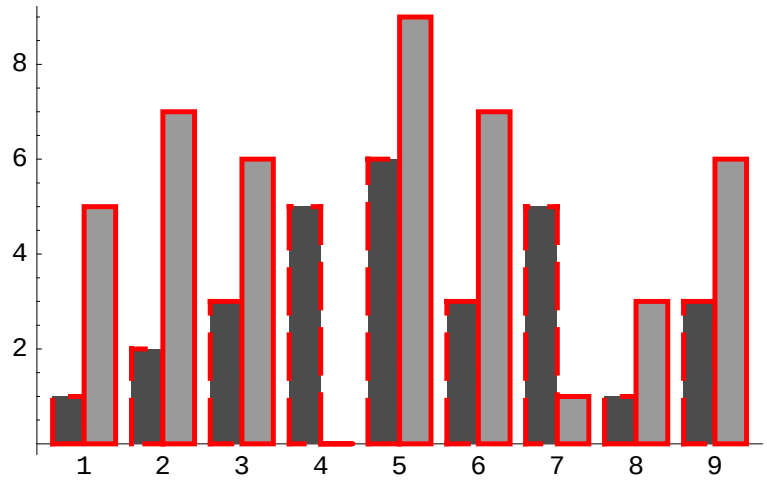
In[149]:=
```
PercentileBarChart[dati1, dati2, dati3,
 BarStyle → {Hue[0.3], Hue[0.5], Hue[0.7]},
 BarOrientation → Horizontal,
 Axes → False,
 Frame → True
]
```



Out[149]= `- Graphics -`

Vediamo un altro esempio:

In[150]:=
```
BarChart[dati1, dati2,
 BarStyle → {GrayLevel[0.3], GrayLevel[0.6]},
 BarEdgeStyle → {{Red, Thickness[0.007], Dashing[{0.06, 0.03}]},
   {Red, Thickness[0.007]}}
]
```



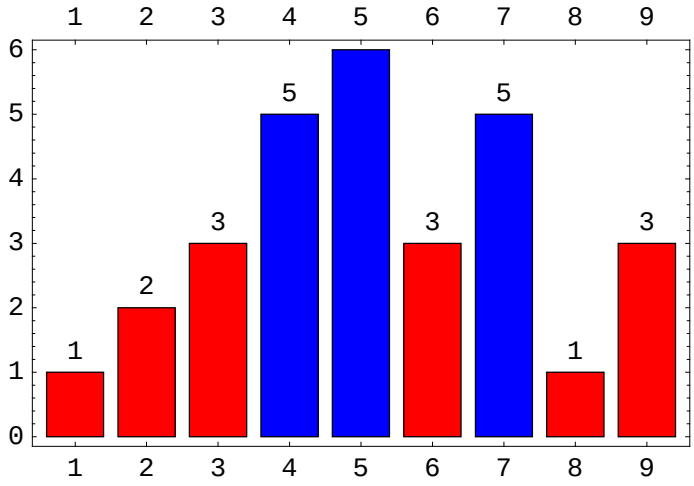
Out[150]= `- Graphics -`

Ci sono alche altre opzioni per una formattazione migliore

`BarValues -> True`	etichetta ogni barra con il suo valore
`BarStyle ->` *function*	colora ogni barra applicando la funzione definita all'altezza della barra stessa
`BarSpacing -> Automatic`	Specifica lo spazio che intercorre fra le barre, come frazione della larghezza di una singola barra
`BarGroupSpacing -> Automatic`	Specifica lo spazio che intercorre fra ogni gruppo di barre, come frazione della larghezza di una singola barra

Ovviamente l'ultima opzione vale solamente per il comando BarChart, dato che negli altri non ci sono gruppi, ma le barre sono impilate fra di loro:

In[151]:=
```
BarChart[dati1,
 BarValues → True,
 BarStyle → (If[# > 3, Blue, Red] &),
 Frame → True,
 Axes → False
]
```



Out[151]= `- Graphics -`

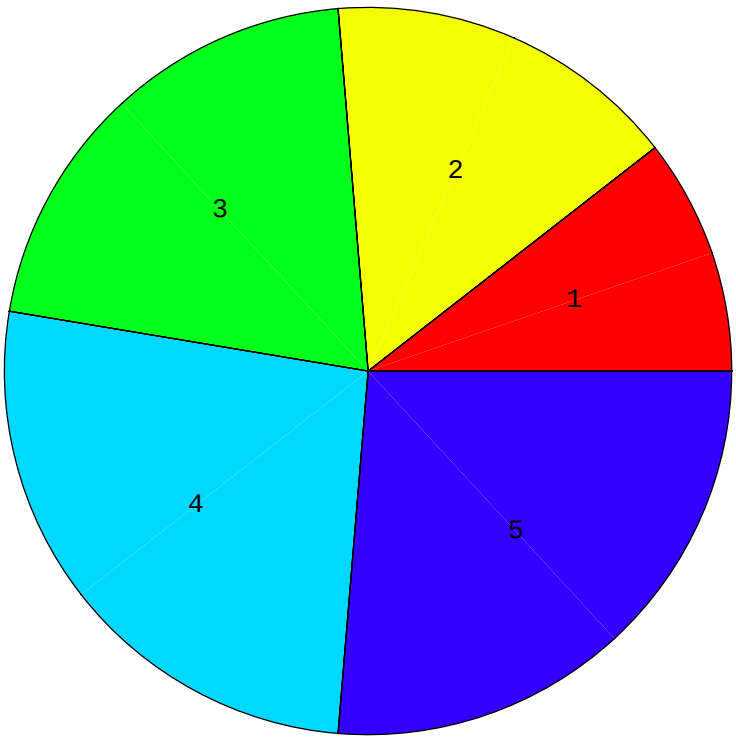
Un altro tipo di grafico molto utilizzato per la presentazione dei dati è il grafico a torta:

`PieChart[`*data*`]`	genera un grafico a torta dei dati

Consideriamo questa volta i seguenti dati:

In[152]:=
```
listatorta = {2, 3, 4, 5, 5};
```

In[153]:= **PieChart[listatorta]**



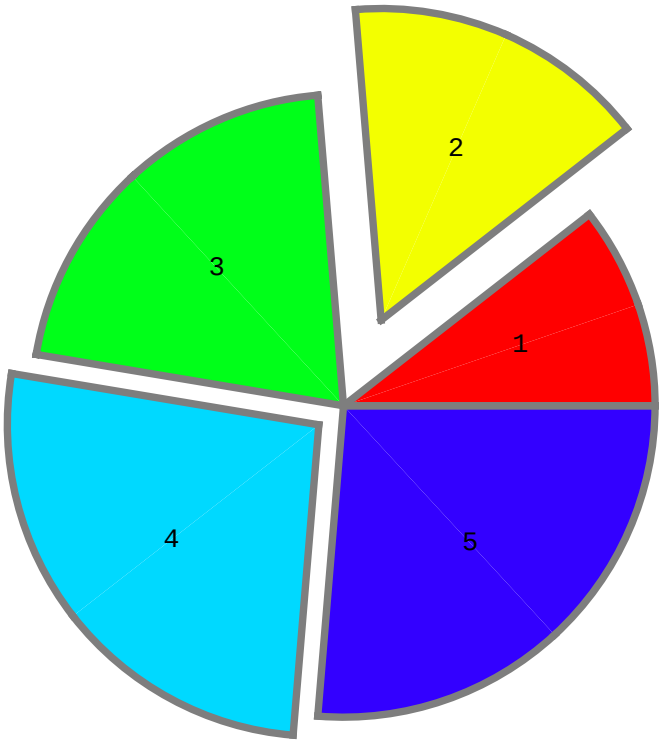
Out[153]= **- Graphics -**

Anche in questo caso sono presenti varie opzioni per personalizzare il grafico ottenuto:

`PieStyle ->` *stylelist*	specifica gli stili che devono essere usati (in maniera ciclica) per la sequenza delle parti del grafico a torta che rappresentano i singoli dati
`PieLineStyle ->` *linestyle*	specifica lo stile che deve essere utilizzato per le linee di bordo del grafico
`PieLabels ->` *lablelist*	specifica le etichette che verranno applicate (ciclicamente), alle 'fette' della torta del grafico che rappresentano i singoli dati
`PieExploded -> All`	genera un grafico *a* torta esploso
`PieExploded -> {`$wedge_1$`,` $wedge_2$`, … }`	esplode solamente le fette corrispondenti agli indici $wedge_1$, $wedge_2$, …

Sempre per i dati di prima, possiamo mostrare il grafico a torta con delle specifiche fette esplose:

In[154]:=
```
PieChart[listatorta,
 PieExploded → {{2, .3}, {4, .1}},
 PieLineStyle → {Thickness[0.011], Gray},
 PieLabels → (# &)
]
```



Out[154]= `- Graphics -`

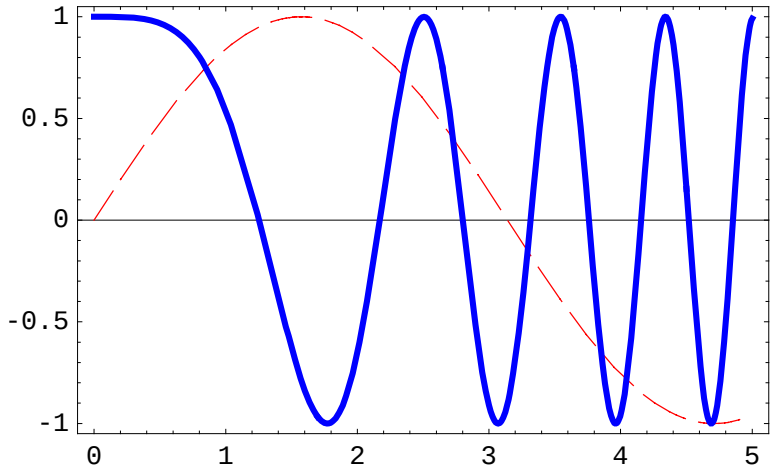
Abbiamo fatto parecchie cosette: prima di tutto, abbiamo notato come nell'opzione PieExploded, oltre a specificare quale fetta debba essere esplosa, abbiamo anche visto che possiamo specificare di quanto debba essere staccata. Poi, invece di indicare le fette con un indice generico, ho creato una funzione pura che restituisce il valore del dato, per cui questo stesso valore è rappresentato all'interno della fetta; meglio sicuramente di avere un indice generico, a mio avviso, ma poi si può creare quello che si vuole, naturalmente. Infine ho anche aggiustato le linee di contorno. Non sarà il massimo dell'estetica, ma è per farvi vedere come funziona...

Un altro aspetto interessante è quando decidiamo di mostrare più diagrammi assieme; il modo convenzionale che si utilizza è quello di realizzare i singoli grafici considerando l'opzione `Display :> Function -> Identity` , e poi mostrarli assieme mediante Show. Tuttavia, per evitare di dover disegnare i grafici prima uno ad uno, questo package include dei comandic he automatizzano questo lavoro:

`DisplayTogether[`*plot*$_1$`,` *plot*$_2$`, … ,` *opts*`]`	combina i grafici ottenuto dai comandi di grafica che ha come argomento, restituendo un unico oggetto
`DisplayTogetherArray[`*plotarray*`,` *opts*`]`	combina i grafici definiti da *plotarray* (una lista di grafici annidata a matrice) in un oggetto `Graphics`.

Questi comandi accettano le opzioni di Graphics e di GraphicsArray:

In[155]:=
```
DisplayTogether[
 Plot[Sin[x], {x, 0, 5}, PlotStyle → {Dashing[{0.06, 0.02}], Red}
 ],
 Plot[Cos[x^2], {x, 0, 5},
  PlotStyle → {Thickness[0.009], Blue}
 ],
 Frame → True
]
```



Out[155]= `- Graphics -`

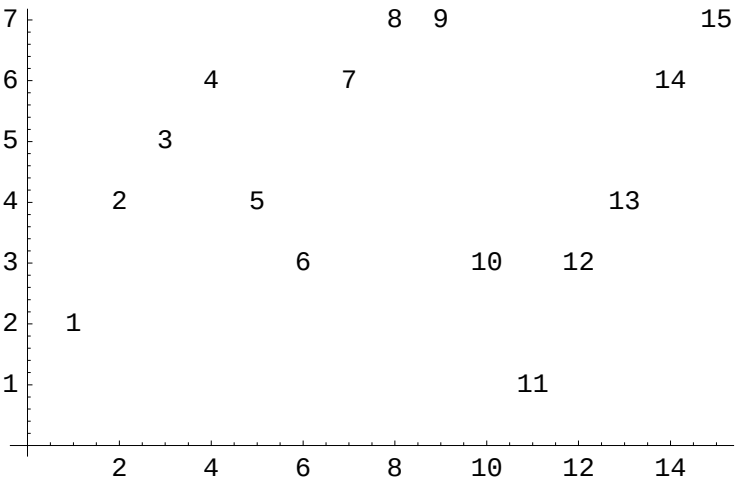
Il package contiene anche comandi per creare tipi di grafico meno usati, ed alcuni anche meno 'ortodossi':

`TextListPlot[{`y_1`, `y_2`, … }]`	disegna il grafico con i dati y_1, y_2, … per valori di x pari a 1, 2, … , mostrando, al posto dei punti, le etichette 1, 2, …
`TextListPlot[{{`x_1`, `y_1`}, {`x_2`, `y_2`}, … }]`	disegna i punti (x_1 , y_1), (x_2 , y_2), … , renderizzandoli come 1, 2, …
`TextListPlot[{{`x_1`, `y_1`, `$expr_1$`}, {`x_2`, `y_2`, `$expr_2$`}, … }]`	disegna i punti (x_1 , y_1), (x_2 , y_2), … renderizzandoli con il testo $expr_1$, $expr_2$, …
`LabeledListPlot[{`y_1`, `y_2`, … }]`	disegna y_1, y_2, … per valori di x pari a 1, 2, … , etichettando i punti con 1, 2, …
`LabeledListPlot[{{`x_1`, `y_1`}, {`x_2`, `y_2`}, … }]`	disegna i punti (x_1 , y_1), (x_2 , y_2), … , etichettandoli con i valori 1, 2, …
`LabeledListPlot[{{`x_1`, `y_1`, `$expr_1$`}, {`x_2`, `y_2`, `$expr_2$`}, … }]`	disegna i punti (x_1 , y_1), (x_2 , y_2), … , etichettandoli con $expr_1$, $expr_2$, …
`ErrorListPlot[{{`y_1`, `d_1`}, {`y_2`, `d_2`}, … }]`	genera un grafico di dati con le barre di errore

Consideriamo una lista di dati numerici, come al solito:

```
In[156]:= data = {2, 4, 5, 6, 4, 3, 6, 7, 7, 3, 1, 3, 4, 6, 7};

— General::spell1 : Possible spelling error: new
    symbol name "data" is similar to existing symbol "dati". More…
```

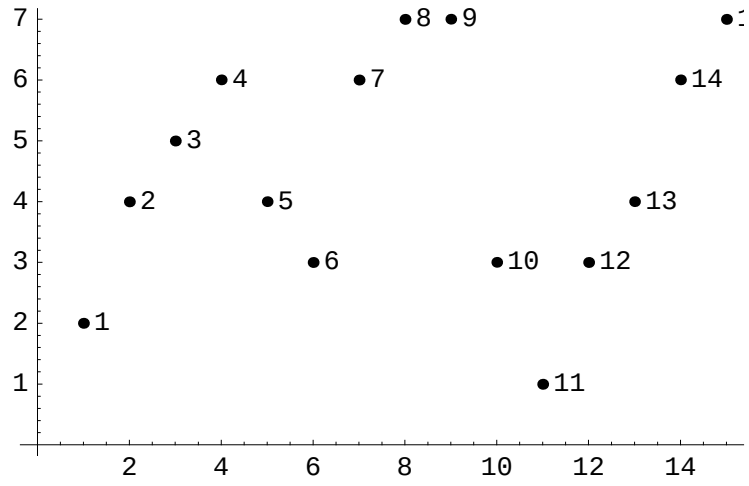
```
In[157]:= TextListPlot[data]
```




```
Out[157]= - Graphics -
```

Tuttavia questo tipo di grafico a me personalmente non piace; trovo più utile il seguente, che è anche più chiaro:

In[158]:=
```
LabeledListPlot[data]
```

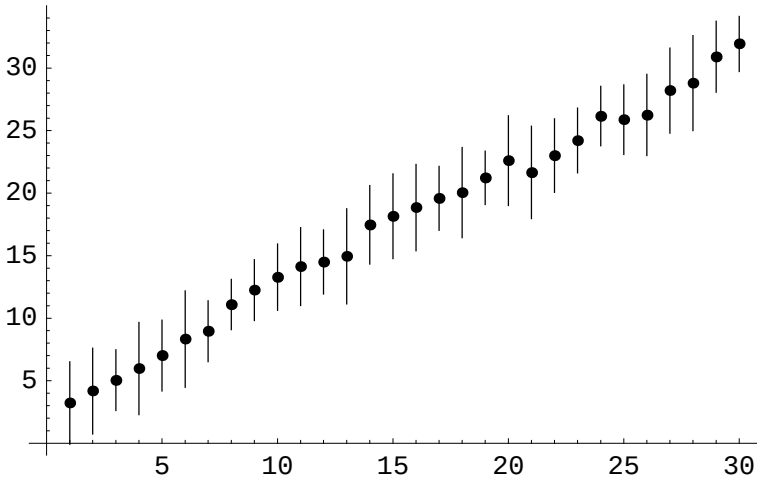


Out[158]=
```
- Graphics -
```

Per risultati sperimentali, è estremamente utile il grafico con i corrispondenti errori, che devono essere comunque misurati a parte:

In[159]:=
```
data = Table[
    {n + Sin[n / 6] + Random[Real, {1, 3}], Random[Real, {2, 4}]}, {n, 30}];
```

In[160]:=
```
ErrorListPlot[data]
```



Out[160]=
```
- Graphics -
```

Naturalmente, questo grafico è molto più utile se i dati hanno un senso...

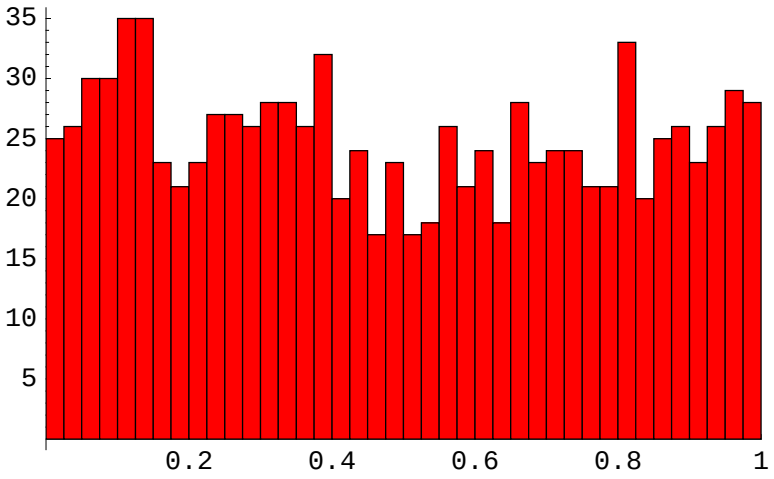
Un altro grafico utile nella rappresentazione di dati sperimentali è dato dall'istogramma:

`Histogram[{`x_1`, `x_2`, … }]`	genera un istogramma dei dati grezzi
`Histogram[{`f_1`, `f_2`, … }, FrequencyData ->`	genera un istogramma dei dati di frequenza, doce, se i valori di taglio sono dati da c_0,c_1,c_2,..., allora f_i rappresenta il numeri di dati in $c_{j-1}<=x<c_j$

Gli istogrammi sono utilizzati per mostrare quanti dati rientrano entro un determinato range di valori, dando una visione maggiormente d'insieme, e più sintetica. Vediamo un esempio con una serie di dati sperimentali:

In[331]:=
```
esperimento = Random[] & /@ Range[0, 1000];
```

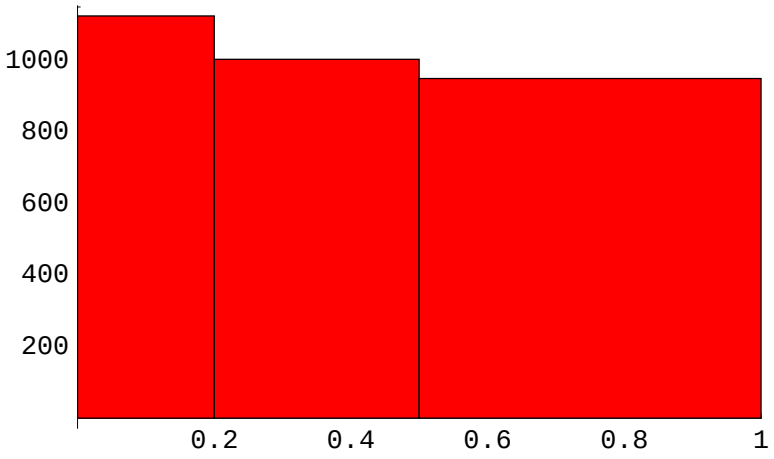
In[332]:=
```
Histogram[esperimento]
```



Out[332]=
```
- Graphics -
```

Di default *Mathematica* sceglie intervalli formati da numeri semplici:

In[334]:=
```
Histogram[esperimento,
 HistogramCategories → {0, .2, .5, 1}]
```



Out[334]= `- Graphics -`

In questo caso abbiamo definito degli intervalli definiti da noi, personalizzando in questa maniera il grafico. Ci sono alcune opzioni importanti per questo tipo di grafico:

option name	*default value*	
`ApproximateIntervals`	`Automatic`	specifica se i limiti degli intervalli debbano essere o meno approssimati a dei numeri semplici; può assumere valore `True`, `False`, o `Automatic`
`FrequencyData`	`False`	specifica se i dati debbano essere considerati grezzi, per i quali le categorie di frequenze devono essere trovati, oppure se i dati debbano essere considerati già dati di frequenza per gli intervalli specificati con `Histogram Categories`
`HistogramCategories`	`Automatic`	specifica gli intervalli dell'istogramma
`HistogramRange`	`Automatic`	specifica il range dei dati che devono essere inclusi
`HistogramScale`	`Automatic`	specifica se le altezze delle barre debbano essere scalate in modo che le misure della somma delle altezze debba essere unitaria

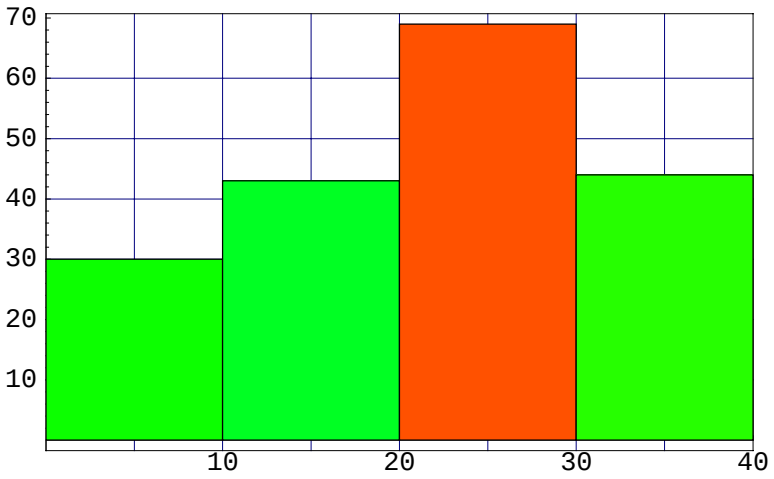
`Ticks -> None`	non disegna marker
`Ticks -> Automatic`	posiziona automaticamente i marker
`Ticks -> Interval Boundaries`	posiziona i marker ai limiti degli intervalle
`Ticks -> IntervalCenters`	posizione i marker ai centri degli intervalli
`Ticks -> {`*xticks*`, `*yticks*`}`	specifica i marker per gli assi

Supponiamo, per esempio, di avere dei dati sperimentali, che dicono che 30 risultati stanno nell'intervallo fra 0 e 10, 43 risultati fra 10 e 20, 69 risultati fra 20 e 30, e 44 risultati fra 30 e 40:

In[164]:=
```
dati = {30, 43, 69, 44};
```

In[165]:=
```
intervalli = {0, 10, 20, 30, 40};
```

In[166]:=
```
Histogram[dati,
 FrequencyData → True,
 HistogramCategories → intervalli,
 BarStyle → (Hue[Random[]] &),
 Frame → True,
 FrameTicks → None,
 GridLines → Automatic,
 Ticks → IntervalBoundaries
]
```



Out[166]= - Graphics -

Potete vedere come, oltre alle opzioni proprie di questo comando, abbia aggiunto anche qualche opzione standard, per fare panza e presenza, e soprattutto per rendere più piacevole l'aspetto del grafico e più efficace

■ Graphics`Graphics3D`

Questo package è analogo a quello visto a prima, per i grafici tridimensionali. Estende il numero di comandi possibili creando nuove tipologie di grafici e nuovi modi per rappresentarli.

Prim di tutto, sono presenti comandi per poter creare un grafico a barre tridimensionale:

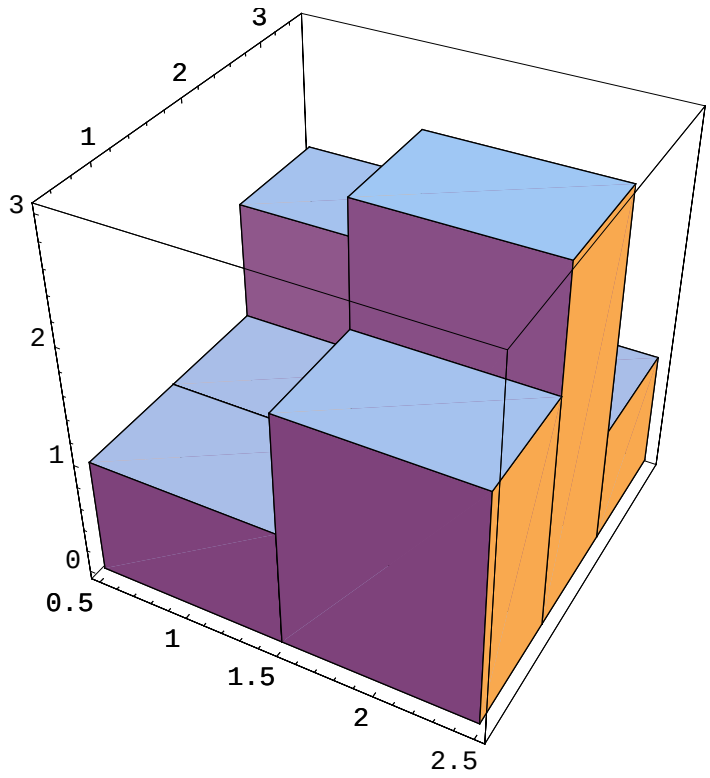
`BarChart3D[{{`z_{11}`,` z_{12}`,` … `}, {`z_{21}`,` z_{22}`,` … `}, … }]`	crea un grafico a barre tridimensionale usando un array bidimensionale delle altezze $z_{x\,y}$
`BarChart3D[{{{`z_{11}`,` *style*$_{11}$`}, {`z_{12}`,` *style*$_{12}$`}, …`	crea un grafico a barre specificando lo stile delle singole barre

Vediamo il funzionamento, che è abbastanza semplice:

In[167]:=
```
<< Graphics`Graphics3D`
```

In[168]:=
```
dati = {{1, 1, 2}, {2, 3, 1}};
```

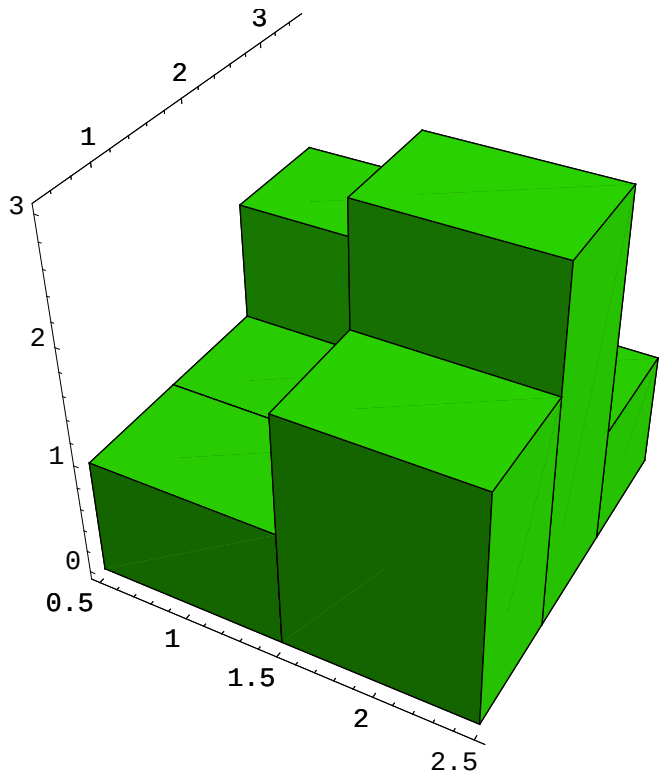
In[169]:=
```
BarChart3D[dati]
```



Out[169]=
```
- Graphics3D -
```

Come potete vedere il comando usa la grafica standard tridimensionale (si riconosce dalle opzioni standard), quindi possiamo anche andare a modificarle come facciamo per i comandi predefiiti:

In[170]:=
```
Show[%, LightSources → {{{5, 4, 6}, Hue[0.3]}}, Boxed → False]
```



Out[170]=
```
- Graphics3D -
```

Oltre a queste opzioni, ci sono anche quelle specifiche per questo comando, che permettono di personalizzare il modo in cui vengono visualizzate le barre:

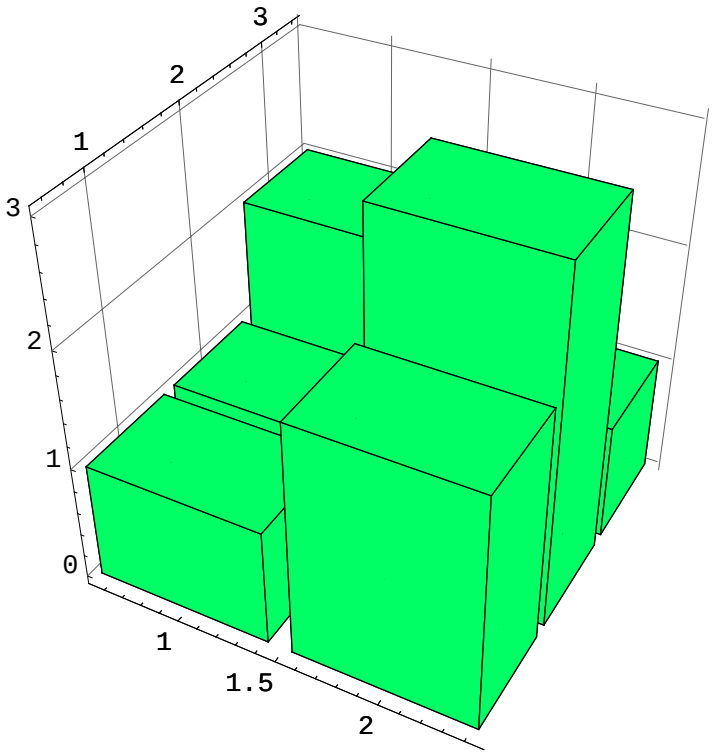
option name	*default value*	
`XSpacing`	`0`	spazio fra le barre nella direzione *x*
`YSpacing`	`0`	spazio fra le barre nella direzione *y*
`SolidBarEdges`	`True`	specifica se disegnare gli spigoli delle barre
`SolidBarEdgeStyle`	`GrayLevel[0]`	specifica lo stile degli spigoli
`SolidBarStyle`	`GrayLevel[0.5]`	specifica lo stile per le facce delle barre

Possiamo utilizzare anche questi comandi per poter disegnare meglio le barre:

In[171]:=
```
massimo = Max[dati]
```

Out[171]=
```
3

3
```

In[172]:=
```
BarChart3D[dati,
 XSpacing → .12,
 YSpacing → .12,
 SolidBarStyle → Hue[0.4],
 Boxed → False,
 Lighting → False,
 FaceGrids → {{0, 1, 0}, {-1, 0, 0}}
]
```



Out[172]=
```
- Graphics3D -
```

Come abbiamo visto, è abbastanza facile personalizzarlo. Un difetto che ho personalmente incontrato in questo comando è che l'opzione SolidBarStyle non accetta funzioni pure, come Hue[#]&, per poter selezionare individualmente il colore di una singola barra; non so se questo si possa fare oppure no, ma non ci sono riuscito... Se qualcuno riesce a farlo, sarebbe molto gentile a farmi sapere come c@##o ha fatto, che io non ci sono riuscito in nessun modo!!!! Lo citerò nella prossima edizione di questi appunti ringranziandolo pubblicamente, naturalmente!

Un altro comando che da questo package riguarda i grafici scatter, cioè a punti, che possiamo

eseguire anche in tre dimensioni (sarebbe l'analogo di ListPlot):

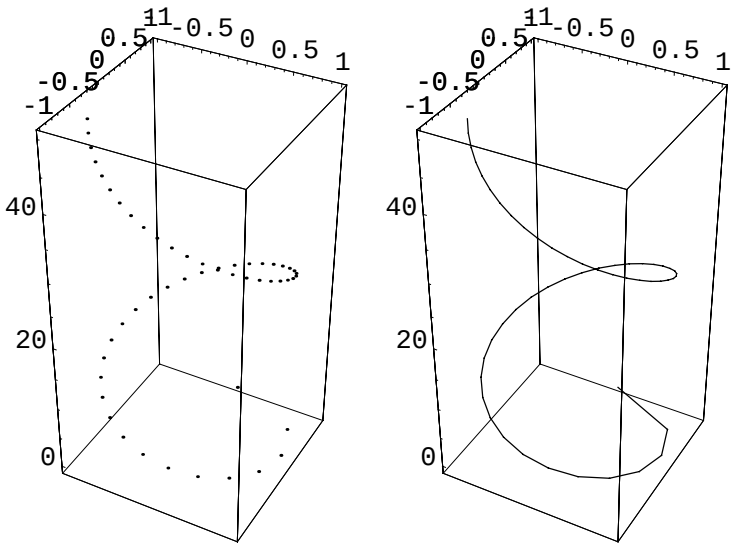
`ScatterPlot3D[{{`x_1`,` y_1`,` z_1`}, {`x_2`,` y_2`,` z_2`},`	genera un grafico scatter nelle tre dimensioni
`ScatterPlot3D[{{`x_1`,` y_1`,` z_1`}, {`x_2`,` y_2`,` z_2`}, … }, PlotJoined ->`	collega assieme *i* punti con una linea
`ListSurfacePlot3D[{{{`x_{11}`,` y_{11}`,` z_{11}`}, {`x_{12}`,` y_{12}`,` z_{12}`}, … }, … }]`	usa la lista di punti per generare una mesh tridimensionale avente i dati come vertici

Notate come, mentre per il caso bidimensionale ListPlot bastava solamente una coordinata del dato, dato che la x era assegnata automaticamente con 1, 2..., in questo caso invece dobbiamo esplicitare per forza tutte e tre le coordinate di ogni singolo punto:

In[173]:=
```
<< Graphics`Graphics`
```

In[174]:=
```
punti = Table[{Sin[x^.6], Cos[x^.6], x}, {x, 0, 50, 1}];
```

In[175]:=
```
DisplayTogetherArray[
 {
  ScatterPlot3D[punti, BoxRatios → {1, 1, 2}],
  ScatterPlot3D[punti, BoxRatios → {1, 1, 2}, PlotJoined → True]
 }
]
```



Out[175]= - GraphicsArray -

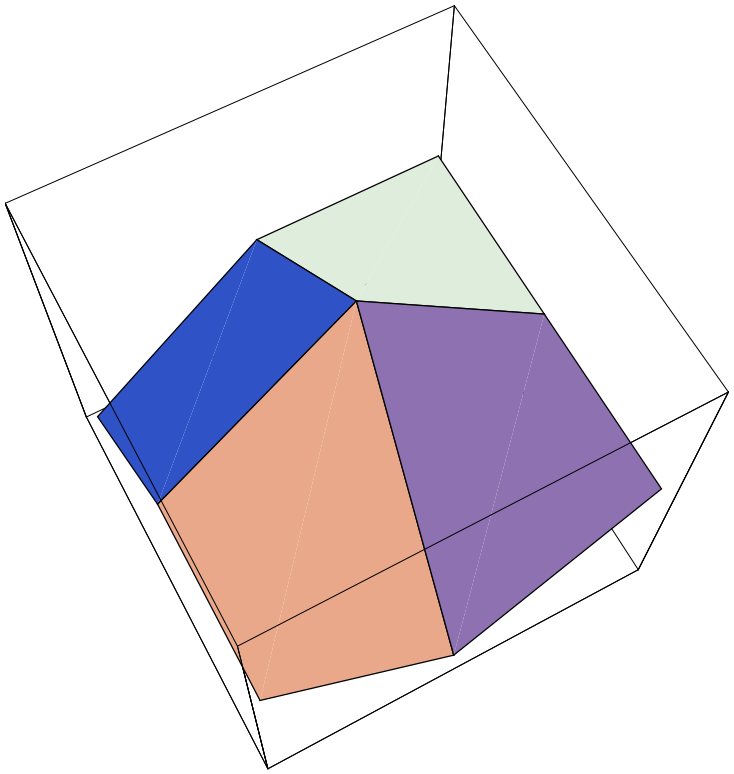
Ho anche usato un comando che abbiamo visto prima... Vedete come apprendere bene i songoli comandi permetta di saperli combinare assieme per raggiungere il risultato ottenuto. Saper utilizzare

tutti i comandi e saperli combinare nella maniera giusta è particolarmente importante, e potete impararlo soltanto con la pratica, perchè solo così saprete che esistono comandi che fanno istantaneamente quello che invece volete fare scrivendo righe e righe di codice...

Possiamo anche rappresentare delle superfici date da liste di punti. Per esempio, potremmo importare dei punti che rappresentano la triangolazione di un territorio, oppure delle serie di misure in cui varia un parametro dell'esperimento, e così via... ogni dato deve contenere le tre coordinate

In[176]:=
```
dati = {
   {{0, 0, 1}, {0, 1, 1}, {0, 2, 1}},
   {{1, 0, 1}, {1, 1, 2}, {1, 2, 0}},
   {{2, 0, 0}, {2, 1, 1}, {2, 2, 1}}
  };
```

In[177]:=
```
ListSurfacePlot3D[dati, ViewPoint -> {0.675, 1.246, 3.073}]
```

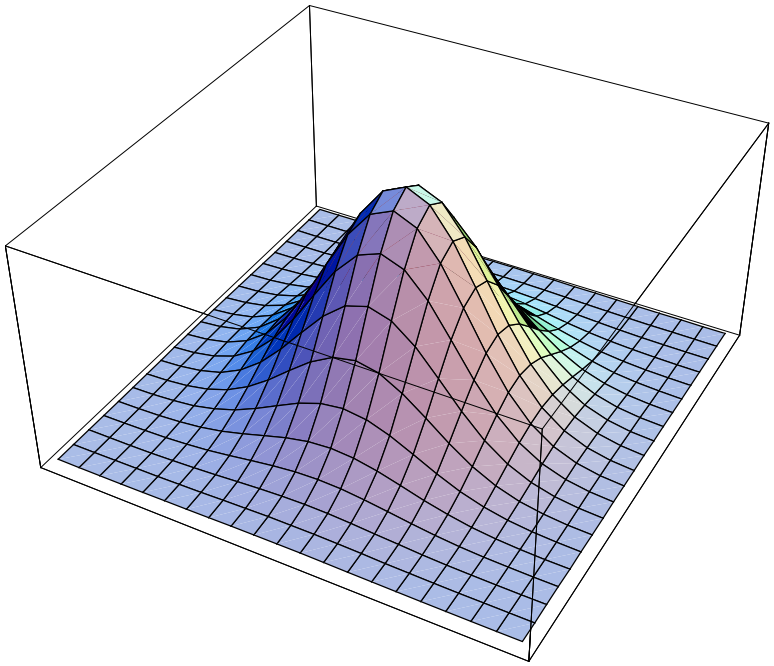


Out[177]= - Graphics3D -

Come potete vedere, occorre che i punti siano spazialmente omogeneri, per ottenere una mesh adeguata: i punti devono essere ordinati in modo da poter rappresentare la prima riga le coordinate della prima riga della mesh, la seconda riga le coordinate della seconda riga della mesh e cos' via. Se utilizziamo punti che si sovrappongono, accadrà lo stesso nel grafico. Per esempio, supponiamo di spostare il punto centrare della lista in modo che si trovi prima della prima riga della mesh. Facciamo un esempio con questa mesh:

In[178]:=
```
dati2 = Table[
   {n, m, Exp[-5 (n^2 + m^2)]},
   {n, -1, 1, .1},
   {m, -1, 1, .1}
  ];
```

In[179]:=
```
ListSurfacePlot3D[dati2, PlotRange → All]
```

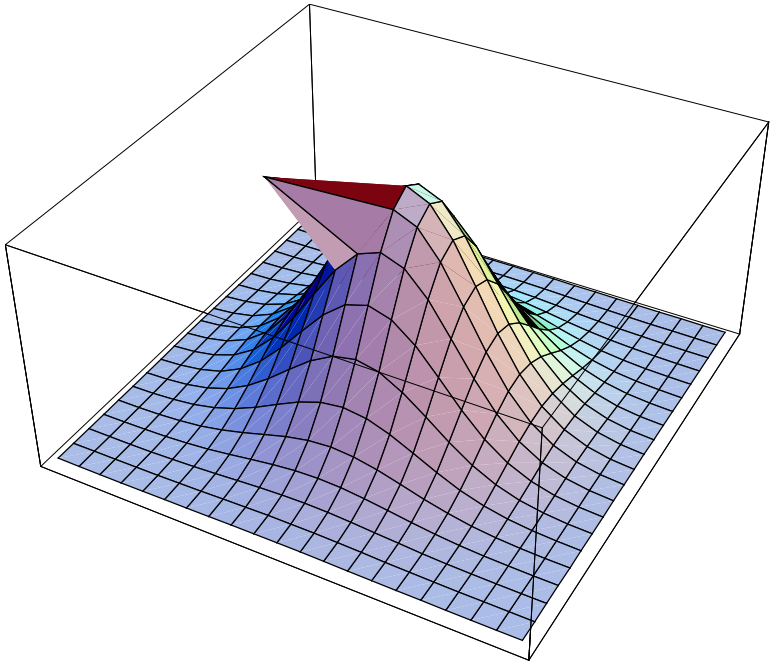


Out[179]=
```
- Graphics3D -
```

Andiamo a modificare il punto centrale della mesh, in modi che le sue coordinate x ed y varino, sovrapponendosi a qualche altro punto della mesh:

In[180]:=
```
dati2[[10, 10]] = {-.5, -.2, 1};
```

In[181]:= **ListSurfacePlot3D[dati2, PlotRange → All]**



Out[181]= **- Graphics3D -**

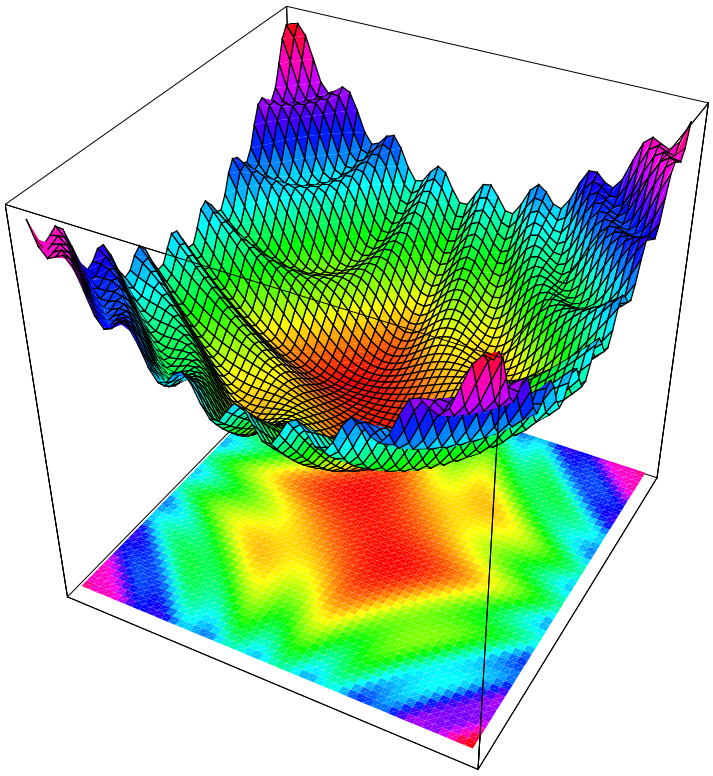
Come vedete, il processo di generazione della mesh ha preso quel punto come appartenente sempre a quella posizione della griglia, deformandola di conseguenza. Questo può essere o meno utile per quello che volete fare, ma in entrambi i casi, state attenti a come impostate l'ordine dei dati nella creazione della matrice.

Un altro comando, a mio avviso molto interessante per la presentazione di un grafico, è il seguente:

`ShadowPlot3D[`*f*`, {`*x*`, `*xmin*`, `*xmax*`}, {`*y*`, `*ymin*`, `	crea un grafico tridimensionale della funzione, e disegna la proiezione nel piano *x*-*y*
`ListShadowPlot3D[{{`z_{11}`, `z_{12}`, … }, {`z_{21}`, `z_{22}`, … }, … }]`	come prima, ma per liste di punti
`Shadow[`*g*`]`	crea delle proiezioni dell'oggetto grafico nei piani degli assi

I comandi sono simili a quelli standard, come pure le opzioni generiche:

In[182]:=
```
ShadowPlot3D[x^2 + y^2 + 3 Sin[x y], {x, -5, 5}, {y, -5, 5},
 PlotPoints → 50,
 Lighting → False,
 ShadowMesh → False
]
```



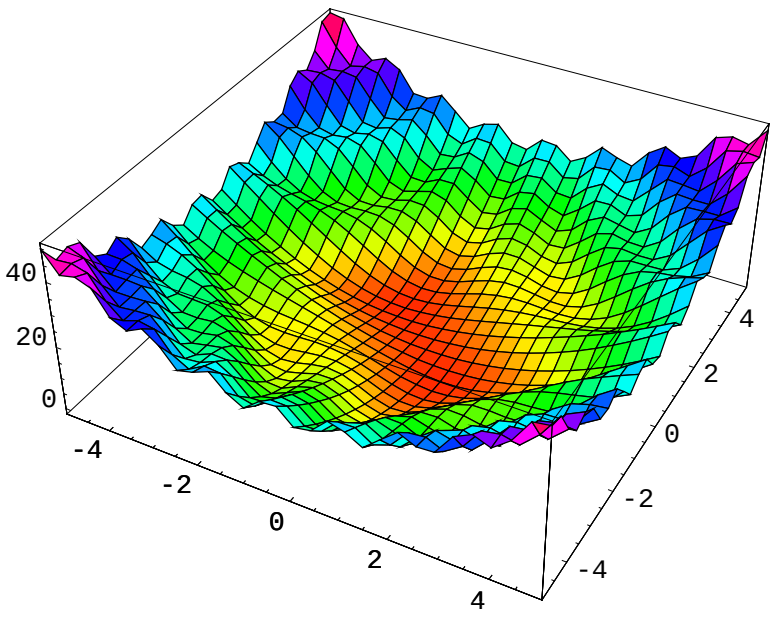
Out[182]= - Graphics3D -

Notate come, con le opzioni di default per il colore della mesh e delle luci, bisogna disabilitare queste ultime affinchè l'ombra sia effettivamente disegnata: in caso contrario, avremmo non il colore della funzione, ma del piano con la luce, quindi un piano uniforme. Notate come varino alcune opzioni di default: per esempio, in Plot3D di default il colore della mesh è uniforme, mentre in questo caso è dato dalla funzione Hue. Inoltre ci sono parametri distinti per personalizzare sia la mesh che la sua proiezione: per esempio, in Plot3D ho l'opzione Mesh, mentre qua ne ho due, SurfaceMesh e ShadowMesh, per personalizzarle in maniera indipendente. Le opzioni specifiche sono:

option name	*default value*	
`ColorFunction`	`Hue`	funzione per determinare il colore della superfice in base all'altezza
`SurfaceMesh`	`True`	specifia se disegnare le linee della mesh della superfice
`SurfaceMeshStyle`	`RGBColor[0, 0, 0]`	direttive grafiche per disegnare la mesh della superfice
`ShadowMesh`	`True`	specifica se disegnare la mesg della proiezione
`ShadowMeshStyle`	`RGBColor[0, 0, 0]`	direttive grafiche per disegnare la mesh della proiezione
`ShadowPosition`	`-1`	specifica se la mesh debba essere disegnata nella superfice inferiore o superiore

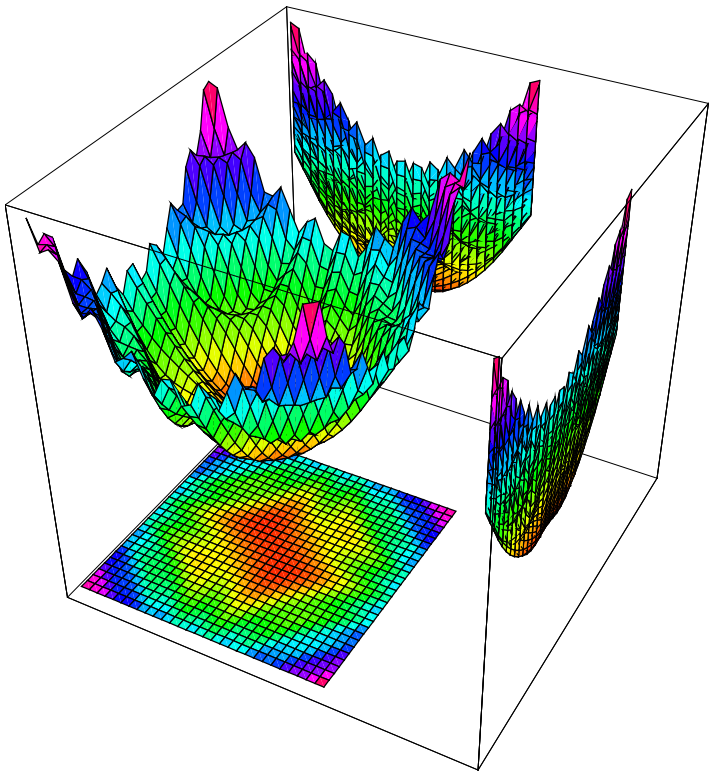
Se vogliamo proiettarlo in altri piani, dobbiaom utilizzare il comando Shadow, il che significa che prima dobbiamo creare il grafico e poi utilizzarlo con Shadow:

In[183]:=
```
grafico = Plot3D[x^2 + y^2 + 3 Sin[x y], {x, -5, 5}, {y, -5, 5},
  PlotPoints → 30,
  Lighting → False,
  ColorFunction → Hue
 ]
```



Out[183]= `- SurfaceGraphics -`

In[184]:=
```
Shadow[grafico, BoxRatios → {1, 1, 1},
 Lighting → False, ShadowMesh → False,
 XShadowPosition → 1]
```



Out[184]= `- Graphics3D -`

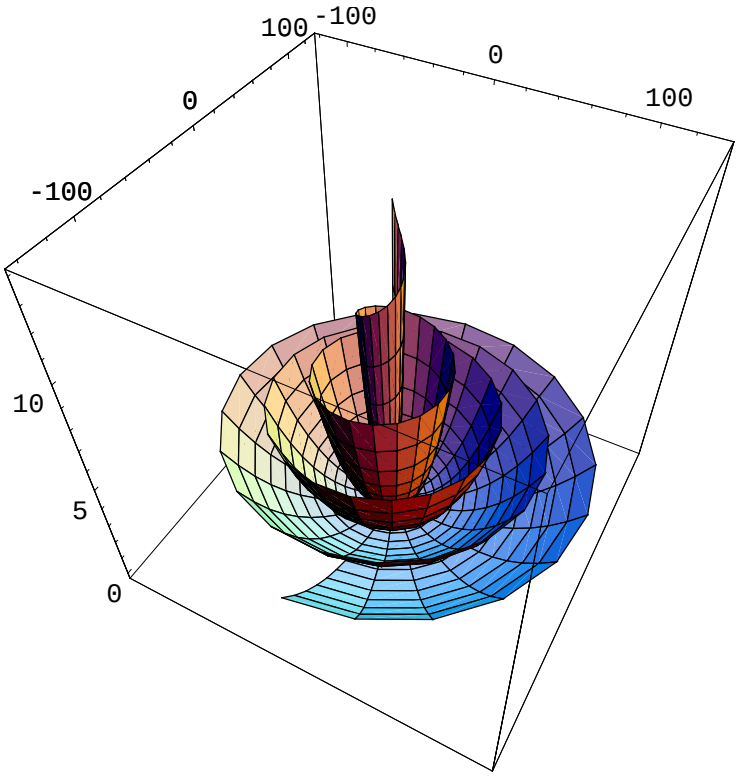
Come potete vedere qua già ci avviciniamo alla qualità dei grafici pubblicati nelle riviste (senza bisogno di andare ad usare programmi come Origin), e ciò sarà ancora più vero quando, più avanti, spiegherò il package per fare le legende.

Se non ci piacciono le direzioni standard di proiezione, perchè magari ci interessano piani inclinati (chi studia cristallografia mi capisce bene...), allora può essere utile quest'altro comando, che esegue proiezioni in qualsiasi piano dello spazio:

`Project[`*g*`, `*pt*`]`	proietta l'oggetto tridimensionale *g* nel piano, la cui normale è rappresentata da un vettore che unisce il centro dell'oggetto con il punto *pt*
`Project[`*g*`, {`e_1`, `e_2`}, `*pt*`]`	proietta l'oggetto *g* nel piano che include i vettori vectors $\{e_1, e_2\}$ basati su *pt*
`Project[`*g*`, {`e_1`, `e_2`}, `*pt*`, `*origin*`]`	proietta nella direzione determinata dal vettore *pt* e *origin*

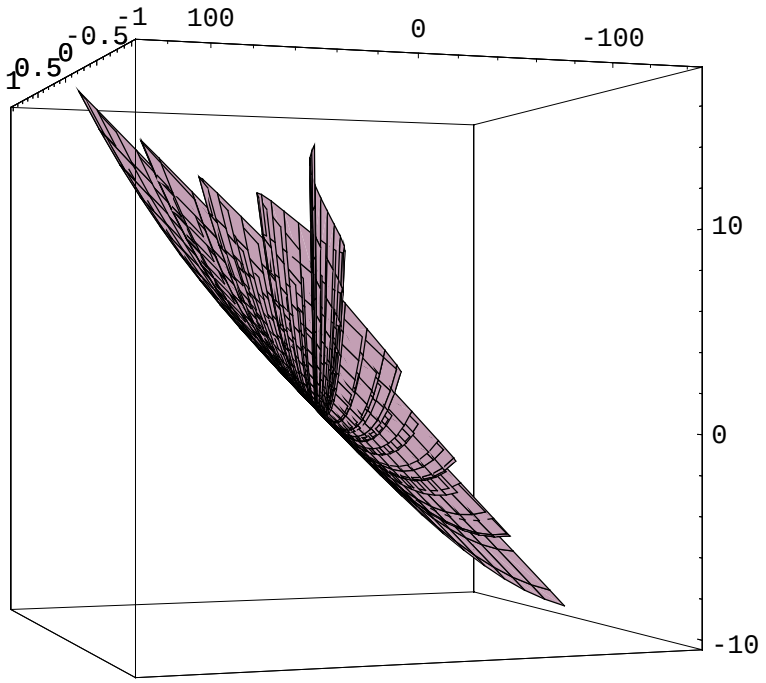
Consideriamo il seguente oggetto grafico tridimensionale:

In[185]:=
```
spirale = ParametricPlot3D[
  {t u Sin[u], t u Cos[u], t^3 / (u + 9)}, {t, 0, 5}, {u, 0, 9 Pi},
  PlotPoints → {13, 90},
  PlotRange → All,
  ViewPoint -> {0.675, -1.246, 1.5},
  BoxRatios → {1, 1, 1}
 ]
```



Out[185]= - Graphics3D -

In[186]:=
```
Show[Project[spirale, {{0, .1, 1}, {0, 1, 0}}, {0, 1, 1}, {0, 0, 0}],
 ViewPoint -> {-3.878, 1.362, 0.000}]
```



Out[186]=
```
- Graphics3D -
```

Un altro comando che può essere utile è il seguente:

`StackGraphics[{`g_1`,` g_2`, … }]`	restituisce un'immagine tridimensionale a partire dagli oggetti tridimensionali g_1,g_2,…

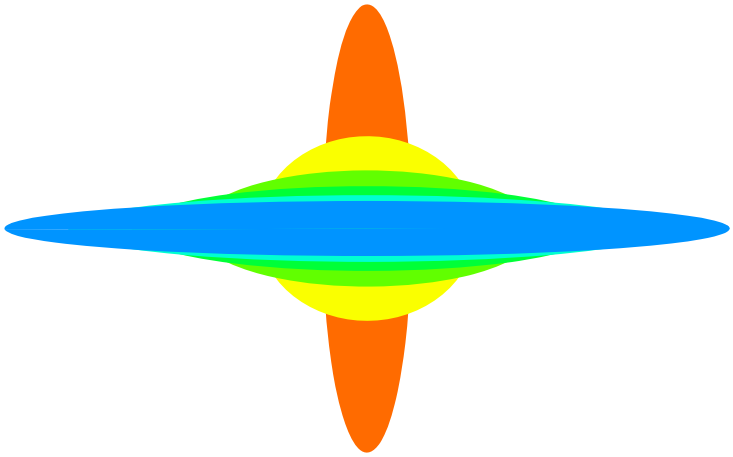
Può essere una maniera diversa di rappresentare un grafico bidimensionale al variare di un parametro; a volte è meno chiaro di una tabella, ma di certo fa più effetto. A parte il fatto che lo potete usare pure se vi serve per creare un particolare grafico.

Supponiamo di avere le seguenti figure:

In[187]:=
```
figure = Table[
  Graphics[
   {Hue[n / 10], Disk[{0, 0}, {n, 1 / n}]}
  ], {n, .7, 6}
 ]
```

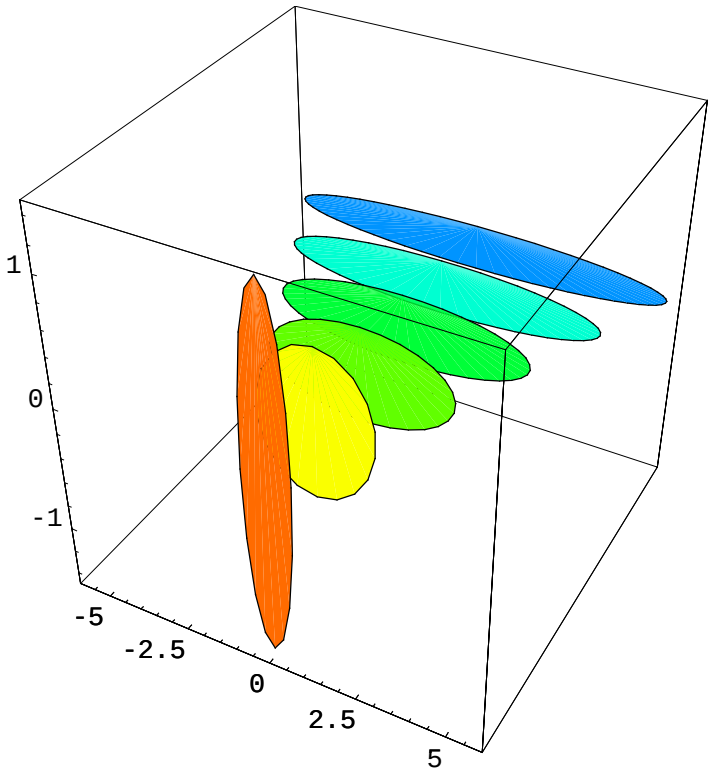
Out[187]=
```
{- Graphics -, - Graphics -, - Graphics -,
 - Graphics -, - Graphics -, - Graphics -}
```

In[188]:=
```
Show[figure]
```



Out[188]=
```
- Graphics -
```

In[189]:=
```
Show[StackGraphics[figure, Lighting → False], PlotRange → All]
```



Out[189]=
```
- Graphics3D -
```

Come potete vedere, quello che fa questo comando è mettere le figure bidimensionali in fila fra di loro in un grafico tridimensionale. Notate come questo comando crea solo il l'oggetto grafico, e come dobbiate utilizzare Show per poterlo anche mostrare.

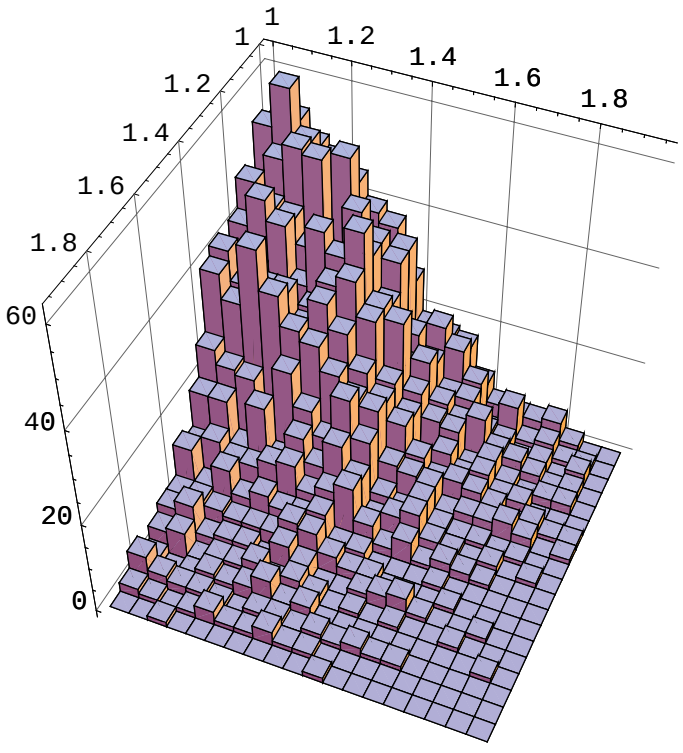
L'ultimo comando che andiamo a vedere per questo package, è quello riguardante il disegno di istogrammi tridimensionali:

`Histogram3D[{{`x_1`,` y_1`}, {`x_2`,` y_2`}, … }]`	genera un istogramma dei dati grezzi
`Histogram3D[{{{`f_{11}`,` f_{12}`,` f_{13}`, … }, {`f_{21}`,` f_{22}`,` f_{23}`}, … }, … }, FrequencyData -> True]`	genera un diagramma delle frequenze relative, dove gli estremi degli intervalli per l'asse x sono dati da c_0,c_1,c_2,..., mentre per l'asse y sono dati da d_0,d_1,d_2,..., e quindi $f_i\ j$ rappresenta il numero di dati nell'intervallo bidimensionale $c_{j-1}<=x<c_j$ e $d_{j-1}<=y<d_j$

Creiamo un esempio di dati bidimensionali con distribuzione Gaussiana:

In[190]:=
```
Table[
  Exp[(Random[]^2 + Random[]^2) / 3], {n, 4000}, {m, 2}];
```

In[191]:=
```
Histogram3D[%,
 ViewPoint -> {2.667, 1.154, 2.906},
 FaceGrids → {{0, -1, 0}, {-1, 0, 0}},
 Boxed → False
]
```



Out[191]= `- Graphics3D -`

Come potete vedere, in questo caso ho la rappresentazione dei dati "sperimentali", grezzi. Si ragiona in maniera analoga al caso bidimensionale, se abbiamo direttamente i dati relativi alle frequenze

relative, specificando il numero di elementi per ogni intervallo, dai dati sperimentali, e anche gli intervalli:

option name	*default value*	
`ApproximateIntervals`	`Automatic`	specifica se gli intervalli debbano essere approssimati con numeri semplici; può assumere valori `True`, `False`, or `Automatic`
`FrequencyData`	`False`	specifica se i dati debbano essere considerati dati grezzi, opure se rappresentano direttemente le frequenze relative, nel qual caso si devono specificare gli intervalli in `HistogramCategories`
`HistogramCategories`	`Automatic`	specifica le categorie (intervalli) del diagramma
`HistogramRange`	`Automatic`	specifica il range dei dati che devono essere inclusi nell' istogramma
`HistogramScale`	`Automatic`	specifica se le altezze delle barre debbano essere scalate in modo che le misure dell'altezza delle densità delle frequenze relative o il volume delle barre debba avere una somma unitaria

Anche qua possiamo sia utilizzare opzioni standard per gli oggetti tridimensionali, sia le opzioni per i marker degli intervalli, che in questo caso sono i seguenti:

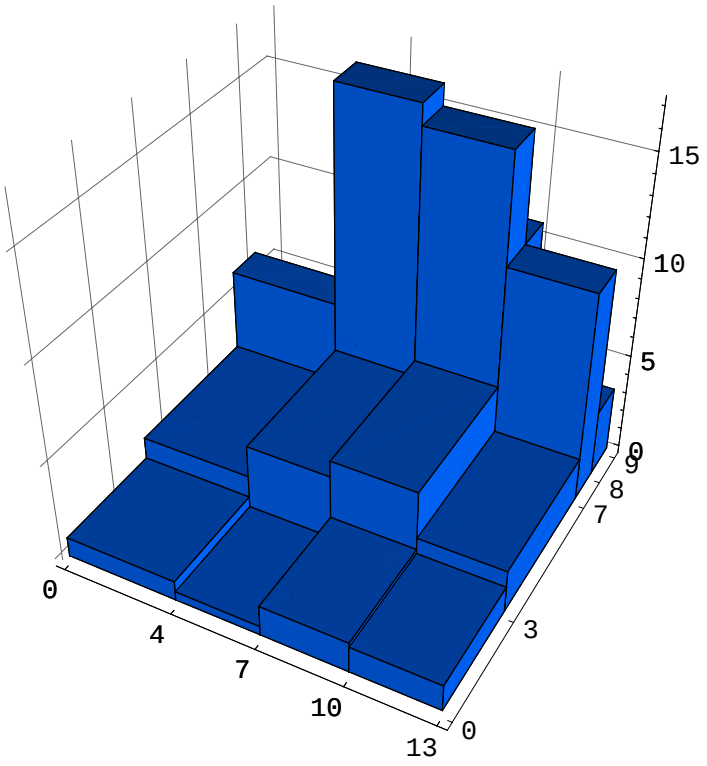
`Ticks -> None`	non disegna i marker
`Ticks -> Automatic`	disegna automaticamente i marker
`Ticks -> IntervalBoundaries`	posizione i marker ai limiti degli intervalli
`Ticks -> IntervalCenters`	posiziona i marker al centro degli intervalli
`Ticks -> {`*xticks*`, `*yticks*`, `*zticks*`}`	specifica *i* marker per ogni asse

Vediamo, quindi, un esempio che racchiuda quello che sappiamo fare con questo comando:

```
In[192]:= dati = {
            {12, 34, 25, 17},
            {4, 45, 51, 33},
            {14, 55, 48, 32},
            {12, 26, 32, 10}
           };
```

In[193]:=
```
intervalli = {
   {0, 4, 7, 10, 13},
   {0, 3, 7, 8, 9}
  };
```

In[194]:=
```
Histogram3D[dati,
 FrequencyData → True,
 HistogramCategories → intervalli,
 SolidBarStyle → (Hue[.6]),
 LightSources → {{{4, -4, 7}, Hue[0.6]}},
 Boxed → False,
 AxesEdge → {{-1, -1}, {1, -1}, {1, 1}},
 Ticks → IntervalBoundaries,
 FaceGrids → {{0, 1, 0}, {-1, 0, 0}}
]
```



Out[194]= `- Graphics3D -`

Anche in questo caso, non sono riuscito a trovare il modo per colorare in modo indipendente le barre... :-(Evidentemente ho ancora molto da imparare...

■ Graphics`ImplicitPlot`

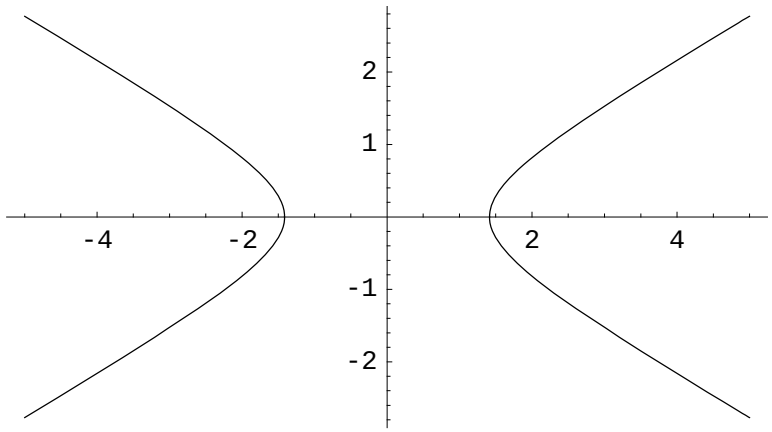
Questo package definisce un comando che permette di disegnare grafici di funzioni implicite, cioè che non possono essere scritte esplicitamente nella forma $y = f[x]$, come capita per moltissime famiglie di curve: per esempio le coniche, definendo il luogo dei punti che rappresentano le soluzioni dell'equazione:

`ImplicitPlot[`*eqn*`, {`*x*`,` *xmin*`,` *xmax*`}]`	disegna la soluzione di *eqn* usando il metodo `Solve`, con *x* che varia da *xmin* a *xmax*
`ImplicitPlot[`*eqn*`, {`*x*`,` *xmin*`,` m_1`,` m_2`, … ,` *xmax*`}]`	disegna la soluzione, evitando i punti m_i
`ImplicitPlot[`*eqn*`, {`*x*`,` *xmin*`,` *xmax*`}, {`*y*`,` *ymin*`,`	disegna il grafico utilizzando il metodo usato da `ContourPlot`
`ImplicitPlot[{`eqn_1`,` eqn_2`, … },` *ranges*`,` *options*`]`	disegna le soluzioni delle equazioni eqn_i

Notiamo come il comando può essere utilizzato in due modi: al suo interno può utilizzare sia il comando Solve, sia un ContourPlot: nel primo caso, definito il range della variabile *x*, trova le soluzioni dell'equazione per vari punti di *x* utilizzando Solve, e poi lo disegna; nel secondo caso, invece, si viene a creare una rappresentazione bidimensionale, creando un grafico di contorno e disegnando solamente la curva di livello corrispondente al valore di z pari a 0. Questo metodo è iterativo e crea curve approssimate, dovendo lavorare con l'opzione PlotPoints per poter creare delle curve più morbide. Il primo metodo, invece, crea grafici migliori, ma pone i limiti di Solve, quali ad esempio quelli riguardanti funzioni periodiche ed impossibilità di risolvere determinati tipi di equazioni. Vediamo adesso entrambi i casi:

In[195]:=
```
<< Graphics`ImplicitPlot`
```

In[196]:=
```
ImplicitPlot[x^2 - 3 y^2 == 2, {x, -5, 5}]
```

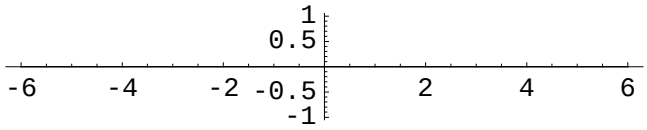


Out[196]=
```
- Graphics -
```

Vediamo che questo è un grafico che non può essere realizzato con Plot, a meno di scindere l'equazione, disegnare i singoli pezzi ed unirli con Show o con DisplayTogether.

In questo caso Solve funziona, e restituisce il grafico corretto. Vediamo, invece, il seguente disegno:

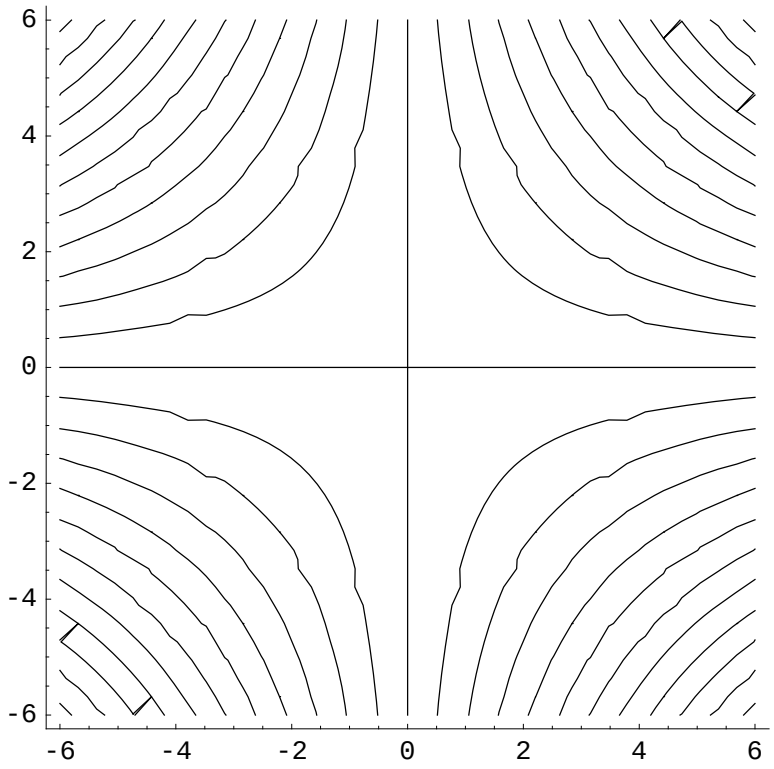
```
In[197]:= ImplicitPlot[Sin[x y] == 0, {x, -6, 6}]

— Solve::incnst : Inconsistent or redundant transcendental equation.
    After reduction, the bad equation is ArcSin[Sin[x y]] == 0. More…

— Solve::ifun :
   Inverse functions are being used by Solve, so some solutions may not
    be found; use Reduce for complete solution information. More…

— Solve::svars : Equations may not give solutions for all "solve" variables. More…

— Solve::ifun :
   Inverse functions are being used by Solve, so some solutions may not
    be found; use Reduce for complete solution information. More…

— Solve::ifun :
   Inverse functions are being used by Solve, so some solutions may not
    be found; use Reduce for complete solution information. More…

— General::stop :
   Further output of Solve::ifun will be suppressed during this calculation. More…
```




```
Out[197]= - Graphics -
```

Come possiamo vedere, il comando Solve genera degli errori, e avverte che esistono possibilità che tutte le soluzioni non vengano trovate, restituendo un risultato sbagliato. Se, invece, specifichiamo pure i limiti per l'asse *y*, *Mathematica* cambia metodo di disegno, utilizzando quello di ContourPlot:

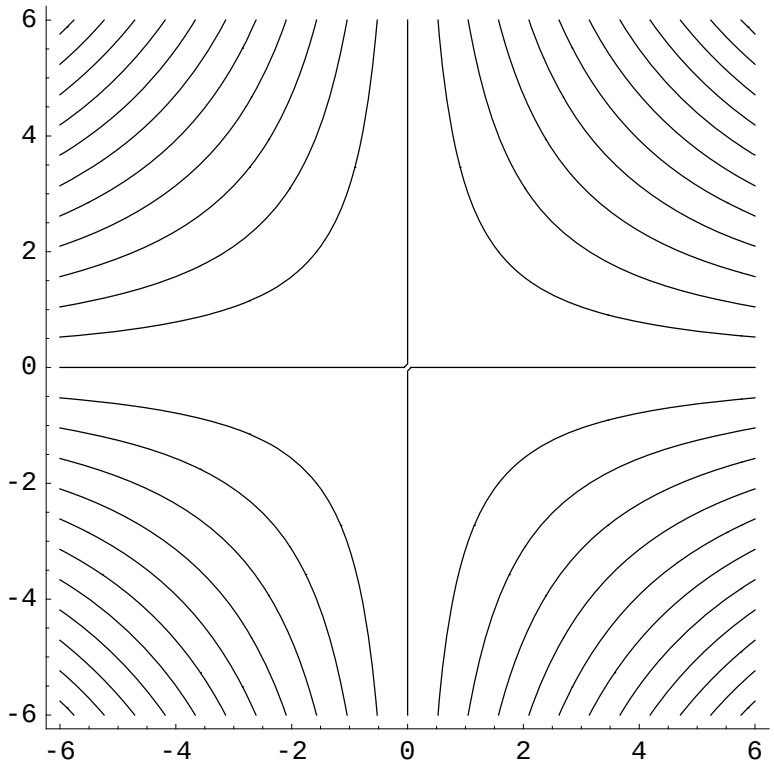
In[198]:=
```
ImplicitPlot[Sin[x y] == 0, {x, -6, 6}, {y, -6, 6}]
```



Out[198]=
```
- ContourGraphics -
```

Come possiamo vedere, in questo caso viene riportato il giusto grafico implicito. Notiamo anche come le curve siano meno definite rispetto al primo caso: per ovviare a questo nella maggior parte dei casi, basta aumentare il numero di punti:

In[199]:=
```
ImplicitPlot[Sin[x y] == 0, {x, -6, 6}, {y, -6, 6}, PlotPoints → 100]
```



Out[199]=
```
- ContourGraphics -
```

Già le cose vanno meglio...

In generale, trovo questo package abbastanza utile, permettendovi di disegnare questi grafici, che almeno io ho usato parecchio nei miei studi... Provatelo, e non lo abbandonerete facilmente...

■ Graphics`InequalityGraphics`

Il package contiene dei comandi per visualizzare, in maniera grafica, delle diseguaglianze, sia che si tratti di numeri reali, che di numeri complessi. Per i numeri reali abbiamo i seguenti comandi:

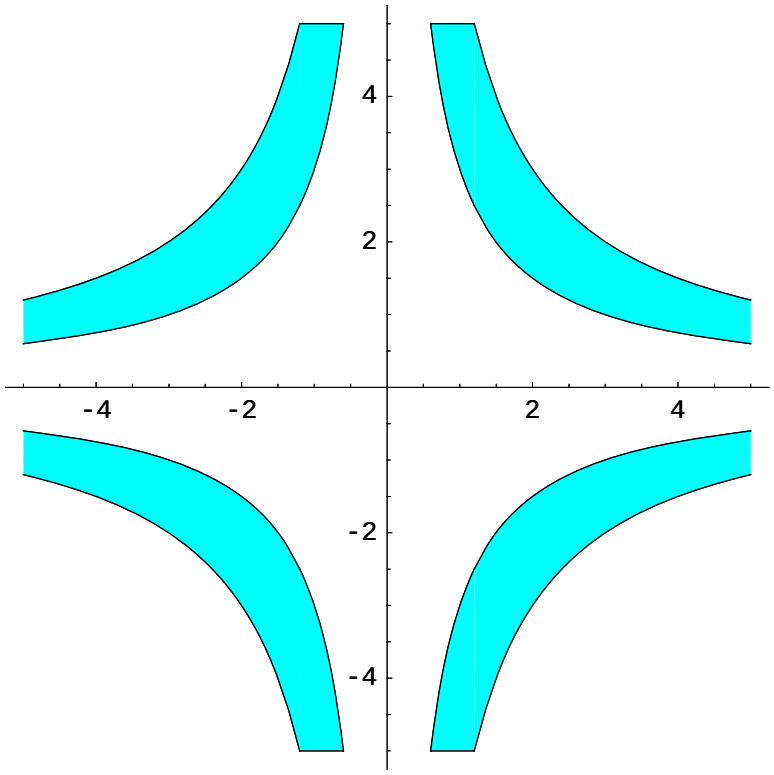
`InequalityPlot[` *ineqs*`, {`*x*`,` *xmin*`,` *xmax*`}, {`*y*`,` *ymin*`,` *ymax*`}]`	rappresenta graficamente il set di diseguaglianze *ineqs* in 2D
`InequalityPlot3D[` *ineqs*`, {`*x*`,` *xmin*`,` *xmax*`}, {`*y*`,` *ymin*`,` *ymax*`}, {`*z*`,`	rappresenta graficamente il set di diseguaglianze *ineqs* in 3D

Nel primo caso si lavora con due variabili, mentre nel secondo caso con tre. Il funzionamento è abbastanza semplice:

In[200]:=
```
<< Graphics`InequalityGraphics`
```

In[201]:=
```
diseguaglianza = 3 < Abs[x y] < 6;
```

In[202]:=
```
InequalityPlot[diseguaglianza, {x, -5, 5}, {y, -5, 5}]
```



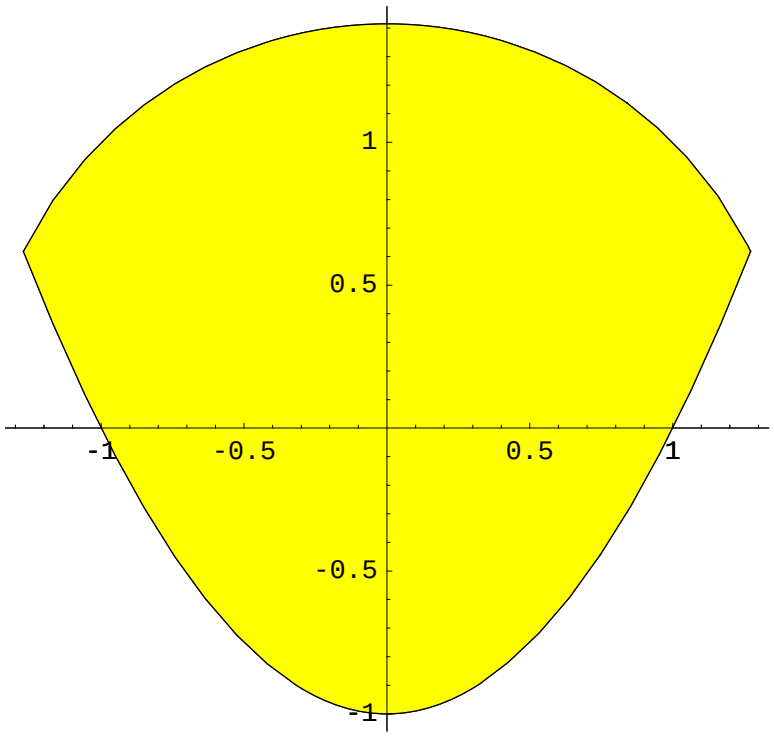
Out[202]=
```
- Graphics -
```

Come potete vedere, viene colorata automaticamente la regione che soddisfa la diseguaglianza.

Nel caso in cui la regione della diseguaglianza è limitata nel piano, cioè sia confinata entro un intervallo finito, *Mathematica* è in grado di disegnarlo interamente, senza la necessità di dover applicare i limiti per le variabili:

In[203]:=
```
diseguaglianza2 = (x^2 - y < 1) ∧ (x^2 + y^2 < 2);
```

In[204]:=
```
InequalityPlot[diseguaglianza2, {x}, {y},
 Fills → Yellow
]
```



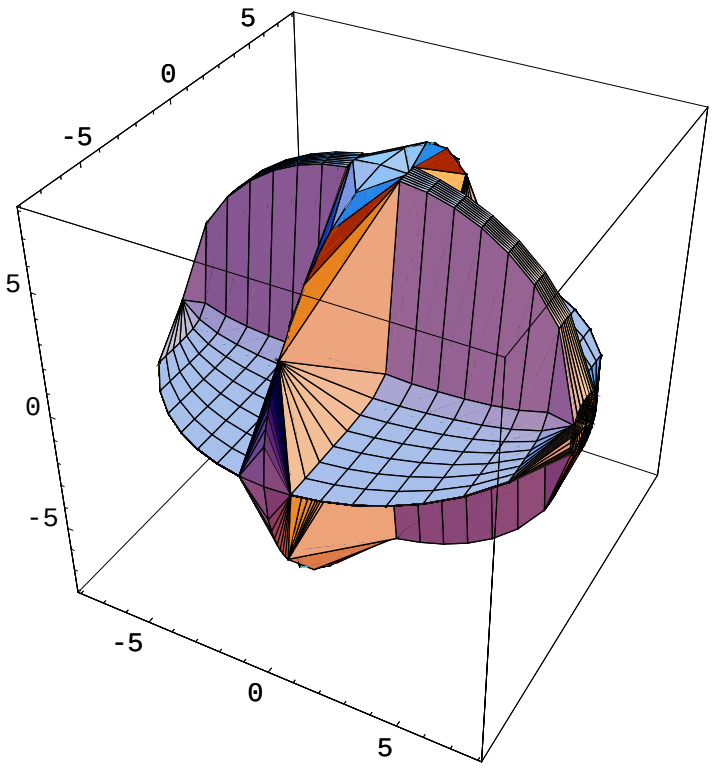
Out[204]= - Graphics -

Come potete vedere, essendo in questo caso la regione limitata, non è stato necessario definire esplicitamente i limiti, anche se ovviamente potevamo farlo, magari perchè ci interessava solamente una regione del piano. Notate anche come sia possibile introdurre disequazioni con la notazione standard matematica: per esempio, ∧ si può scrivere come Esc-and-Esc. Andate a rivedervi la parte riguardante le abbreviazioni, perchè sono molto utili per scrivere notazioni in matematica standard.

Oltre ai casi bidimensionali, possiamo creare anche grafici tridimensionali rappresentani diseguaglianze:

In[205]:=
```
dis3D = Abs[x y^5 z^2] < 2 ∧ x^2 + y^2 + z^2 < 60;
```

In[206]:=
```
InequalityPlot3D[dis3D, {x}, {y}, {z}, PlotRange → All]
```

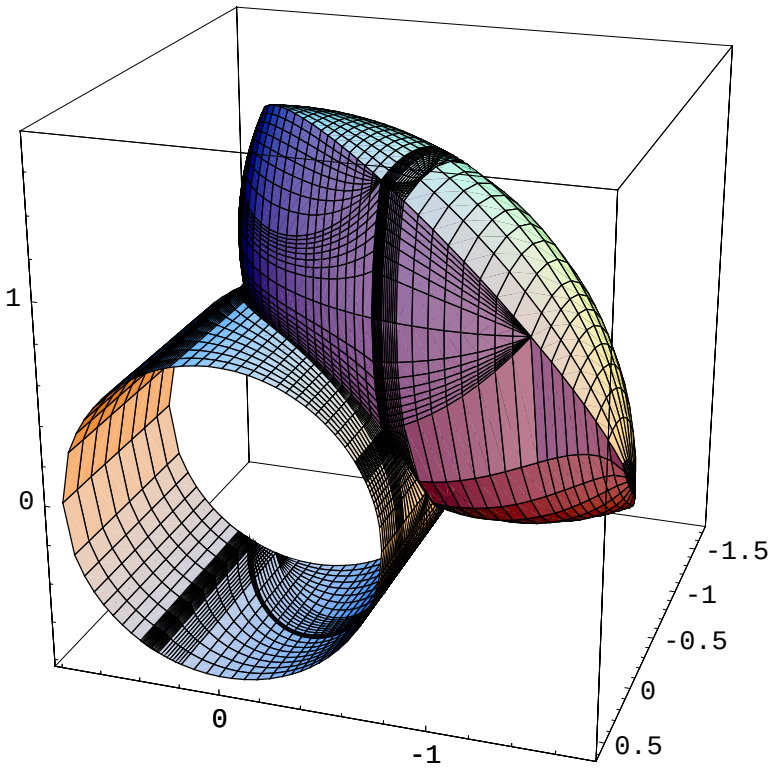


Out[206]=
```
- Graphics3D -
```

Tuttavia, ho scoperto alcuni problemucci, quando si tratta di gestire piani perfettamente paralleli ad uno dei piani verticali. Nell'esempio seguente, infatti, non riesco a disegnare le basi del cilindro, che rimangono aperte:

In[207]:=
```
dis3D2 = x^2 + y^2 + z^2 ≤ 3 ∧ (x + 1)^2 + (y + 1)^2 + (z - 1)^2 ≤ 2 ∨
    x^2 + z^2 ≤ .6 ∧ Abs[y] ≤ .6;
```

In[208]:=
```
InequalityPlot3D[dis3D2, {x}, {y},
 {z}, ViewPoint -> {-1.207, 3.619, 1.528}]
```



Out[208]= `- Graphics3D -`

Come potete vedere ci sono aluni problemi nel gestire porzioni di regioni perfettamente parallere ai piani verticali, e non sono riuscito a trovare un modo per evitarlo, neanche andando a spulciare i metodi nascosti andando ad aprire il file del package... Mi sa che è un difetto intrinseco del comando. Comunque, per tutto il resto funziona alla grande.

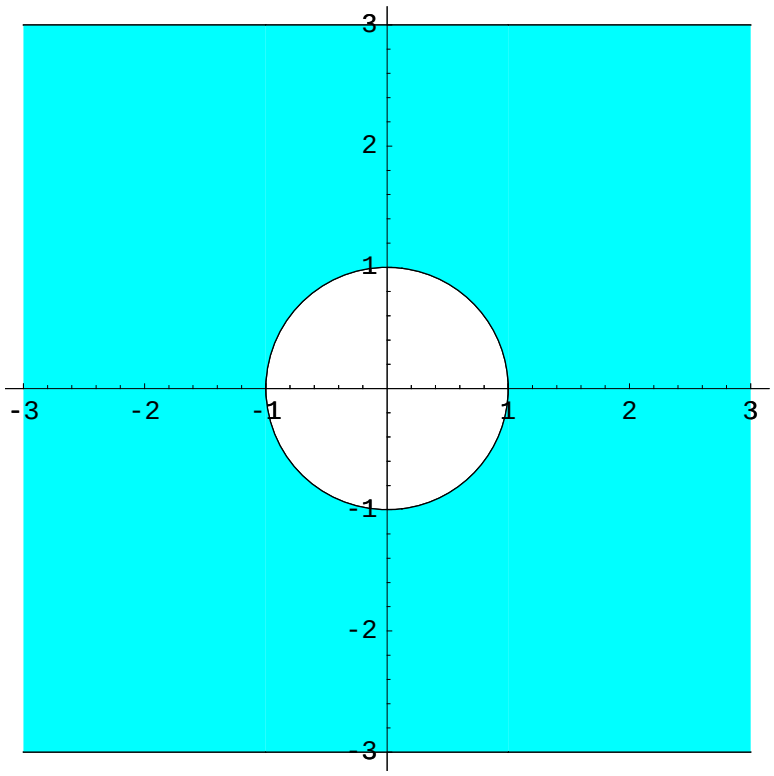
Comunque, andiamo avanti. Probabilmente esiste un metodo per risolvere questo problema ed io sono troppo stupido per trovarlo...

Dicevamo, che oltre che con numeri reali, il package ci permette di definire regioni di numeri complessi:

`ComplexInequalityPlot[` *ineqs*`, {`*z*`,` *zmin*`,` *zmax*`}]`	genera una rappresentazione grafica del set di disequazioni *ineqs* nella variabile complessa *z* nella regione di piano complesso definita da *zmin* and *zmax*.

Anche in questo caso, come nel precedente, possiamo evitare di usare i limiti del piano, se la regione che soddisfa la disequazione è limitata:

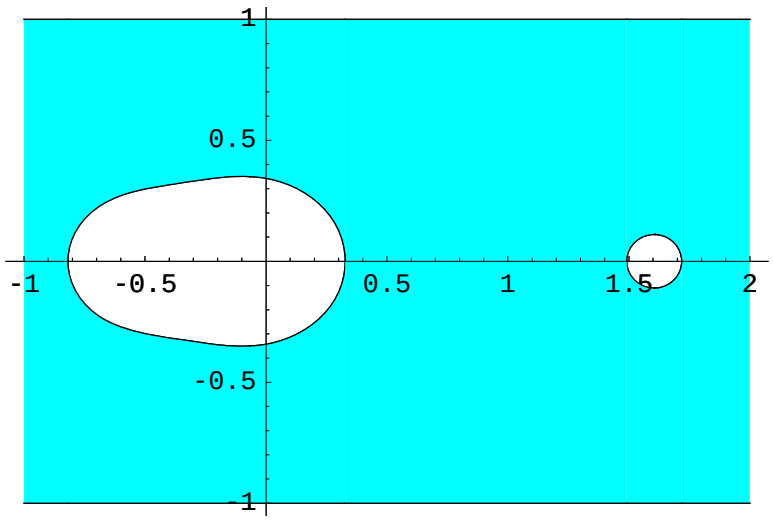
In[209]:=
```
ComplexInequalityPlot[
 -3 < Re[z] < 3 && -3 < Im[z] < 3 && Abs[z] > 1, {z}
]
```



Out[209]= - Graphics -

Ohhh, che bel buchino!!! Il procedimento di tracciamento del grafico è analogo a quello del primo comando, ma la possibilità di gestire parametri complessi permette di gestire con maggior facilità questi numeri, invece di dover esplicitare sempre parte reale e parte immaginaria:

In[210]:=
```
ComplexInequalityPlot[Abs[z^3 - z^2 - z] > .4, {z, -1 - 1 I, 2 + 1 I}]
```



Out[210]=
```
- Graphics -
```

■ Graphics`Legend`

Questo package permette di creare delle legende da usare per i grafici. Sebbene non sia importante per quanto riguarda lo studio vero e proprio (si presume che sappiate cosa state disegnando...), tuttavia è molto utile quando si decide di esportare i grafici, magari per utilizzarli in un articolo oppure se dovete stamparli in una slide... Insomma, in tutti quei casi dove una maggiore leggibilità è d'aiuto.

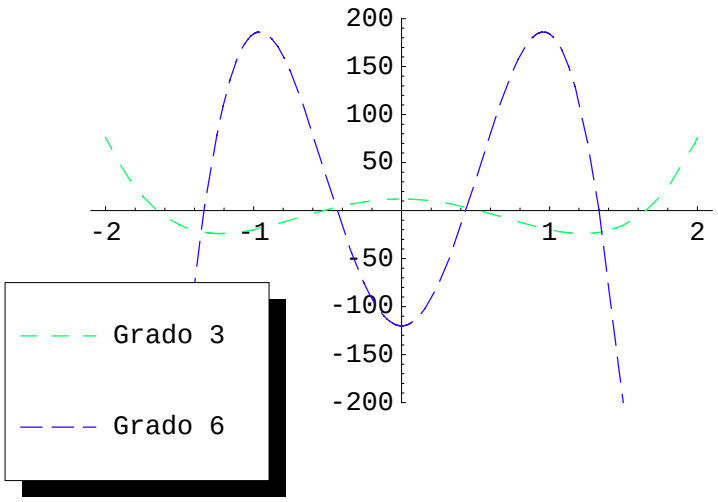
Il package aggiunge un opzione per il comando Plot, che ci permette di creare legende direttamente da esso, ed un comando che ci permette di prendere un oggetto grafico ed andare a sovrapporre la legenda:

`PlotLegend -> {`*text*$_1$`,` *text*$_2$`, … }`	opzione per `Plot` per inserire una legenda delle curve
`ShowLegend[`*graphic*`,` *legend*$_1$`,` *legend*$_2$`, … ]`	posizione le legende *legend*$_i$ sopra l'oggetto grafico
`{{{`*box*$_1$`,` *text*$_1$`}, … },` *opts*`}`	specifica una legenda, con colori od oggetti grafici in genere per *box*$_i$ ed espressioni sfruttabili per il posizionamenti della primitiva `Text` in *text*$_i$
`{`*colorfunction*`,` *n*`,` *minstring*`,` *maxstring*`,` *opts*`}`	specifica una legenda con *n* riquadri, ognuno colorato con `colorfunction`; anche con stringhe opzionali per la fine dei riquadri

Vediamo un esempio semplice, che permette di creare una legenda veloce con l'opzione per il comando Plot. Notate come bisogna specificare con PlotStyle lo stule delle varie curve, perchè PlotLegend crea solamente le etichette: se non specifichiamo gli stili la legenda sarà leggermente inutile

In[211]:=
```
<< Graphics`Legend`
```

In[212]:=
```
Plot[
 {HermiteH[4, x], HermiteH[6, x]}, {x, -2, 2},
 PlotRange → {Automatic, {-200, 200}},
 PlotStyle →
  {{Hue[0.4], Dashing[{.03}]}, {Hue[0.7], Dashing[{.04, .02}]}},
 PlotLegend → {"Grado 3", "Grado 6"}
]
```



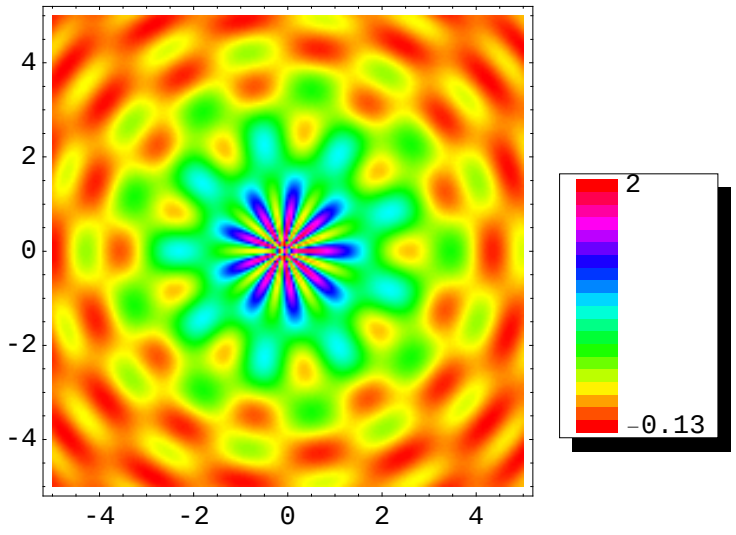
Out[212]= - Graphics -

Naturalmente, ci sono anche delle opzioni che permettono di personalizzare l'aspetto della legenda all'interno del grafico:

option name	*default value*	
`LegendPosition`	`{-1, -1}`	specifica la posizione della legenda in relazione all'oggetto grafico, dove il centro del grafico corrisponde a `{0, 0}`
`LegendSize`	`Automatic`	determina la lunghezza, oppure le dimensioni in $\{x, y\}$ nelle stesse coordinate di sistema dell'opzione `LegendPosition`
`LegendShadow`	`Automatic`	`None` non crea nessun'ombra per il riquadro, mentre $\{x, y\}$ restituisce l'offset per l'ombra della
`LegendOrientation`	`Vertical`	`Horizontal` o `Vertical`, determina l'orientamento della legenda
`LegendLabel`	`None`	etichetta per la legenda
`LegendTextDirection`	`Automatic`	le direzione del testo viene ruotata come avviene per la primitiva grafica
`LegendTextOffset`	`Automatic`	offset del testo, come per la primitiva grafica `Text`

Se andiamo ad utilizzare altri tipi di grafico, dobbiamo utilizzare il comando ShowLegend. Vediamo l'esempio e, siccome sarà complesso, lo commenterò come si fa con ognu buon programma:

In[213]:=
```
ShowLegend[
 (*
  * Creazione del grafico di
    densità. L'opzione DisplayFunction→True,
  * è necessaria, per evitare di disegnare
   due grafici invece di uno.
   * Altrimenti dovrei prima creare il grafico
   memorizzandolo in una variabile,
  * e dopo usarla nel comando,
  anche se anche in questo caso otterrei due grafici.
  *)
 DensityPlot[
  Exp[-(r2)/9] + Cos[.5 (r2)]/Sqrt[(r2 + 1)] Cos[9 Arg[(x + .1 + I y)]] /.
   r2 → x^2 + y^2,
  {x, -5, 5}, {y, -5, 5},
  PlotPoints → 200,
  DisplayFunction → Identity,
  ColorFunction → Hue,
  Mesh → False,
  PlotRange → All
 ],
 (*
  Definizione delle opzioni della legenda
  *)
 {
  (* Funzione di colore della legenda,
   che combacia con quella della funzione *)
  Hue[1 - #] &,
  (* Numero di valori della legenda: più piccolo è il numero,
   più scalettata è la legenda *)
  20,
  (* Etichette della Legenda *)
  "2", "-0.13",
  (* Posizione della Legenda *)
  LegendPosition → {1.1, -.65},
  (* Dimensioni della Legenda *)
  LegendSize → {.6, 1},
  (* Spazio attorno la legenda *)
  LegendBorderSpace → .4
 }
]
```



Out[213]= - Graphics -

In[214]:= ,

– *Syntax::sntxi : Incomplete expression; more input is needed.* More…

– ,

Sembra lungo e complicato, ma, se levate i commenti, ed il disegno della funzione (che effettivamente ho scelto abbastanza complicata, ma che da un risultato grafico niente male...), le righe sono veramente poche, con poche battute per ogni opzione della legenda.

Inoltre, il package definisce le funzioni per poter disegnare l'ombra, oltre che il comando per restituire l'oggetto grafico che rappresenta la legenda, per utilizzarlo magari in altri contesti:

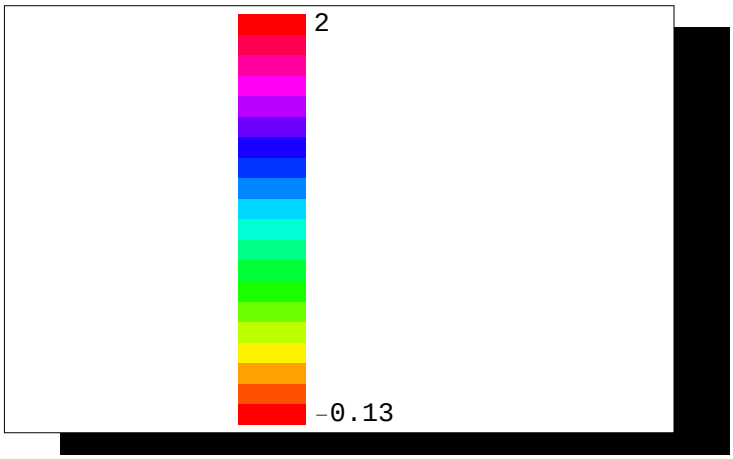
`Legend[`*legendargs*`, `*opts*`]`	restituisce la primitiva grafica che descrive la legenda
`ShadowBox[`*pos*`, `*size*`, `*opts*`]`	restituisce un rettangolo con un'ombra definita dalle dimensioni e dalle opzioni

Vediamo come viene rappresentata la legenda sotto forma di primitive grafiche:

In[214]:=
```
Legend[
 Hue[1 - #] &, 20, "2", "-0.13", LegendPosition → {1.1, -.65},
 LegendSize → {.6, 1},
 LegendBorderSpace → .4
]
```

Out[214]=
```
{GrayLevel[0], Rectangle[{1.15, -0.7}, {1.75, 0.3}],
 GrayLevel[1], Rectangle[{1.1, -0.65}, {1.7, 0.35}],
 Thickness[0.001], GrayLevel[0], Line[
  {{1.1, -0.65}, {1.7, -0.65}, {1.7, 0.35}, {1.1, 0.35}, {1.1, -0.65}}],
 Rectangle[{1.1, -0.65}, {1.7, 0.35}, - Graphics -]}
```

In[215]:=
```
Show[Graphics[%]]
```



Out[215]=
```
- Graphics -
```

Vediamo come creare un rettangolo con l'ombreggiatura:

In[216]:=
```
ShadowBox[{0, 0}, {1, 1}, ShadowBackground → GrayLevel[.7]];
```

In[217]:=
```
Show[Graphics[%]]
```

Out[217]=
```
- Graphics -
```

■ Graphics`PlotField`

Questo package permette di disegnare campivettoriali bidimensionali, che vengono rappresentati come un insieme di vettori nel piano con direzione e modulo concordi al valore del campo vettoriale in quel punto:

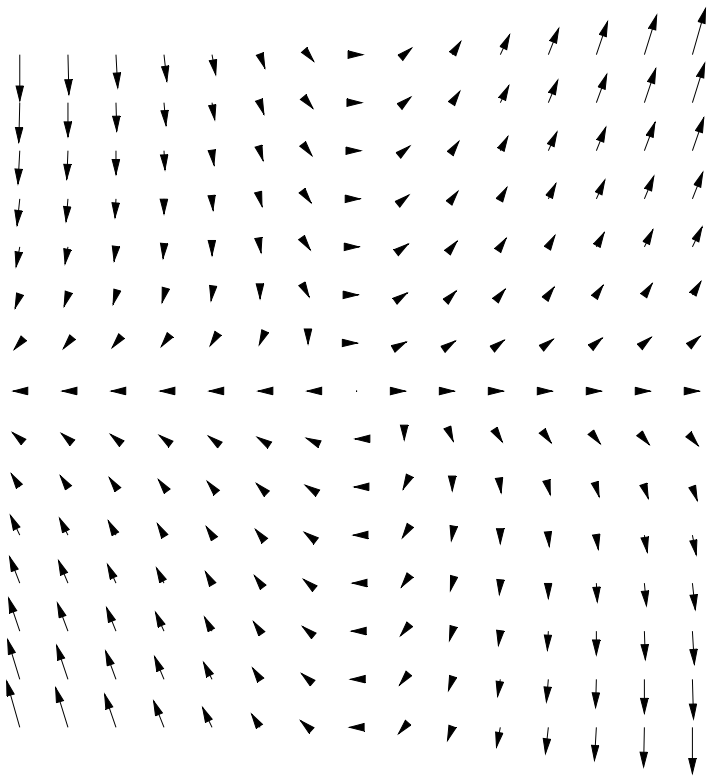
`PlotVectorField[{`f_x`,` f_y`}, {`x`,` *xmin*`,` *xmax*`}, {`y`,` *ymin*`,` *ymax*`}]`	disegna il campo vettoriale definito dalla lista delle due funzioni componenti, nel range di valori
`PlotGradientField[`f`, {`x`,` *xmin*`,` *xmax*`}, {`y`,` *ymin*`,` *ymax*`}]`	disegna il gradiente della funzione scalare f
`PlotHamiltonianField[`f`, {`x`,` *xmin*`,` *xmax*`}, {`y`,` *ymin*`,` *ymax*`}]`	disegna il campo vettoriale hamiltoniano della funzione scalare f

Vediamo subito come funziona:

In[218]:=
```
<< Graphics`PlotField`
```

In[219]:=
```
campovet = {(x + y), x y};
```

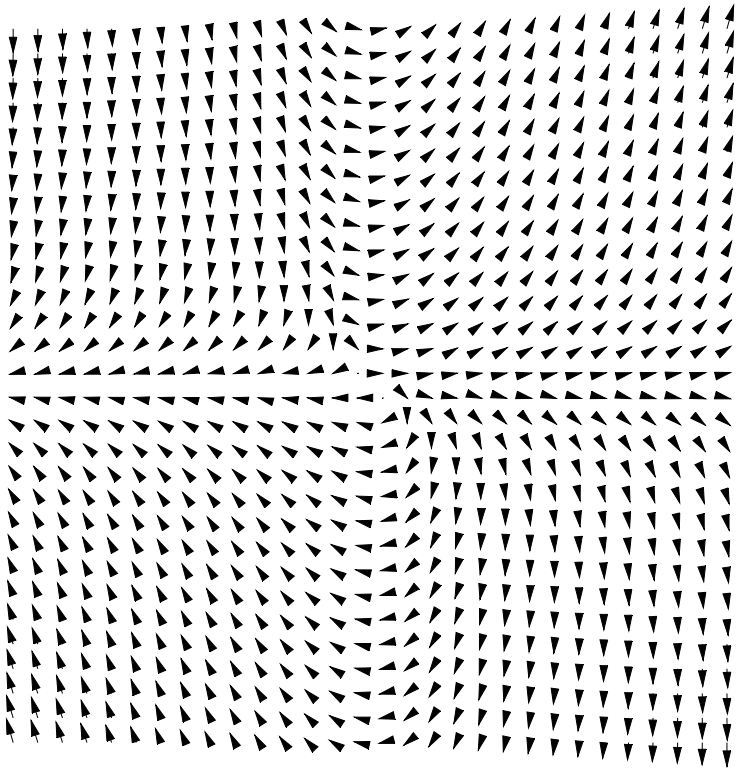
In[220]:=
```
PlotVectorField[campovet, {x, -7, 7}, {y, -7, 7}]
```



Out[220]= `- Graphics -`

Ovviamente, anche in questi casi è possibile determinare il numero di punti (quindi di frecce) che definiscono il grafico:

In[221]:=
```
PlotVectorField[campovet, {x, -7, 7}, {y, -7, 7}, PlotPoints → 30]
```



Out[221]=
```
- Graphics -
```

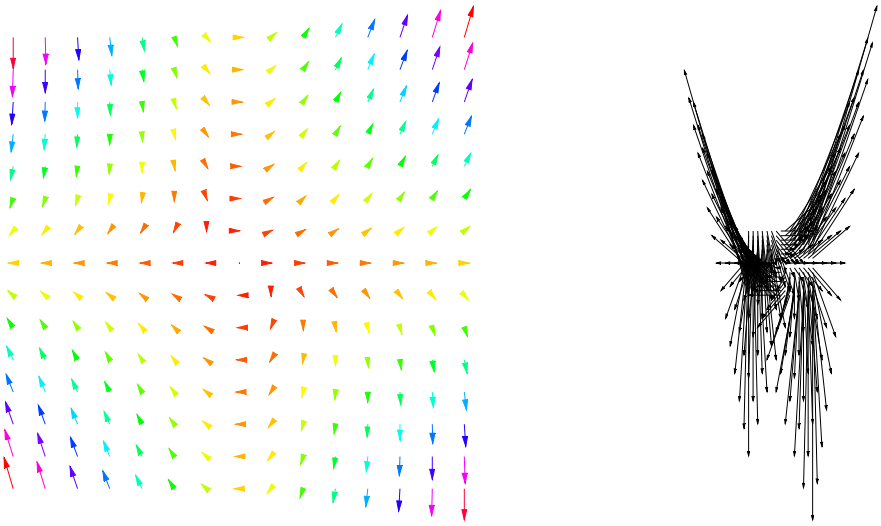
Per questi comandi sono presenti delle opzioni specifiche:

option name	*default value*	
`ScaleFactor`	`Automatic`	scala linearmente i vettori in maniera che quello più lungo abbia lunghezza determinata da questo valore; `Automatic` scala il vettore automaticamente per farlo stare nella mesh, `None` non riscala
`ScaleFunction`	`None`	la funzione utilizzata per scalare il modulo dei vettori
`MaxArrowLength`	`None`	lunghezza massima del vettore che verrà disegnato, applicato dopo `ScaleFunction` ma prima di `ScaleFactor`
`ColorFunction`	`None`	funzione utilizzata per colorare i vettori a seconda del loro modulo
`PlotPoints`	`15`	numero di punti in cui vengono calcolati i vettori in ogni direzione.

Rivediamo il primo grafico, in due diverse salse, usando due diversi set di opzioni, in modo da rappresentare in maniera diversa la stessa cosa:

In[222]:=
```
<< Graphics`Graphics`
```

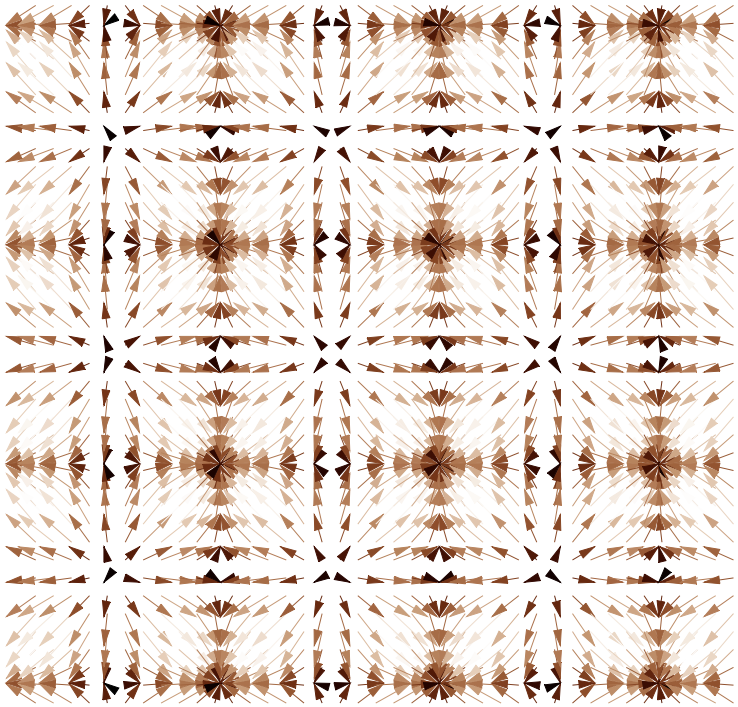
In[223]:=
```
DisplayTogetherArray[
 {{
   PlotVectorField[campovet,
    {x, -7, 7}, {y, -7, 7}, ColorFunction → (Hue[#] &)],
   PlotVectorField[campovet, {x, -7, 7},
    {y, -7, 7}, ScaleFactor → None]
  }}
]
```



Out[223]=
```
- GraphicsArray -
```

Come possiamo vedere, la rappresentazione cambia; prima avevamo scalato il tutto in modo da far apparire tutte le frecce all'interno del grafico, e sembravano tutte di modulo simile. Adesso, invece, abbiamo visto (prima con i colori, dopo con l'eliminazione della scalatura automatica), che i moduli dei vettori effettivamente cambiano molto, ed il campo è in realtà diverso da come sembrava a prima vista.

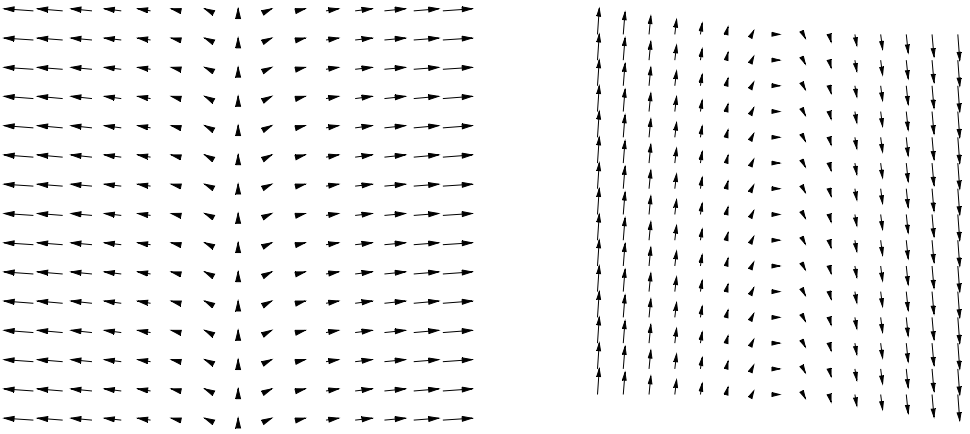
In[224]:=
```
PlotVectorField[
 {Cos[x], Sin[y]}, {x, -10, 10}, {y, -10, 10},
 PlotPoints → 40,
 ColorFunction → (RGBColor[#, #^2, #^3] &),
 ScaleFactor → None
]
```



Out[224]= `- Graphics -`

Le altre funzioni richiedono invece una funzione scalare, restituendo il gradiente oppure l'hamiltoniana:

In[225]:=
```
DisplayTogetherArray[
 {{
   PlotGradientField[x^2 + y, {x, -7, 7}, {y, -7, 7}, ScaleFactor → 1],
   PlotHamiltonianField[
    x^2 + y, {x, -7, 7}, {y, -7, 7}, ScaleFactor → 1]
  }}
]
```

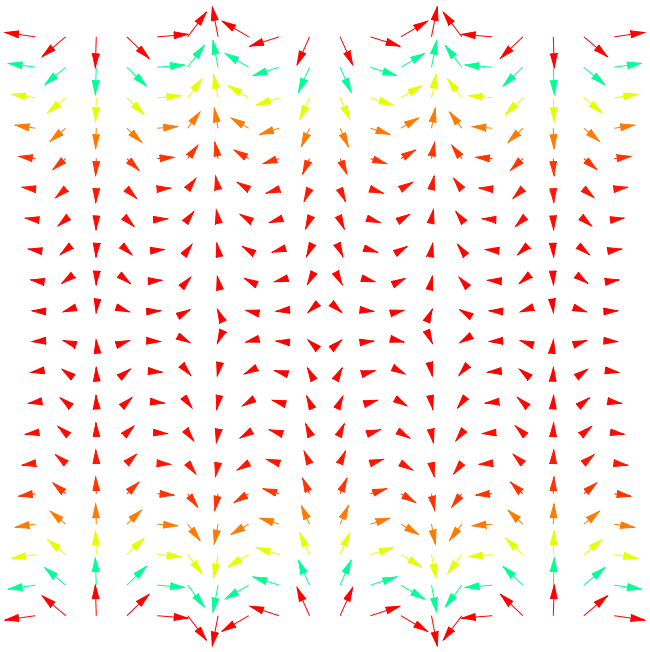


Out[225]=
```
- GraphicsArray -
```

Un altro modo di usare i campi vettoriali consiste nella rappresentazioni di funzioni complesse: infatti, ad un numero complesso ne corrisponde un altro. Allora è possibile creare un campo vettoriale dove, ad ogni punto del campo complesso, corrisponde un vettore rappresentate il corrispondente punto complesso nel codominio. Questa rappresentazione è detta di Polya:

`PlotPolyaField[`*f*`,` `{`*x*`,` *xmin*`,` *xmax*`},` `{`*y*`,` *ymin*`,` *ymax*`}]`	disegna la funzione complessa utilizzando la rappresentazione di Polya

In[226]:=
```
PlotPolyaField[Sin[x + I y], {x, -8, 8},
 {y, -8, 8}, PlotPoints → 20, ColorFunction → Hue]
```



Out[226]= `- Graphics -`

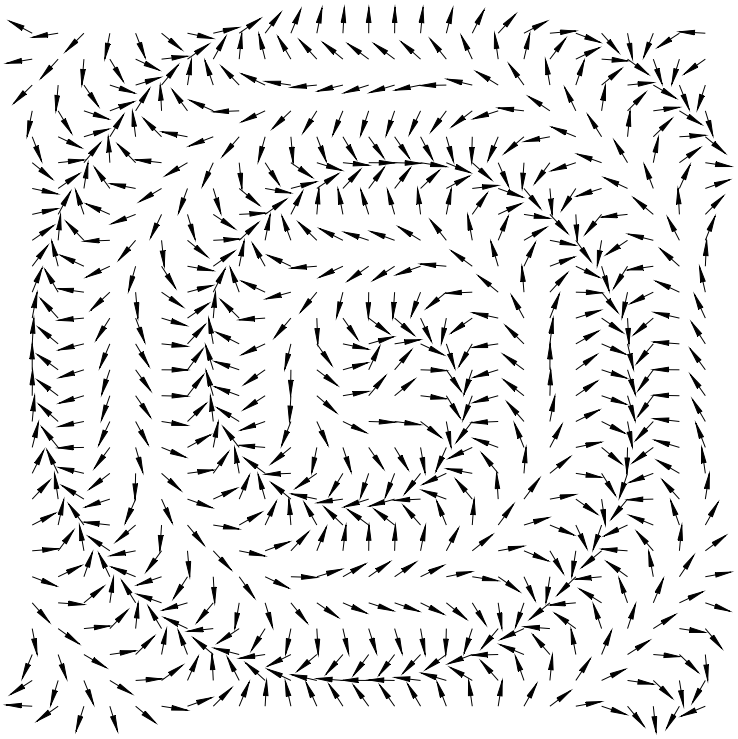
Inoltre, sono presenti delle funzioni analoghe, che lavorano con liste di dati: il comando accetta due diversi tipi di dati: nel primo caso l'elemento della lista contiene solamente un vettore, e quindi saranno rappresentati in un array bidimensionale; nel secondo caso, invece, in ogni elemento della lista è presente sia le componenti del vettore, che il punto in cui esso compare:

`ListPlotVectorField[{{`$vect_{11}$`, `$vect_{12}$`, … }, {`$vect_{21}$`, `$vect_{22}$`, … }, … }]`	crea un grafico vettoriale della matrice di vettori $vect_{xy}$
`ListPlotVectorField[{{`pt_1`, `$vect_1$`}, {`pt_2`, `$vect_2$`}, … }]`	restituisce il grafico vettoriale dei vettori dati nei determinati punti

Vediamo il primo caso:

In[227]:=
```
listavettori = Table[
   {Sin[Sqrt[n^2 + m^2]], Cos[Sqrt[n^2 + m^2]]},
   {n, -4 Pi, 4 Pi, .3 Pi}, {m, -4 Pi, 4 Pi, .3 Pi}
  ];
```

In[228]:=
```
ListPlotVectorField[listavettori, HeadWidth → 0.3]
```



Out[228]=
```
- Graphics -
```

Notate come abbia utilizzato un opzione che permette di modificare l'aspetto delle frecce:

In[229]:=
```
Options[ListPlotVectorField]
```

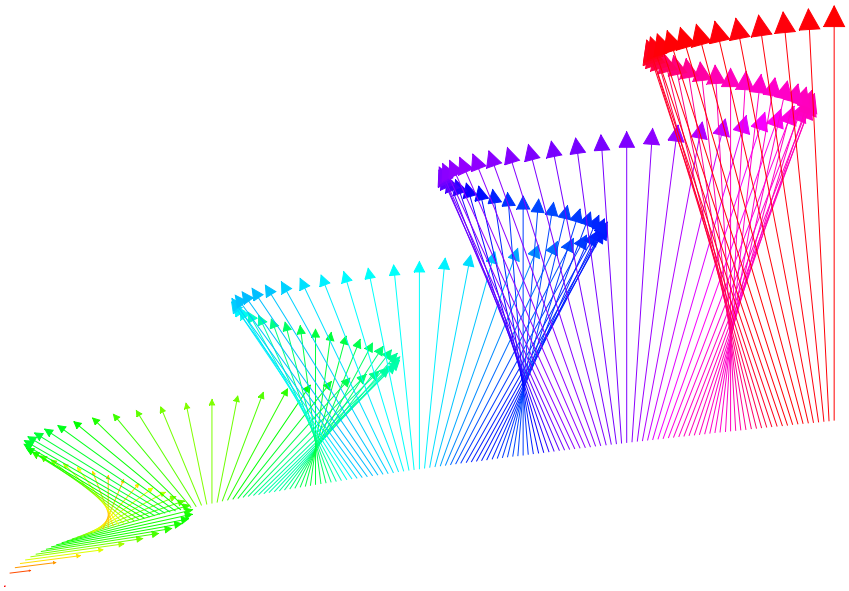
Out[229]=
```
{ScaleFactor → Automatic, ScaleFunction → None,
 MaxArrowLength → None, ColorFunction → None, AspectRatio → Automatic,
 HeadScaling → Automatic, HeadLength → 0.02, HeadCenter → 1,
 HeadWidth → 0.5, HeadShape → Automatic, ZeroShape → Automatic,
 AspectRatio → 1/GoldenRatio, Axes → False, AxesLabel → None,
 AxesOrigin → Automatic, AxesStyle → Automatic, Background → Automatic,
 ColorOutput → Automatic, DefaultColor → Automatic,
 DefaultFont :→ $DefaultFont, DisplayFunction :→ $DisplayFunction,
 Epilog → {}, FormatType :→ $FormatType, Frame → False, FrameLabel → None,
 FrameStyle → Automatic, FrameTicks → Automatic, GridLines → None,
 ImageSize → Automatic, PlotLabel → None, PlotRange → All,
 PlotRegion → Automatic, Prolog → {}, RotateLabel → True,
 TextStyle :→ $TextStyle, Ticks → Automatic, AxesFront → False}
```

Come vedete, ci sono opzioni come HeadScaling ed HeadLength. Non ho lo spazio di spiegarvi tutto, però date sempre un'occhiata alle opzioni di una funzione che utilizzate, perchè potreste trovarne qualcuna nascosta veramente utile, come in questo caso.

Vediamo, adesso, un esempio di come possiamo creare una lista con posizioni e vettori assieme, e la differenza nel visualizzarlo:

In[230]:=
```
listavettori2 = Table[
   {{n, Sqrt [n]}, {4 Sin[ n], n / 2}}, {n, 0, 8 Pi, .05 Pi}
  ];
```

In[231]:=
```
ListPlotVectorField[listavettori2,
 ColorFunction → Hue,
 HeadWidth → 1,
 HeadLength → .05,
 HeadScaling → Relative
]
```



Out[231]= `- Graphics -`

Come potete vedere, in questo caso il campo vettoriale non viene disegnato più in un array bidimensionale, ma vengono semplicemente disegnati i vettori nei punti indicati nella lista

■ Graphics`PlotField3D`

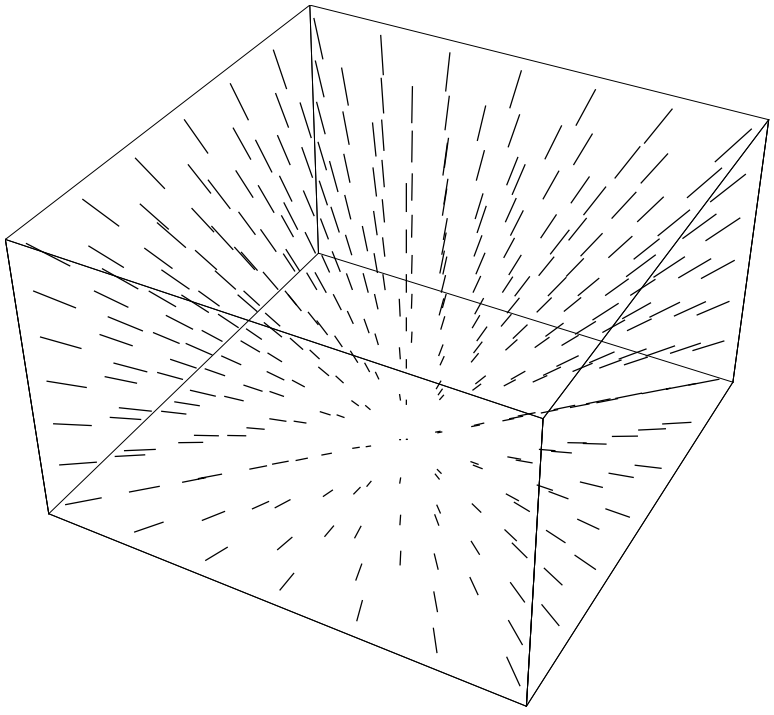
Questo package è l'analogo tridimensionale di quello precedente, permettendo di creare grafici vettoriali nelle tre dimensioni:

`PlotVectorField3D[{`f_x`,` f_y`,` f_z`}, {`*x*`,` *xmin*`,` *xmax*`}, {`*y*`,` *ymin*`,` *ymax*`}, {`*z*`,` *zmin*`,` *zmax*`}]`	disegna il campo vettoriale dao dalla funzione vettoriale, nel range specificato
`PlotGradientField3D[`*f*`, {`*x*`,` *xmin*`,` *xmax*`}, {`*y*`,` *ymin*`,` *ymax*`}, {`*z*`,` *zmin*`,` *zmax*`}]`	disegna il campo gradiente della funzione scalare *f*

I comandi lavorano in maniera quasi identica a quelli del caso bidimensionale, con l'ovvia aggiunta della terza dimensione:

In[232]:=
```
<< Graphics`PlotField3D`
```

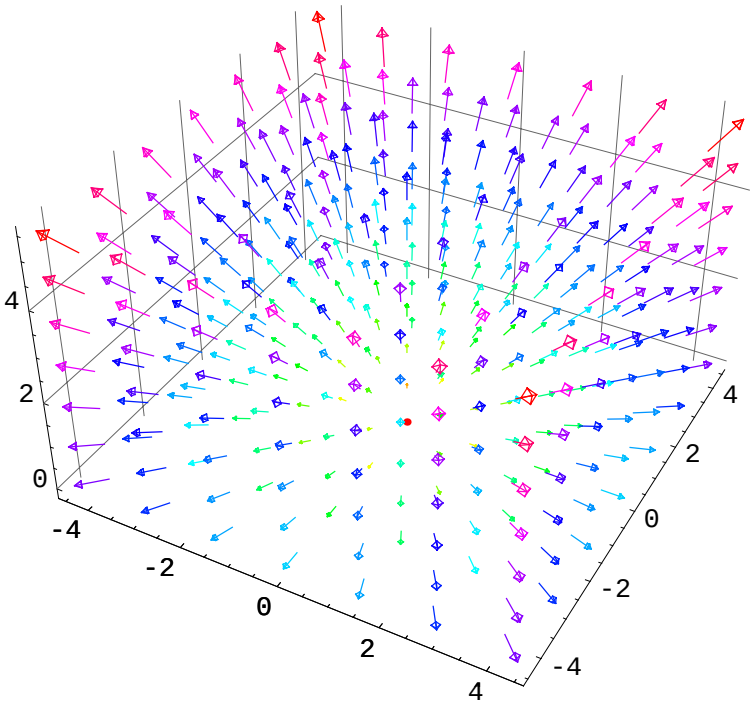
In[233]:=
```
PlotVectorField3D[{x, y, z},
 {x, -4, 4}, {y, -4, 4}, {z, 0, 5}]
```



Out[233]=
```
- Graphics3D -
```

Come potete vedere, in questo caso si disegnano soltanto delle linee, senza le teste delle frecce. Se vogliamo inserirle, dobbiamo esplicitarlo con l'opzione esatta:

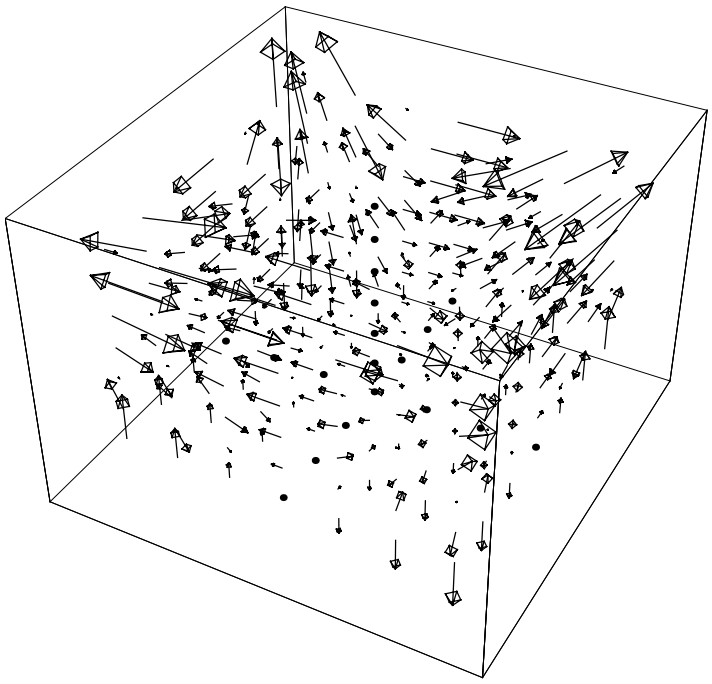
In[234]:=
```
PlotVectorField3D[{x, y, z},
 {x, -4, 4}, {y, -4, 4}, {z, 0, 5},
 VectorHeads → True,
 ColorFunction → Hue,
 Boxed → False,
 Axes → True,
 FaceGrids → {{0, 1, 0}, {-1, 0, 0}}
]
```



Out[234]= - Graphics3D -

Possiamo anche disegnare i gradienti:

In[235]:=
```
PlotGradientField3D[Sin[x y z], {x, -4, 4}, {y, -4, 4}, {z, 0, 5},
 ScaleFactor → 2, VectorHeads → True
]
```



Out[235]= - Graphics3D -

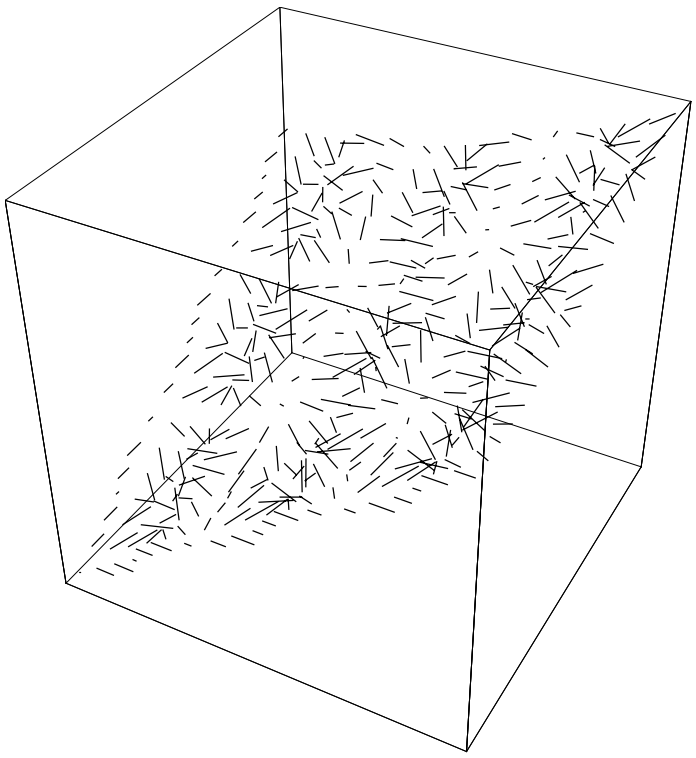
Inoltre, possiamo anche rappresentare liste di vettori:

`ListPlotVectorField3D[{{`pt_1`, `$vect_1$`}, {`pt_2`, `$vect_2$`}, … }]`	disegna una lista di vettori nei punti specificati

Vediamo l'ultimo esempio di questo package (e me ne vado a dormire, che sono le 12.30 e domani devo alzarmi presto):

In[236]:=
```
vettori3D = Flatten[
   Table[
    {
     {n, m, Sqrt[n^2 + m^2]},
     {Sin[n], Sin[m], Sin[m n]}
    },
    {n, 0, 6 Pi}, {m, 0, 6 Pi}
   ],
   1
  ];
```

In[237]:=
```
ListPlotVectorField3D[vettori3D, BoxRatios → {1, 1, 1}]
```



Out[237]=
```
- Graphics3D -
```

Ed ora, con il vostro permesso, me ne andrei a dormire, dato che il sonno avanza e non riesco più ad inventarmi niente di interessante.

Buonanotte.

■ Graphics`SurfaceOfRevolution`

Non è bello smettere perchè si ha sonno, e ricominciare il giorno dopo a mezzanotte passata... Devo gestire meglio i miei impegni, ma d'altronde è periodo d'esame...

Ma torniamo a noi.

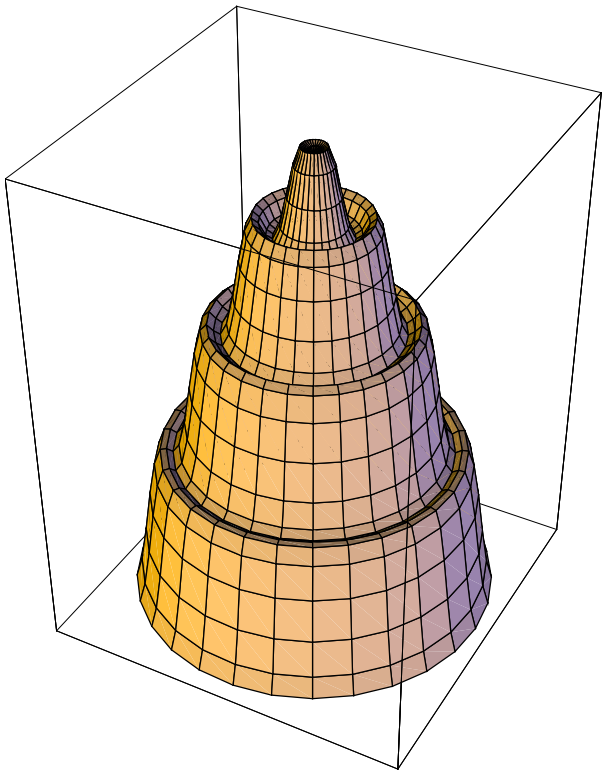
Questo package serve principalmente per generare delle figure tridimensionali facendo ruotare una curva attorno ad un asse. La curva può essere sia ad una sola variabile, a due oppure a tre, in questii due ultimi casi descritta parametricamente e dipendente da un parametro (d'altronde, ci serve una curva, non una superficie...):

`SurfaceOfRevolution[`*f*`, {`*x*`, `*xmin*`, `*xmax*`}]`	disegna la superfice di rotazione ottenendo facendo ruotare la curca definita da *f* nel piano *x*-*z* fra *xmin* e *xmax*
`SurfaceOfRevolution[{`f_x`, `f_z`}, {`*t*`, `*tmin*`, `*tmax*`}]`	disegna la superfice di rivolusione ottenuta ruotando la curva descritta parametricamente nel piano *x*-*z* nel parametro *t*
`SurfaceOfRevolution[{`f_x`, `f_y`, `f_z`}, {`*t*`, `*tmin*`, `*tmax*`}]`	analogo *a* sopra, ma per una curva parametrica *a* tre componenti

Vediamo subito un esempio:

In[238]:=
```
<< Graphics`SurfaceOfRevolution`
```

In[239]:=
```
SurfaceOfRevolution[-x^1.5 + Sin[4 x], {x, 0, 6},
 Axes → False,
 LightSources → {
   {{2, -2, 2}, RGBColor[0.7, 0.6, 0.8]},
   {{-2, -1, 1}, Hue[0.12]}}
]
```



Out[239]= - Graphics3D -

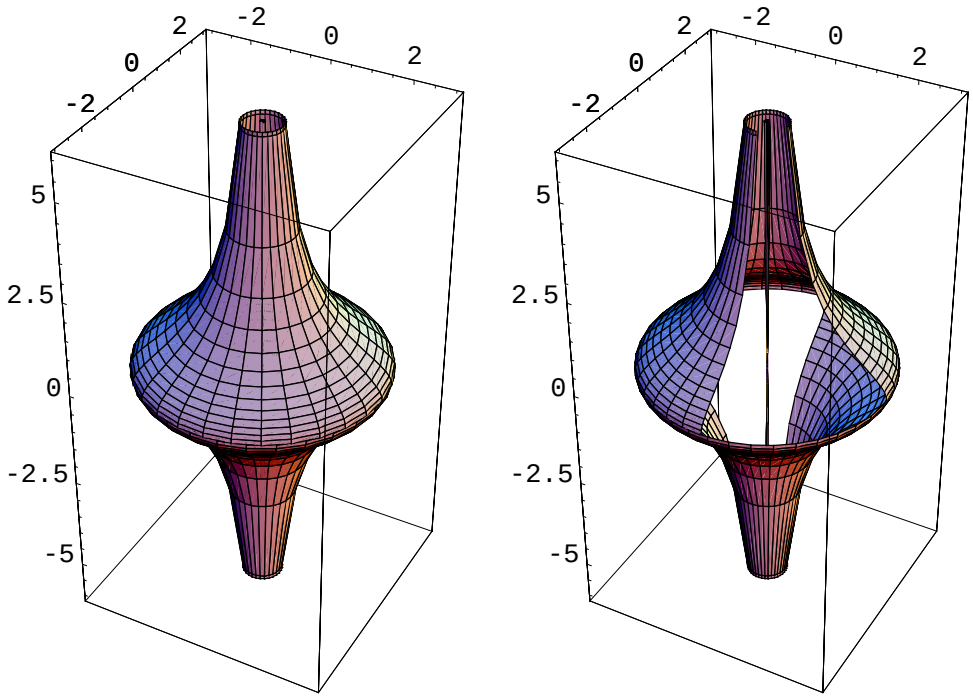
Per poter essere esguito, il comando utilizza ParametricPlot3D, per cui valgono tutte le opzioni valide per quest'ultimo comando.

L'esempio riportatao sotto, invece, defiinisce una superfice di rivoluzione ottenuta da una curva parametrica bidimensionale. Inoltre, se si aggiunge un secondo parametro, possiamo specificare l'angolo di rivoluzione:

In[240]:=
```
<< Graphics`Graphics`
```

In[241]:=
```
curva = {3 Cos[t], Tan[t]};
```

In[242]:=
```
DisplayTogetherArray[{{
   SurfaceOfRevolution[curva, {t, 0, Pi}],
   SurfaceOfRevolution[curva, {t, 0, Pi}, {u, 0, 3 Pi / 2}]
}}]
```



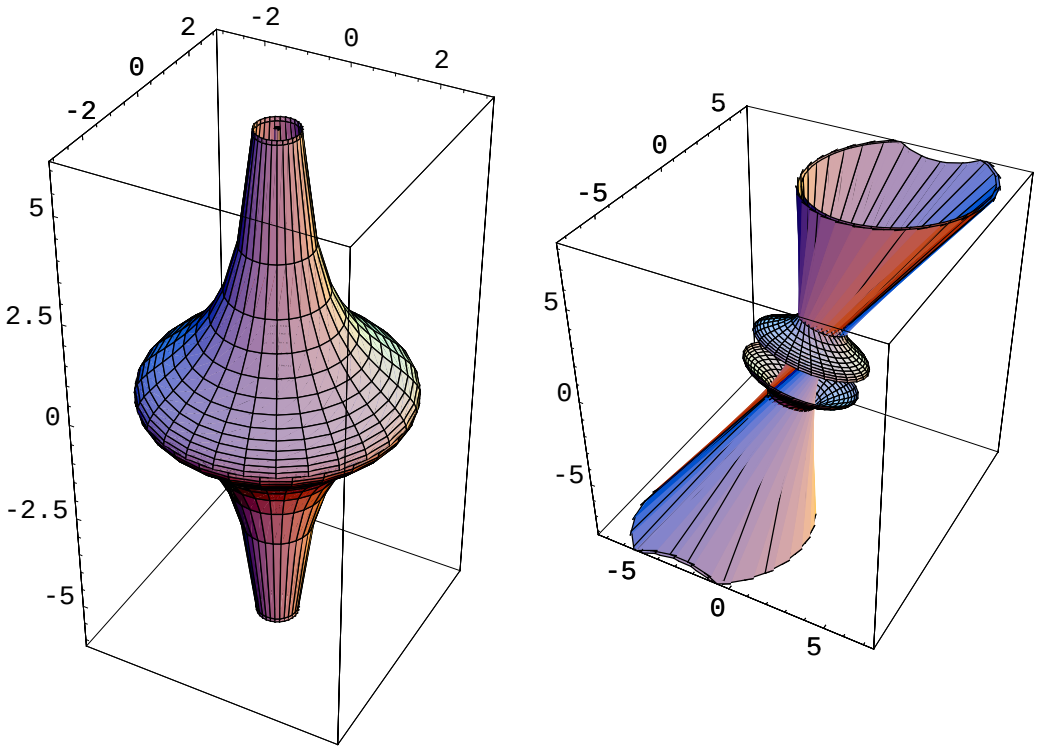
Out[242]= - GraphicsArray -

L'asse di rotazione, comunque sia definita la curva, è sempre dato dall'asse *z*. Se vogliamo cambiarlo, dobbiamo specificare il nuovo asse con la giusta opzione:

`RevolutionAxis -> {x, z}`	ruota la curva intorno all'asse di rotazione che connette l'origine degli assi con il punto specificato nel piano *x*-*z*
`RevolutionAxis -> {x, y, z}`	ruota la curva intorno all'asse che passa per l'origine e per il punto specificato

Pur mantenendo lo stesso angolo, la direzione dell'asse fa variare la superficie, perchè la curva è sempre definita nel piano x-z:

In[243]:=
```
DisplayTogetherArray[{{
   SurfaceOfRevolution[curva, {t, 0, Pi}],
   SurfaceOfRevolution[curva, {t, 0, Pi}, RevolutionAxis → {2, 3, 8}]
  }}]
```



Out[243]= - GraphicsArray -

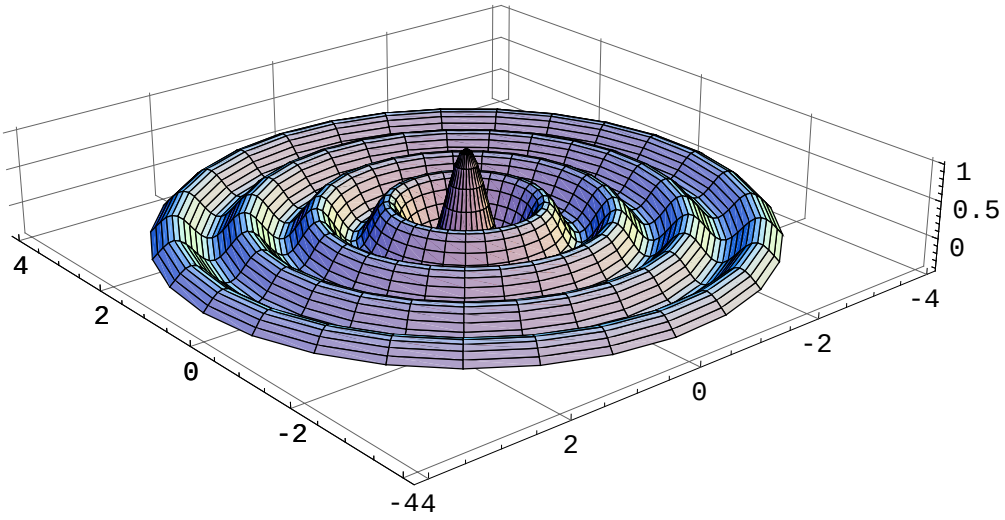
Per finire con questo package, vediamo il comando per effettuare la superfice di rivoluzione a partire da una curva definita da dei dati, invece che da una funzione:

`ListSurfaceOfRevolution[{` *point*$_1$`,` *point*$_2$`, … }]`	genera una superfice di rivoluzione a partire dalla curva specificata per punti
`ListSurfaceOfRevolution[{` *point*$_1$`,` *point*$_2$`, … }, {`*theta*`,` *thetamin*`,` *thetamax*`}]`	genra una superfice di rivoluzione con un determinato range per l'angolo di rivoluzione

Vediamo come funziona il comando:

In[244]:=
```
dati = Table[
   {BesselJ[0, 7 n], n}, {n, 0, 4, .07}
  ];
```

In[245]:=
```
ListSurfaceOfRevolution[
 dati, {t, 0, 2 Pi},
 RevolutionAxis → {3, 0},
 PlotRange → All,
 ViewVertical → {1, 0, 0},
 PlotPoints → 30,
 Boxed → False,
 FaceGrids → {{0, 1, 0}, {-1, 0, 0}, {0, 0, -1}}
]
```



Out[245]= `- Graphics3D -`

E con questo package ho finito (credo, spero...) di descrivere quelli riguardanti comandi grafici avanzati...

Vediamo il resto, adesso...

■ LinearAlgebra`FourierTrig`

Il package contiene le varianti reali della trasformata di Fourier, vedendola come le parti in seno e coseno:

`FourierCos[{`a_1`,` a_2`, ... ,` a_{n+1}`}]`	trasformata discreta del coseno
`FourierSin[{`a_1`,` a_2`, ... ,` a_{n+1}`}]`	trasformata discreta del seno

In[246]:=
```
<< LinearAlgebra`FourierTrig`
```

```
In[247]:= Table[1 - 2 ChebyshevT[2, x], {x, .1, .9, .03}]

Out[247]= {2.96, 2.9324, 2.8976, 2.8556, 2.8064, 2.75, 2.6864, 2.6156, 2.5376,
 2.4524, 2.36, 2.2604, 2.1536, 2.0396, 1.9184, 1.79, 1.6544, 1.5116,
 1.3616, 1.2044, 1.04, 0.8684, 0.6896, 0.5036, 0.3104, 0.11, -0.0976}

In[248]:= Chop[FourierCos[%]]

Out[248]= {13.241, 4.47343, -0.893379, 0.501916, -0.226638, 0.184256,
 -0.103225, 0.0968197, -0.0601015, 0.0609337, -0.0402234,
 0.0428777, -0.029518, 0.0326164, -0.0231674, 0.026318, -0.0191642,
 0.0222679, -0.0165554, 0.0196115, -0.0148469, 0.0178917,
 -0.0137685, 0.0168559, -0.0131714, 0.0163679, -0.01298}

In[249]:= FourierCos[%]

Out[249]= {2.96, 2.9324, 2.8976, 2.8556, 2.8064, 2.75, 2.6864, 2.6156, 2.5376,
 2.4524, 2.36, 2.2604, 2.1536, 2.0396, 1.9184, 1.79, 1.6544, 1.5116,
 1.3616, 1.2044, 1.04, 0.8684, 0.6896, 0.5036, 0.3104, 0.11, -0.0976}
```

Come potete vedere, si deve eseguire il calcolo su una lista di dati, ed inoltre i comandi sono normalizzati, nel senso che fare la trasformata della trasformata restituisce la lista principale.

■ LinearAlgebra`MatrixManipulation`

Questo package contiene varie funzioni per una manipolazione più agevole ed avanzata delle matrici, ed anche se la maggior parte di loro sono implementabili attraverso semplici funzioni che possiamo definire, la loro completezza, il loro numero permettono di cominciare subito a lavorare con le matrici senza perdere tempo iniziale a dover definire le nostre funzioni.

Le prime funzioni che andiamo a vedere sono quelle che ci permettono di ottenere una matrice unendo delle sottomatrici:

`AppendColumns[`m_1`, `m_2`, … ]`	unisce le colonne delle matrici m_1, m_2, …
`AppendRows[`m_1`, `m_2`, … ]`	unisce le righe delle matrici m_1, m_2, …
`BlockMatrix[`*blocks*`]`	unisce le righe e le colonne delle matrici definite nella lista *blocks* per formare una nuova matrice

Questi comandi ci permettono, in altre parole, di creare delle matrici a blocchi:

```
In[250]:= << LinearAlgebra`MatrixManipulation`
```

Definiamo due matrici:

```
In[251]:= mat1 = {{a, b}, {c, d}}; mat2 = {{1, 2}, {3, 4}};
```

Vediamo adesso la matrice che si ottinene unendo queste due:

```
In[252]:= AppendRows[mat1, mat2] // MatrixForm
```

```
Out[252]//MatrixForm=
(a  b  1  2)
(c  d  3  4)
```

```
In[253]:= AppendColumns[mat1, mat2] // MatrixForm
```

```
Out[253]//MatrixForm=
(a  b)
(c  d)
(1  2)
(3  4)
```

```
In[254]:= BlockMatrix[{{mat1.mat2, mat2.mat1}, {mat1, mat2}}] // MatrixForm
```

```
Out[254]//MatrixForm=
(a + 3 b  2 a + 4 b  a + 2 c  b + 2 d)
(c + 3 d  2 c + 4 d  3 a + 4 c  3 b + 4 d)
(a  b  1  2)
(c  d  3  4)
```

Questi comandi permettono di creare facilmente delle matrici a partire dai loro blocchi, evitando di dover scrivere cicli e quant'altro per eseguire gli stessi comandi senza usare il package.

Dopo aver visto questi comandi, andiamo a vedere quelli che ci permettono di estrarre parti di una matrice definita:

`TakeRows[`*mat*`, `*n*`]`	restituisce le prime *n* righe presenti in *mat*
`TakeRows[`*mat*`, -`*n*`]`	restituisce le ultime *n* righe presenti in *mat*
`TakeRows[`*mat*`, {`*m*`, `*n*`}]`	restituisce le righe dalla *m* alla *n* della matrice *mat*
`TakeColumns[`*mat*`, `*n*`]`	restituisce le prime *n* colonne in *mat*
`TakeColumns[`*mat*`, -`*n*`]`	restituisce le ultime *n* colonne in *mat*
`TakeColumns[`*mat*`, {`*m*`, `*n*`}]`	restituisce le colonne dalla *m* alla *n* della matrice *mat*
`TakeMatrix[`*mat*`, `pos_1`, `pos_2`]`	restituisce la sottomatrice limitata dagli elementi che si trovano nella posizione pos_1 e pos_2
`SubMatrix[`*mat*`, `*pos*`, `*dim*`]`	restituisce la sottomatrice *mat* di dimensione *dim* che parte dalla posizione *pos*

Creiamo una matrice 5x4:

In[255]:=
```
mat = ToExpression[
   Table["a" <> ToString[m] <> ToString[n], {m, 5}, {n, 4}]
  ];
mat // MatrixForm
```

Out[256]//MatrixForm=

$$\begin{pmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \\ a31 & a32 & a33 & a34 \\ a41 & a42 & a43 & a44 \\ a51 & a52 & a53 & a54 \end{pmatrix}$$

Per creare gli elementi della matrice, prima li ho creati come stringhe, e poi li ho convertriti in simboli di nuovo, per non aver a che fare con delle stringhe come elementi; a questo punto, possiamo cominciare a manipolare la nostra matrice come più ci piace. Per esempio, possiamo prendere le prime due righe, oppure le ultime due:

In[257]:=
```
TakeRows[mat, 2] // MatrixForm
```

Out[257]//MatrixForm=

$$\begin{pmatrix} a11 & a12 & a13 & a14 \\ a21 & a22 & a23 & a24 \end{pmatrix}$$

In[258]:=
```
TakeRows[mat, -2] // MatrixForm
```

Out[258]//MatrixForm=

$$\begin{pmatrix} a41 & a42 & a43 & a44 \\ a51 & a52 & a53 & a54 \end{pmatrix}$$

Analogamente possiamo fare per le colonne:

In[259]:=
```
TakeColumns[mat, 2] // MatrixForm
```

Out[259]//MatrixForm=

$$\begin{pmatrix} a11 & a12 \\ a21 & a22 \\ a31 & a32 \\ a41 & a42 \\ a51 & a52 \end{pmatrix}$$

In[260]:=
```
TakeColumns[mat, -2] // MatrixForm
```

Out[260]//MatrixForm=

$$\begin{pmatrix} a13 & a14 \\ a23 & a24 \\ a33 & a34 \\ a43 & a44 \\ a53 & a54 \end{pmatrix}$$

Vediamo di prendere, adesso, la sottomatrice centrale. Possiamo farlo in due modi: specificando le dimensioni ed il primo elemento della sottomatrice, oppure specificandp il primo e l'ultimo elemento:

```
In[261]:= TakeMatrix[mat, {2, 2}, {4, 3}] // MatrixForm
```

```
Out[261]//MatrixForm=
( a22  a23 )
( a32  a33 )
( a42  a43 )
```

```
In[262]:= SubMatrix[mat, {2, 2}, {3, 2}] // MatrixForm
```

```
Out[262]//MatrixForm=
( a22  a23 )
( a32  a33 )
( a42  a43 )
```

Nel primo caso abbiamu utlizzato gli indici del primo e dell'ultimo elemento, mentre nel secondo abbiamo sfruttato le dimensioni.

Oltre a questi semplici comandi di manipolazione, il package contiene anche comandi per creare dei particolari tipi di matrici:

`UpperDiagonalMatrix[`*f*`, `*n*`]`	crea una matrice diagonale superiore *n*×*n* con glielementi non nulli pari a *f*[*i*, *j*]
`LowerDiagonalMatrix[`*f*`, `*n*`]`	crea una matrice diagonale inferiore *n*×*n* con glielementi non nulli pari a *f*[*i*, *j*]
`TridiagonalMatrix[`*f*`, `*n*`]`	crea una matrice tridiagonale *n*×*n* con gli elementi della tridiagonale pari a *f*[*i*, *j*]
`ZeroMatrix[`*n*`]`	crea una matrice nulla *n*×*n*
`ZeroMatrix[`*m*`, `*n*`]`	crea una matrice nulla *m*×*n*
`HilbertMatrix[`*n*`]`	crea una matrice di Hilbert *n*×*n*, i cui elementi sono dati da 1/(*i*+*j*−1)
`HilbertMatrix[`*m*`, `*n*`]`	crea una matrice di Hilbert *m*×*n*
`HankelMatrix[`*n*`]`	crea una matrice di Hankel *n*×*n* con gli elementi della prima colonna pari a 1, 2, …, *n* la seconda da 2, 3, …, *n* ,0 e così via
`HankelMatrix[`*list*`]`	crea una matrice di Hankel con la prima colonna definita da *list* e che segue il ragionamento di sopra
`HankelMatrix[`*col*`, `*row*`]`	crea una matrice di Hankel con la prima colonna data dalla lista *col* e dall'ultima riga data dalla lista

Il funzionamento di questi comandi non è complicato:

In[263]:=
```
UpperDiagonalMatrix[Sin[2 #1 + a #2] &, 4] // MatrixForm
```

Out[263]//MatrixForm=

$$\begin{pmatrix} \text{Sin}[2+a] & \text{Sin}[2+2a] & \text{Sin}[2+3a] & \text{Sin}[2+4a] \\ 0 & \text{Sin}[4+2a] & \text{Sin}[4+3a] & \text{Sin}[4+4a] \\ 0 & 0 & \text{Sin}[6+3a] & \text{Sin}[6+4a] \\ 0 & 0 & 0 & \text{Sin}[8+4a] \end{pmatrix}$$

Notate come il comando richieda solamente l'head della funzione oppure, in modo alternativo, la corrispondente funzione pura:

In[264]:=
```
LowerDiagonalMatrix[Plus, 5] // MatrixForm
```

Out[264]//MatrixForm=

$$\begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 3 & 4 & 0 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 \\ 5 & 6 & 7 & 8 & 0 \\ 6 & 7 & 8 & 9 & 10 \end{pmatrix}$$

Funziona anche con gli head classici...

Per la matrice di Hilbert, invece, occorrono solamente le dimensioni:

In[265]:=
```
HilbertMatrix[4] // MatrixForm
```

Out[265]//MatrixForm=

$$\begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \end{pmatrix}$$

Le matrici tridiagonali, invece, sono particolari tipi di matrici, in cui sono diversi da zero gli elementi della diagonale principale, della sovradiagonale e della sottodiagonale:

In[266]:=
```
TridiagonalMatrix[f, 5] // MatrixForm
```

Out[266]//MatrixForm=

$$\begin{pmatrix} f[1,1] & f[1,2] & 0 & 0 & 0 \\ f[2,1] & f[2,2] & f[2,3] & 0 & 0 \\ 0 & f[3,2] & f[3,3] & f[3,4] & 0 \\ 0 & 0 & f[4,3] & f[4,4] & f[4,5] \\ 0 & 0 & 0 & f[5,4] & f[5,5] \end{pmatrix}$$

Le matrici di Hankel, invece, sono delle matrici che si ottengono scrivendo la prima colonna; la colonna generica si ottiene shiftando di un posto verso l'alto la colonna precedente, e riempendo l'ultimo elemento con uno zero:

In[267]:=
```
HankelMatrix[5] // MatrixForm
```

Out[267]//MatrixForm=

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 0 \\ 3 & 4 & 5 & 0 & 0 \\ 4 & 5 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Possiamo anche definire da soli la nostra prima colonna:

In[268]:=
```
HankelMatrix[{a, b, c, d, e}] // MatrixForm
```

Out[268]//MatrixForm=

$$\begin{pmatrix} a & b & c & d & e \\ b & c & d & e & 0 \\ c & d & e & 0 & 0 \\ d & e & 0 & 0 & 0 \\ e & 0 & 0 & 0 & 0 \end{pmatrix}$$

Inoltre, possiamo riempire anche la parte sottostante, definendo l'ultima riga, che sarà shiftata in maniera equivalente:

In[269]:=
```
HankelMatrix[{a, b, c, d, e}, {e, 2, 3, 4, 5, 6}] // MatrixForm
```

Out[269]//MatrixForm=

$$\begin{pmatrix} a & b & c & d & e & 2 \\ b & c & d & e & 2 & 3 \\ c & d & e & 2 & 3 & 4 \\ d & e & 2 & 3 & 4 & 5 \\ e & 2 & 3 & 4 & 5 & 6 \end{pmatrix}$$

Naturalmente, se si costruisce in questa maniera, l'ultimo elemento della prima lista deve essere uguale al primo elemento della seconda lista, dato che l'ultimo elemento della prima colonna ed il primo elemento dell'ultima riga coincidono: se si specifica diversamente il comando non funziona:

In[270]:=
```
HankelMatrix[{1, 2, 3}, {a, b, c}]
```

Out[270]=
```
HankelMatrix[{1, 2, 3}, {a, b, c}]
```

In questo caso non siamo in grado di costruire la matrice.

Un altro comando molto iutile è quello che permette, dato un sistema lineare scritto sotto forma di equazioni, di ricavarne la matrice dei coefficienti e quella dei termini noti:

`LinearEquationsToMatrices[` *eqns*`,` *vars*`]`	restituisce una lista del tipo {*mat*, *vec*}, dove *mat* rappresenta la matrice dei coefficienti del sistema lineare nelle variabili specificate, e *vec* rappresenta il vettore dei termini noti

Supponiamo di avere il seguente sistema:

```
In[271]:= sistema = {
            a x + b y == 4 + t,
            5 t + p == 4 x + p y,
            x - y == t
          };
```

Se adesso voglio andare a trovarmi la forma matriciale basta scrivere:

```
In[272]:= LinearEquationsToMatrices[sistema, {x, y, t}]

Out[272]= {{{a, b, -1}, {-4, -p, 5}, {1, -1, -1}}, {4, -p, 0}}
```

Sono date in un'unica lista, ma ovviamente noi sappiamo come estrarre gli elementi, vero?

```
In[273]:= coeff = %[[1]]; termnoti = %[[2]];
```

```
In[274]:= coeff // MatrixForm
```

Out[274]//MatrixForm=

$$\begin{pmatrix} a & b & -1 \\ -4 & -p & 5 \\ 1 & -1 & -1 \end{pmatrix}$$

```
In[275]:= termnoti // MatrixForm
```

Out[275]//MatrixForm=

$$\begin{pmatrix} 4 \\ -p \\ 0 \end{pmatrix}$$

Notate come, dato che ho concatenato due operazioni in un unico Input, % faccia riferimento all'output precedente, che corrisponde all'ultimo Out creato prima della riga, e come nel secondo comando concatenato, di conseguenza, uso % e non %%. Si impara sempre qualcosa di nuovo, vero???

Sono anche presenti due comandi per la fattorizzazione di una matrice:

`PolarDecomposition[`*mat*`]`	restituisce una lisa nella forma `{`*u*`, s}`, dove *s* rappresenta una matrice definita positiva, mentre *u*.*u** è uguale alla matrice identità, e *u*.*s*=*m a t*
`LUMatrices[`*lu*`]`	restituisce una lista nella forma `{`*l*`, u}`, dove *l* ed *u* rappresentano le matrici diagonali inferiori, *e* diagonale superiore di una fattorizzazione LU di una matrice, ed *lu* rappresenta il primo elemento della decomposizione `LUDecomposition[`*mat*`]`

Ovviamente, dovete sapere cosa rappresenta una decomposizione polare, ed usarla quando vi serve. Un consiglio: utilizzate questo comando soltanto con valori numerici; utilizzare numeri esatti oppure costanti di solito da luogo a risultati estremamente lunghi ed inutili. Provate con la solita matrice {{*a*, *b*}, {*c*, *d*}}, e mi darete ragione:

In[276]:=
```
mat = {
   {3., 5., 4.},
   {7., -6., -2.},
   {4., 4., 7.}
  };
```

In[277]:=
```
PolarDecomposition[mat]
```

Out[277]=
```
{{{0.529976, 0.83848, -0.126798}, {0.815019, -0.544932, -0.196962},
  {0.234245, -0.00104205, 0.972177}}, {{8.23204, -1.30325, 2.12958},
  {-1.30325, 7.45782, 4.43649}, {2.12958, 4.43649, 6.69197}}}
```

In[278]:=
```
MatrixForm[%[[1]]]
```

Out[278]//MatrixForm=

$$\begin{pmatrix} 0.529976 & 0.83848 & -0.126798 \\ 0.815019 & -0.544932 & -0.196962 \\ 0.234245 & -0.00104205 & 0.972177 \end{pmatrix}$$

In[279]:=
```
GridBox[
  {
   {MatrixForm[%[[1]]],
    MatrixForm[%[[2]]]
   }
  }
 ] // DisplayForm
```

Out[279]//DisplayForm=

$$\begin{pmatrix} 0.529976 \\ 0.83848 \\ -0.126798 \end{pmatrix} \quad \begin{pmatrix} 0.815019 \\ -0.544932 \\ -0.196962 \end{pmatrix}$$

Come potete vedere ho usato il comando GridBox, che serve per creare delle tabelle. Andate a vedervi i comandi di formattazione... Comunque, lo scopo era farvi vedere più chiaramente le due matrici così ottenute.

```
In[280]:= %%[[1]].%%[[2]]

Out[280]= -1.953×10^-16
```

Come potete vedere, il prodotto di queste due matrici è effettivamente la matrice originale.

Vediamo invece l'utilità del secondo comando, LUMatrices. Effettuiamo prima di tutto la decomposizione LU di una matrice, con il comando predefinito di *Mathematica*:

```
In[281]:= LUDecomposition[mat]

Out[281]= {{{7., 0.428571, 0.571429}, {-6., 7.57143, 0.981132},
            {-2., 4.85714, 3.37736}}, {2, 1, 3}, 9.46927}
```

```
In[282]:= %[[1]] // MatrixForm

Out[282]//MatrixForm=
          ( 7.   0.428571  0.571429 )
          (-6.   7.57143   0.981132 )
          (-2.   4.85714   3.37736  )
```

Il comando restituisce una lista di tre elementi: il primo è rappresentato dalla matrice ottenuta combinando la matrice diagonale inferiore e superiore: il secondo elemento è il vettore delle permutazioni dei pivot, ed il terzo un numero rappresenta il numero L^{∞}. Se vogliamo adesso estrarre dal primo elemento di questo risultato le due matrici, invece di averne una singola, possiamo creare un programmino, oppure utilizzare il comando definito da questo package:

```
In[283]:= ris = LUMatrices[%%[[1]]]; GridBox[
            {
             {MatrixForm[ris[[1]]],
              MatrixForm[ris[[2]]]
             }
            }
           ] // DisplayForm

Out[283]//DisplayForm=
          ( 1.        0.        0. )   ( 7.  -6.       -2.      )
          ( 0.428571  1.        0. )   ( 0   7.57143  4.85714 )
          ( 0.571429  0.981132  1. )   ( 0   0        3.37736 )
```

Come poete vedere adesso ris contiene le due matrici correttamente suddivise nella triangolare inferiore e superiore:

In[284]:=
```
ris[[1]].ris[[2]]
```

Out[284]=
```
{{7., -6., -2.}, {3., 5., 4.}, {4., 4., 7.}}
```

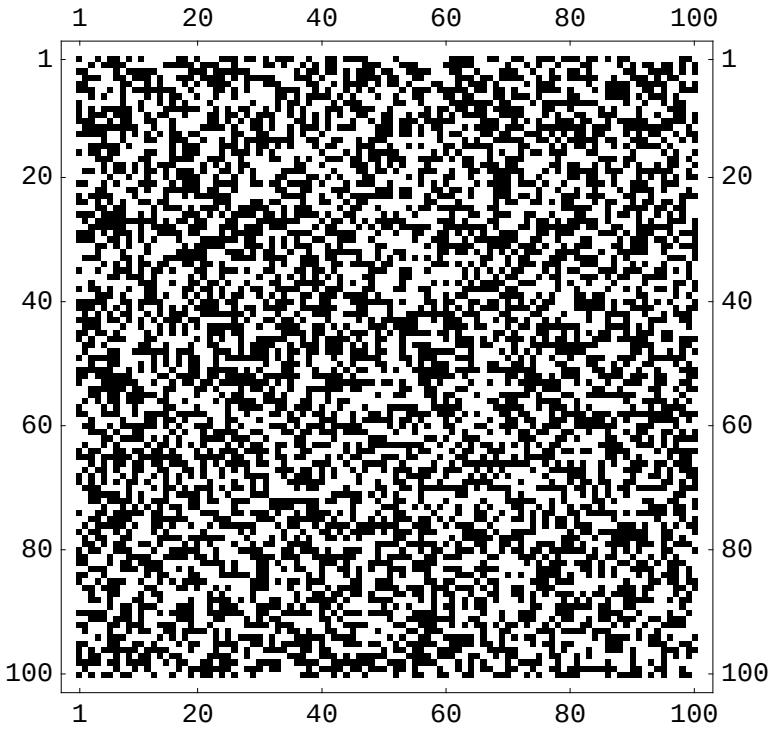
Di nuovo la matrice di partenza...

Un altro comando utile, specialmente quando si ha a che fare con matrici di grandi dimensioni, è il seguente:

`MatrixPlot[`*mat*`]`	mostra graficamente la struttura di *mat*
`MatrixPlot[`*mat*`, `*cen*`]`	mostra graficamente la struttura di *mat* con valore centrale *cen*

In pratica, crea qualcosa di simile ad un grafico di densità; tuttavia non è la stessa cosa, perchè questo comando mostra solamente la struttura: una cella bianca per un elemento nullo, ed una nera per un elemento non nullo:

In[285]:=
```
MatrixPlot[Table[Random[Integer], {100}, {100}]]
```

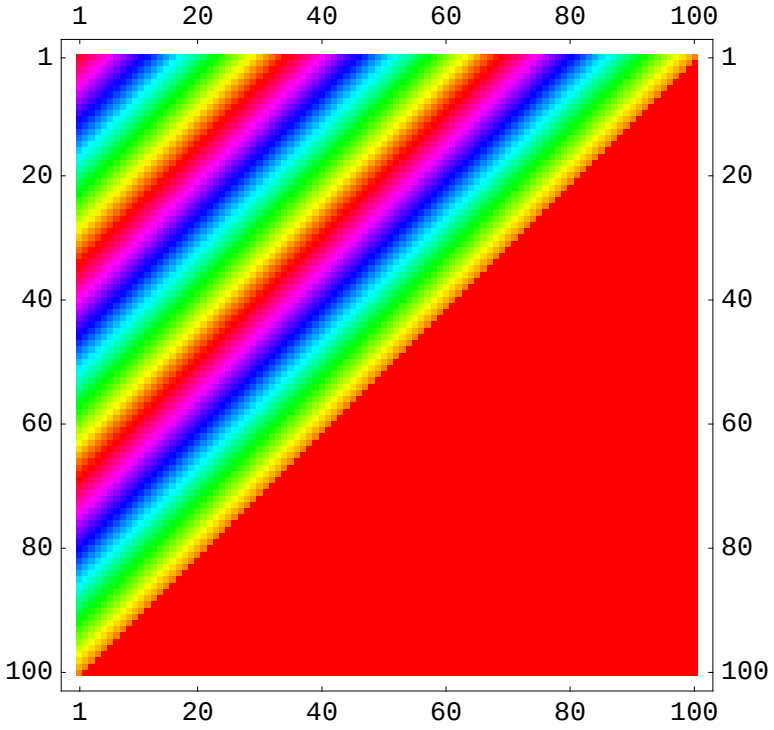


Out[285]=
```
- Graphics -
```

Ci sono un paio di cosette da aggiungere riguardo questo comando: prima di tutto, bisogna notare come possiamo definire comunque una funzione per il colore della matrice, passando dal semplice bianco e nero ad una scala rappresentante i valori della matrice, con la solita opzione ColorFunction:

In[286]:=
```
mat2 = HankelMatrix[100];
```

In[287]:=
```
MatrixPlot[mat2, ColorFunction → (Hue[# / 1.03] &)]
```

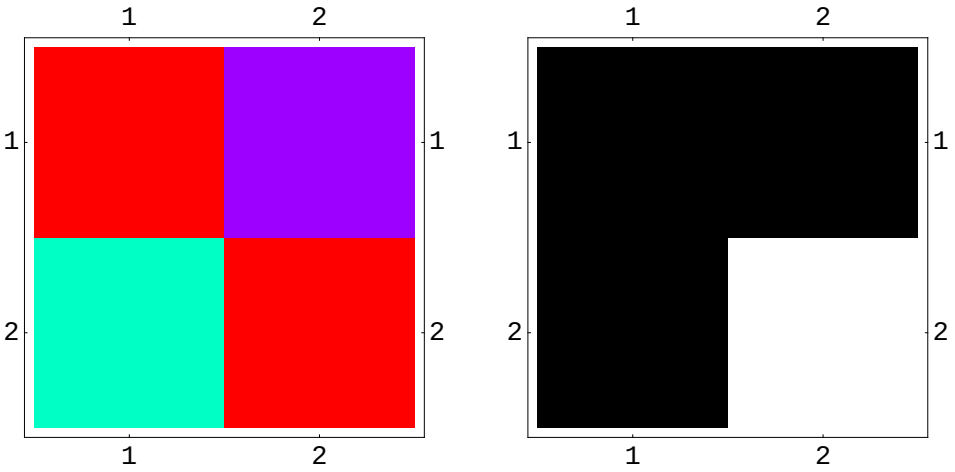


Out[287]=
```
- Graphics -
```

Come potete vedere, in questo caso abbiamo utilizzato una funzione per evidenziare con il colore la quantità numerica contenuta nella matrice. Tuttavia, questo vale solamente per matrici numeriche. Se la matrice contiene anche elementi simbolici. Per rendercene conto guardiamo il seguente esempio:

In[288]:=
```
<< Graphics`Graphics`
```

In[289]:=
```
DisplayTogetherArray[{{
   MatrixPlot[{{a, 1}, {.6, 0}}, ColorFunction → (Hue[# / 1.3] &)],
   MatrixPlot[{{a, 1}, {.6, 0}}]
}}]
```



Out[289]= `- GraphicsArray -`

Come possiamo vedere, la struttura della matrice contiene solamente un elemento nullo. Questo viene riprodotto correttamente nella seconda matrice. Nella prima, invece, il comando non è in grado di valutare un colore per il simbolo, per cui viene colorato nella stessa maniera in cui viene colorato l'elemento nullo, portando ad una errata interpretazione della struttura della matrice.

L'altra opzione è la seguente:

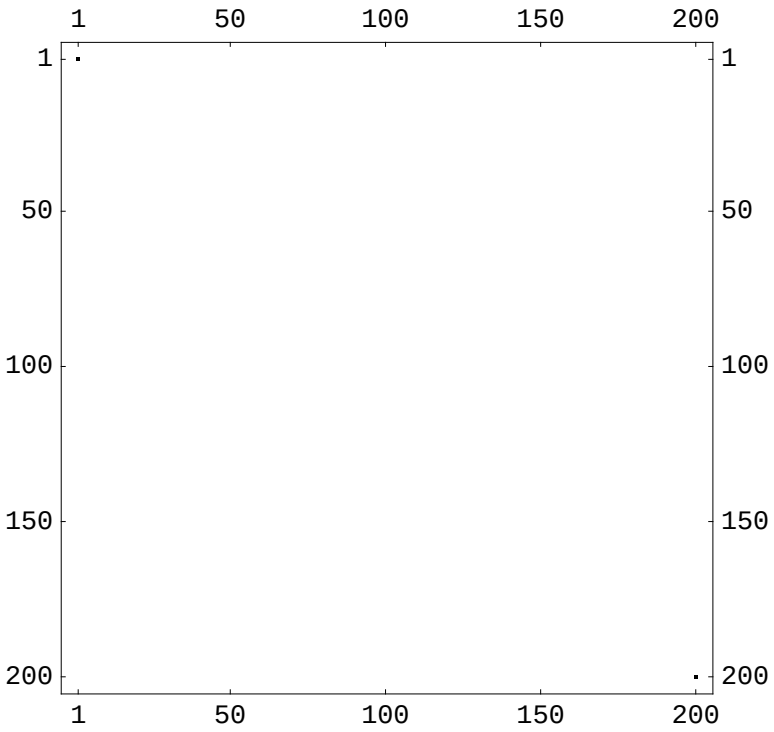
`MaxMatrixSize -> n`	massima dimensione della per la visualizzazione della matrice

Prima di disengare la matrice, il comando effettua un downsampling per poterla visualizzare sullo schermo. Per fare questo, raggruppa gruppi di elementi in un unica cella quadrata. Se almento un elemento del gruppo di elemento della cella è non nullo, allora la cella corrispondente risulterà non vuota. Supponiamo, per esempio, di avere la seguente matrice sparsa, dove solamente il primo e l'ultimo elemeno sono non nulli:

In[290]:=
```
mat3 = SparseArray[{{1, 1} → 1, {200, 200} → 1}]
```

Out[290]=
```
SparseArray[<2>, {200, 200}]
```

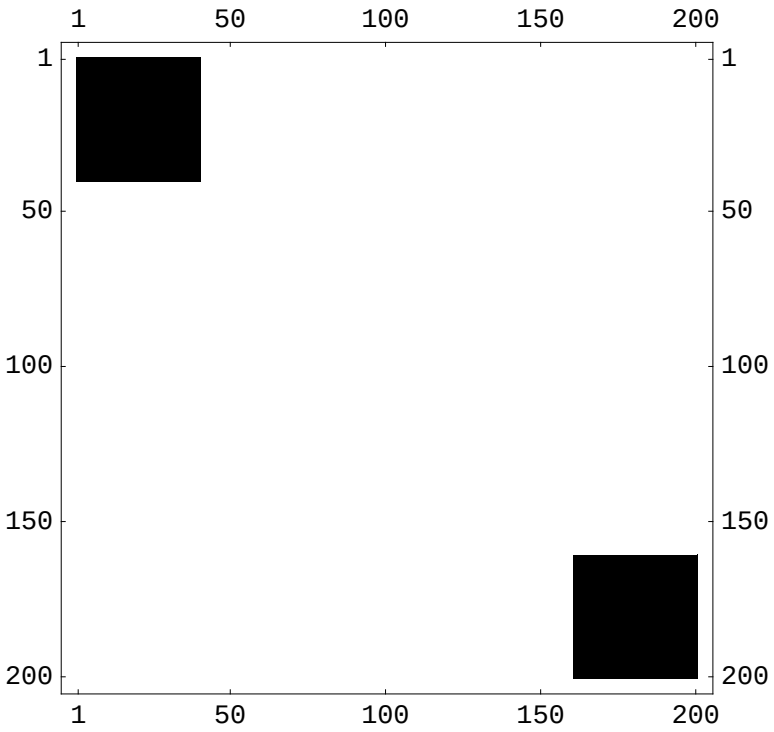
In[291]:= **MatrixPlot[mat3]**



Out[291]= **-** Graphics **-**

Come potete vedere, il grafico è vuoto, eccezion fatta per i due angolini che si fanno pure fatica a distinguerli. Veriamo di effettuare un downsampling;

In[292]:= **MatrixPlot[mat3, MaxMatrixSize → 5]**



Out[292]= **- Graphics -**

Come possiamo vedere, abbiamo effettuato un downsampling, visualizzando una matrice 5 x 5 . Siccome il primo ed ultimo elemento di questa matrice contengono elemento non nulli della matrice principale, sono considerati anch'essi non nulli. Questo può agevolare la visualizzazione di matrici grandi, ma notate come si vengano a perdere, in questo modo, informazioni sulla struttura, apparendo più grossolana.

■ LinearAlgebra`Orthogonalization`

Questo package contiene alcuni comandi utili per gestirvi le basi dei vettori, andandovi a trovare le basi ortonormali a partire da una qualsiasi, a proiettare un vettore su di un altro e così via.

Il comando che tutti conoscerete (almeno quelli che hanno fatto Teoria dei Segnali), riguarda la procedura di Gram-Schmidt. Il comando corrispondente effettua l'ortonormalizzazione in maniera simbolica. Tutttavia a volte può dare problemi, e verrà dato un altro comando che lavora esclusivamente su dati numerici, risultando più stabile.

Vediamo i comandi principali:

`GramSchmidt[{`v_1`, `v_2`, … }]`	genera una base ortonormale a partire dalla lista di vettori reali
`Normalize[`*vect*`]`	normalizza il vettore *vect*
`Projection[`$vect_1$`, `$vect_2$`]`	restituisce la proiezione ortogonale di $vect_1$ su $vect_2$

Vediamo un esempio pratico:

In[293]:=
```
<< LinearAlgebra`Orthogonalization`
```

In[294]:=
```
vettori = {{1, 3, 4, 6}, {-3, 6, -3, 2}, {0, 3, 3, 1}, {1, 4, 4, 4}};
```

In[295]:=
```
{b1, b2, b3, b4} = GramSchmidt[vettori]
```

Out[295]=

$$\left\{\left\{\frac{1}{\sqrt{62}}, \frac{3}{\sqrt{62}}, 2\sqrt{\frac{2}{31}}, 3\sqrt{\frac{2}{31}}\right\},\right.$$
$$\left\{-\frac{201}{\sqrt{209002}}, \frac{327}{\sqrt{209002}}, -123\sqrt{\frac{2}{104501}}, 17\sqrt{\frac{2}{104501}}\right\},$$
$$\left\{-\frac{38}{\sqrt{347213}}, \frac{1416}{5\sqrt{347213}}, \frac{356}{\sqrt{347213}}, -\frac{1863}{5\sqrt{347213}}\right\},$$
$$\left.\left\{\frac{9}{\sqrt{103}}, \frac{17}{5\sqrt{103}}, -\frac{3}{\sqrt{103}}, -\frac{6}{5\sqrt{103}}\right\}\right\}$$

Come potete vedere, abbiamo ottenuto la normalizzazione in maniera simbolica.

Inoltre, possimamo lavorare con le opzioni di questo comando per poter comandare l'ortonormalizzazione:

`GramSchmidt[{`v_1`, `v_2`, … }, InnerProduct -> `*func*`]`	genera un set ortonormale, utilizzando come prodotto interno dello spazio la funzione *func*
`Normalize[`*vect*`, InnerProduct -> `*func*`]`	normalizza *vect* utilizzando il prodotto interno dello spazio *func*
`Projection[`$vect_1$`, `$vect_2$`, InnerProduct`	restitusce la proiezione ortogonale di $vect_1$ su $vect_2$ utilizzando il prodotto interno dello spazio
`GramSchmidt[{`v_1`, `v_2`, … }, Normalized -> False]`	Crea una base ortogonale, senza andarla *a* normalizzare

Ripetiamo lo stesso procedimento di prima, ma stavolta evitando di normalizzare i vettori della base:

In[296]:=
```
{b1, b2, b3, b4} = GramSchmidt[vettori, Normalized → False]
```

Out[296]=
$$\left\{\{1, 3, 4, 6\}, \left\{-\frac{201}{62}, \frac{327}{62}, -\frac{123}{31}, \frac{17}{31}\right\}, \left\{-\frac{570}{3371}, \frac{4248}{3371}, \frac{5340}{3371}, -\frac{5589}{3371}\right\}, \left\{\frac{261}{515}, \frac{493}{2575}, -\frac{87}{515}, -\frac{174}{2575}\right\}\right\}$$

Abbiamo ottenuto la base di poco fa, evitando di normalizzare i vettori. Se li andiamo a normalizzare, riotteniamo la base di poco fa:

In[297]:=
```
Normalize /@ %
```

Out[297]=
$$\left\{\left\{\frac{1}{\sqrt{62}}, \frac{3}{\sqrt{62}}, 2\sqrt{\frac{2}{31}}, 3\sqrt{\frac{2}{31}}\right\}, \left\{-\frac{201}{\sqrt{209002}}, \frac{327}{\sqrt{209002}}, -123\sqrt{\frac{2}{104501}}, 17\sqrt{\frac{2}{104501}}\right\}, \left\{-\frac{38}{\sqrt{347213}}, \frac{1416}{5\sqrt{347213}}, \frac{356}{\sqrt{347213}}, -\frac{1863}{5\sqrt{347213}}\right\}, \left\{\frac{9}{\sqrt{103}}, \frac{17}{5\sqrt{103}}, -\frac{3}{\sqrt{103}}, -\frac{6}{5\sqrt{103}}\right\}\right\}$$

Ho applicato la funzione Normalize ad ogni elemento della lista, tramite /@ che è il modo abbreviato di utilizzare Map. Ma questo ve lo ricordavate, vero???

Possiamo anche lavorare con spazi funzionali, dove un elemento è costituito da una combinazione lineare di funzioni (qualcuno ha detto Fourier?): in questo caso bisogna specificare, al posto dei vettori, le funzioni, ed inoltre bisogna specificare il prodotto interno dello spazio. Nel caso delle funzioni, di solito si utilizza l'integrale. Vediamo adesso la base per uno spazio i cui elementi si possono scrivere come $a\,\mathrm{Cos}[x] + b\,x\,\mathrm{Sin}[x] + c\,x^2$, e il prodotto interno è dato da $\int_{-\pi}^{\pi} \mathrm{v2}\,\mathrm{v1}\,dx$:

In[298]:=
```
base = {Cos[x], x Sin[x], x^2};
```

In[299]:=
```
GramSchmidt[base, InnerProduct → (Integrate[#1 #2, {x, -π, π}] &)] //
 FullSimplify
```

Out[299]=
$$\left\{\frac{\text{Cos}[x]}{\sqrt{\pi}},\ \sqrt{\frac{3}{-9\,\pi+4\,\pi^3}}\ (\text{Cos}[x]+2\,x\,\text{Sin}[x]),\right.$$
$$\sqrt{\frac{-45+20\,\pi^2}{-11040\,\pi+3040\,\pi^3-258\,\pi^5+8\,\pi^7}}$$
$$\left.\left(x^2+4\,\text{Cos}[x]-\frac{12\,(-7+\pi^2)\,(\text{Cos}[x]+2\,x\,\text{Sin}[x])}{-9+4\,\pi^2}\right)\right\}$$

Questa rappresenta la nostra base ortonormale per lo spazio definito da quelle funzioni.

Consideriamo adesso una gaussiana:

In[300]:=
```
gaussiana[x_] := Exp[-x^2 / (2 σ^2)]
```

Come possiamo vedere, manca in questo caso il coefficiente di normalizzazione. Possiamo calcolare la funzione normalizzata o facendo l'integrale e dividendo la funzione per il risultato, oppure usare Normalize, definendo il prodotto interno dello spazio, cioè come si calcola il modulo:

In[301]:=
```
Normalize[gaussiana[x],
 InnerProduct → (Integrate[#1 #2, {x, -∞, ∞}] &)]
```

Out[301]=
$$e^{-\frac{x^2}{2\sigma^2}} \Bigg/ \left(\sqrt{\text{If}\left[\text{Im}[\sigma]^2 < \text{Re}[\sigma]^2,\ \frac{\sqrt{\pi}}{\sqrt{\frac{1}{\sigma^2}}},\right.}\right.$$
$$\left.\left.\text{Integrate}\left[e^{-\frac{x^2}{\sigma^2}},\ \{x,\ -\infty,\ \infty\},\ \text{Assumptions} \to \text{Im}[\sigma]^2 \geq \text{Re}[\sigma]^2\right]\right]\right)$$

MMMMmmmmm.... si vede che in questo caso dobbiamo specificare un paio di cosette... Infatti, non avendo fatto nessuna assunzione per σ, viene trattata come caso generale, anche se in questo è incluso il caso che ci interessa. Specifichiamo che la variabile deve essere reale positiva:

In[302]:=
```
Assuming[
 σ > 0,
 Normalize[gaussiana[x],
  InnerProduct → (Integrate[#1 #2, {x, -∞, ∞}] &)]
]
```

Out[302]=
$$\frac{e^{-\frac{x^2}{2\sigma^2}}}{\pi^{1/4}\sqrt{\sigma}}$$

Comep possiamo vedere, avendo fatto la giusta assunzione per la deviazione standard, il risultato è corretto ed è quello che ci aspettavamo...

Vediamo adesso un esempio di Projection, prioiettando un vettore su di un altro:

```
In[303]:= Projection[{1, 1, 1}, {1, 5, 9}]
```

$$\text{Out[303]=} \left\{\frac{15}{107}, \frac{75}{107}, \frac{135}{107}\right\}$$

In questo caso ho ottenuto la proiezione del vettore sull'altro...

```
In[304]:= Projection[{1, 0, 0}, {0, a, 0}]

Out[304]= {0, 0, 0}
```

In questo caso i due vettori sono ortogonali fra di loro, e la proiezione di consegenza è nulla. Consideriamo adesso questo esempio:

```
In[305]:= basenum = {{0, 0, 1.}, {1., 0, 0},
              {-0.12988785514152842, 0.3997814966837186, 0.5468181006215335},
              {1.013982920708332, -0.02721531177817664, -0.18567966396292607}};
```

Effettuiamo l'ortonormalizzazione:

```
In[306]:= GramSchmidt[basenum]

Out[306]= {{0, 0, 1.}, {1., 0, 0.}, {0., 1., 0.}, {0., -1., 0.}}
```

C'è un problema... Risultano quattro vettori, mentre la base è di tre vettori, in questo caso. Questo è uno dei (rarissimi) casi in cui l'algoritmo di Gram-Schmidt non funziona, e dobbiamo utilizzare l'altro comando:

```
In[307]:= Householder[basenum]

Out[307]= {{0., 0., -1.}, {-1., 0., 0.}, {0., 1., 0.}, {0., 0., 0.}}
```

In questo caso si riconosce il vettore nullo, e di conseguenza la base restituita è di soli tre vettori, com'è giusto che sia.

■ Miscellaneous`Units`

Questo package è utile soprattutto negli esercizi dove conta la fisica, ed il risultato è pesantemente influenzato dalle unità di misura, cosa che in ingegneria capita abbastanza spesso, direi...

Il comando principale è quello che ci permette di effettuare la conversione fra due unità di misura:

`Convert[`*old*`, `*newunits*`]`	converte *old* in una forma che coinvolge una combinazione di *newunits*

Vediamo come sia facile utilizzare le unità di misura con questo package:

In[308]:=
```
<< Miscellaneous`Units`
```

In[309]:=
```
Convert[2314 Meter / Second, Kilo Meter / Hour]
```

Out[309]=
$$\frac{41652\ \text{Kilo Meter}}{5\ \text{Hour}}$$

Inoltre, sono definite anche i prefissi per le unità di misura; anche nel caso di sopra, il chilometro è stato definito come metro per Kilo, che rapresenta 10^3, come tutti sappiamo.

Un comando a parte merita la temperatura, perchè non è una conversione semplicemente moltiplicativa:

In[310]:=
```
ConvertTemperature[37, Celsius, Fahrenheit]
```

Out[310]=
$$\frac{493}{5}$$

Inoltre, possiamo prendere una grandezza, ed espromerla sotto forma delle unità di misura standard:

`SI[`*expr*`]`	converte *expr* nel sistema SI (International System)
`MKS[`*expr*`]`	converte *expr* nel sistema MKS (meter/kilogram/second)
`CGS[`*expr*`]`	converte *expr* nel sistema CGS (centimeter/gram/second)

Per esempio:

In[311]:=
```
SI[13 Mile]
```

Out[311]=
$$\frac{2615184\ \text{Meter}}{125}$$

Qua sotto sono rappresentate le tabelle delle varie unità di misuta incluse in questo package:

Prefissi

`Yocto`	10^{-24}	`Deca`	10^{1}
`Zepto`	10^{-21}	`Hecto`	10^{2}
`Atto`	10^{-18}	`Kilo`	10^{3}
`Femto`	10^{-15}	`Mega`	10^{6}
`Pico`	10^{-12}	`Giga`	10^{9}
`Nano`	10^{-9}	`Tera`	10^{12}
`Micro`	10^{-6}	`Peta`	10^{15}
`Milli`	10^{-3}	`Exa`	10^{18}
`Centi`	10^{-2}	`Zetta`	10^{21}
`Deci`	10^{-1}	`Yotta`	10^{24}

Unità elettriche

Abampere	Abcoulomb
Abfarad	Abhenry
Abmho	Abohm
Abvolt	Amp
Biot	Coulomb
Farad	Gilbert
Henry	Mho
Ohm	Siemens
Statampere	Statcoulomb
Statfarad	Stathenry
Statohm	Statvolt
Volt	

Unità di lunghezza:

AU	Bolt
Cable	Caliber
Centimeter	Chain
Cicero	Cubit
Didot	DidotPoint
Ell	Fathom
Feet	Fermi
Foot	Furlong
Hand	Inch
League	LightYear
Link	Meter
Micron	Mil
Mile	NauticalMile
Parsec	Perch
Pica	Point
Pole	PrintersPoint
Rod	Rope
Skein	Span
Stadion	Stadium
StatuteMile	SurveyMile
XUnit	Yard

Unità di informazione:

Baud	Bit
Byte	Nibble

Unità di tempo:

Century	Day
Decade	Fortnight
Hour	Millennium
Minute	Month
Second	SiderealSecond
SiderealYear	TropicalYear
Week	Year

Unità di massa:

AMU	AtomicMassUnit
Dalton	Geepound
Gram	Kilogram
MetricTon	Quintal
Slug	SolarMass
Tonne	

Unità di peso:

AssayTon	AvoirdupoisOunce
AvoirdupoisPound	Bale
Carat	Cental
Drachma	Grain
GrossHundredweight	Hundredweight
Libra	Mina
NetHundredweight	Obolos
Ounce	Pennyweight
Pondus	Pound
Shekel	ShortHundredweight
ShortTon	Stone
Talent	Ton
TroyOunce	Wey

Unità di forza:

Dyne	GramWeight
KilogramForce	KilogramWeight
Newton	Poundal
PoundForce	PoundWeight
TonForce	

Unità di lunghezza inversa:

Diopter	Kayser

Unità di volume:

Bag	Barrel
BoardFoot	Bucket
Bushel	Butt
Cord	Cup
Drop	Ephah
Fifth	Firkin
FluidDram	FluidOunce
Gallon	Gill
Hogshead	Jeroboam
Jigger	Last
Liter	Magnum
Minim	Noggin
Omer	Pint
Pony	Puncheon
Quart	RegisterTon
Seam	Shot
Stere	Tablespoon
Teaspoon	Tun
UKGallon	UKPint
WineBottle	

Unità di viscosità:

Poise	Reyn
Rhes	Stokes

Unità di energia luminosa:

Apostilb	Candela
Candle	FootCandle
Hefner	Lambert
Lumen	Lumerg
Lux	Nit
Phot	Stilb
Talbot	

Unità di radiazione:

Becquerel	Curie
GrayDose	Rad
Roentgen	Rontgen
Rutherford	

Unità di angolo:

ArcMinute	ArcSecond
Circle	Degree
Grade	Quadrant
Radian	RightAngle
Steradian	

Unità di potenza:

ChevalVapeur	Horsepower
Watt	

Unità di area:

Acre	Are
Barn	Hectare
Rood	Section
Township	

Unità di quantità di sostanza:

Dozen	Gross
Mole	

Unità di accelerazione di gravità:

Gal	Gravity

Unità di forza magnetica:

BohrMagneton	Gauss
Gamma	Maxwell
NuclearMagneton	Oersted
Tesla	Weber

Moltiplicatori di unità:

ArcSecond	BakersDozen
Circle	Degree
Dozen	Grade
Gross	Percent
Quadrant	RightAngle

Unità di pressione:

Atmosphere	Bar
Barye	InchMercury
MillimeterMercury	Pascal
Torr	

Unità di energia:

BritishThermalUnit	BTU
Calorie	ElectronVolt
Erg	Joule
Rydberg	Therm

Unità di frequenza:

Hertz

Come vedete, ce n'è davvero per tutti i gusti...

■ NumberTheory`Recognize`

Questo package permette di definire dei polinomi, a partire dalle loro radici conosciute:

`Recognize[x, n, t]`	crea un polinomio di grado massimo *n* nella variabile *t* in modo che *x* sia uno zero approssimato del polinomio
`Recognize[x, n, t, k]`	ficrea un polinomio di grado massimo *n* nella variabile *t* in modo che *x* sia uno zero approssimato del polinomio, e con un coefficiente di penalità *k* per i polinomi di grado superiore

Supponiamo di volere una retta che passi per un determinato punto, per esempio 3.45:

```
In[312]:= << NumberTheory`Recognize`

In[313]:= Recognize[3.45, 1, x]

Out[313]= 69 - 20 x
```

In genere, se trova un polinomio soddisfacente per gradi inferiori, allora si blocca, evitando il calcolo di polinomi di grado superiore:

```
In[314]:= Recognize[3.45, 4, x]

Out[314]= 69 - 20 x
```

Questo accade perchè il package crea dei polinomi con coefficienti interi, e non reali. Di conseguenza vengono soddisfatte le condizioni già per un polinomio di tal grado... Possiamo avere soluzioni per la quale questo non è valido. Per esempio:

```
In[315]:= sol = 1.462624652;
```

In questo caso ho:

```
In[316]:= Recognize[sol, 1, x]
```

$$\text{Out[316]= } -44723006 + 30577227\,x$$

Tuttavia, aumentando il grado, possiamo avere polinomi con coefficienti più piccoli:

```
In[317]:= Recognize[sol, 4, x]
```

$$\text{Out[317]= } -512 - 209\,x - 370\,x^2 - 511\,x^3 + 701\,x^4$$

Che è già più fattibile come numeri:

```
In[318]:= sol = N[Sqrt[3]];
```

```
In[319]:= Recognize[sol, 1, x]
```

$$\text{Out[319]= } -50843527 + 29354524\,x$$

```
In[320]:= Recognize[sol, 2, x]
```

$$\text{Out[320]= } -3 + x^2$$

```
In[321]:= Recognize[sol, 3, x]
```

$$\text{Out[321]= } -3 + x^2$$

Come vedete, anche in questo caso la soluzione più efficace è quella riguardante il polinomio di secondo grado: d'altronde, era scontato, dato che la nostra soluzione era una radice quadrata...

Se vogliamo comunque evitare di scrivere equazioni di grado troppo elevato, possiao utilizzare un coefficiente che limita l'uso dei polinomi di grado massimo:

```
In[322]:= sol = N[4^(1 / 7)]
```

```
Out[322]= 1.21901
```

```
In[323]:= Recognize[sol, 10, x]
```

$$\text{Out[323]= } -4 + 8\,x - 4\,x^2 + x^7 - 2\,x^8 + x^9$$

In[324]:=
```
Roots[% == 0, x] // N
```

Out[324]=
```
x == 0.760043 + 0.953063 i ||
 x == -0.271256 + 1.18845 i || x == -1.09829 + 0.52891 i ||
 x == -1.09829 - 0.52891 i || x == -0.271256 - 1.18845 i ||
 x == 0.760043 - 0.953063 i || x == 1.21901 || x == 1. || x == 1.
```

Come vedete, abbiamo le soluzioni contengono quella iniziale. Se vogliamo limitare il grado, possiamo utilizzare il coefficiente di peso:

In[325]:=
```
Recognize[sol, 10, x, 7]
```

Out[325]=
$$-1952 + 1927\,x - 327\,x^2 + 32\,x^3 + 14\,x^4$$

In[326]:=
```
Roots[% == 0, x] // N
```

Out[326]=
```
x == 2.15641 - 3.15924 i ||
 x == 2.15641 + 3.15924 i || x == -7.81756 || x == 1.21901
```

In questo caso abbiamo detto a *Mathematica* che, se la differenza di soluzione fra un grado e quello inferiore non è così grande (così grande dipende dal coefficiente), allora ci accontentiamo della soluzione del grado inferiore, anche se non è precisa come quella di grado superiore. Notate come le soluzioni paiano uguali, ma tenete sempre conto che il dato non viene rappresentato, di default, con tutte le cifre decimali, e che *Mathematica* lavora comunque, volendo, con precisione arbitraria, per cui l'approssimazione si fa sentire man mano che aumentiamo la precisione. Bisogna quindi scegliere il giusto compromesso fra precisione e semplicità del polinomio.