Grafica e suoni

Introduzione

Abbiamo visto, fino ad adesso, come si possono elaborare funzioni, risolvere equazioni e così via. Tuttavia, molte volte è necessario trovare il modo di poter visualizzare tramite grafici ed immagini i risultati così ottenuti. Per esempio, se si crea un filtro per un'immagine, viene naturale, invece di andare a guardare liste interminabili di numeri di cui non capite niente, fare direttamente un confronto fra l'immagine iniziale e quella finale, oppure rappresentare graficamente l'andamento di una funzione, o visualizzare un campo vettoriale.

Mathematica in questo campo ha numerose funzioni, che permettono di visualizzare praticamente ogni cosa voi vogliate. Ci sono comandi specifici per grafici a due e tre dimensioni, ed anche per la tipologia di grafico.

Come ultima nota, vedrete che questo capitolo sarà uno dei più lunghi (se non il più lungo). Non spaventatevi di ciò, e dal numero di pagine, perchè il loro elevato numero è dovuto al fatto che saranno infarcite di numerose immagini. Con ciò, naturalmente, non voglio dire che starete per leggere un fumetto... Di carne al fuoco ce n'è tanta...

Grafica

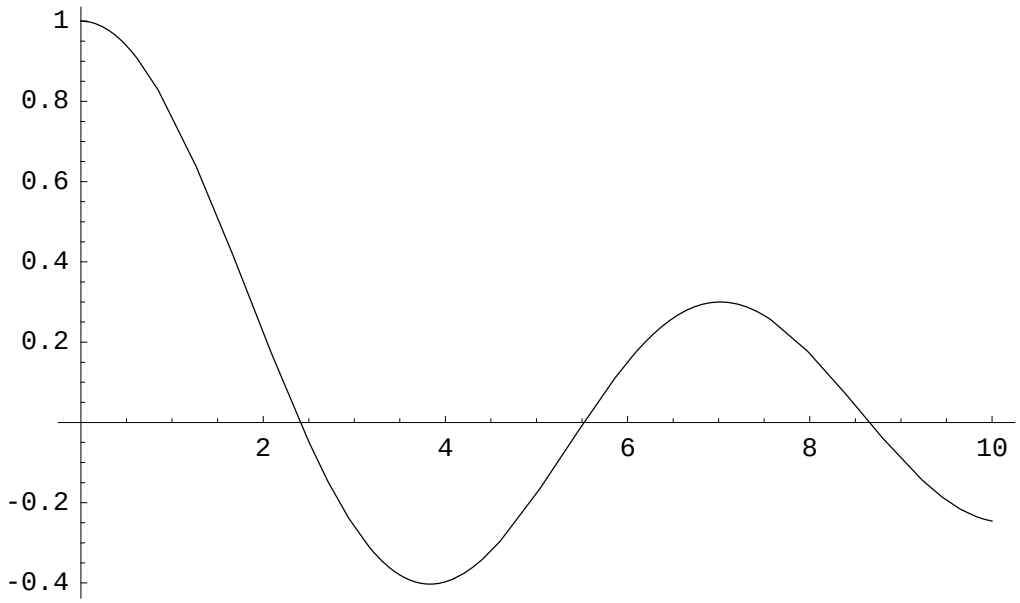
Funzioni ad una variabile

Il comando principale con cui si disegnano le funzioni (fra cui quasi tutte quelle che avete visto finora) è Plot:

`Plot[`f`, {`x`, `x_{min}`, `x_{max}`}]`	disegna f nella variabile x da x_{min} a x_{max}
`Plot[{`f_1`, `f_2`, … }, {`x`, `x_{min}`, `x_{max}`}]`	disegna assieme più funzioni

Il comando ha moltissime opzioni che permettono di personalizzare il grafico, ma per ora ci basta sapere questo. Vediamo subito un esempio di come si disegna una funzione:

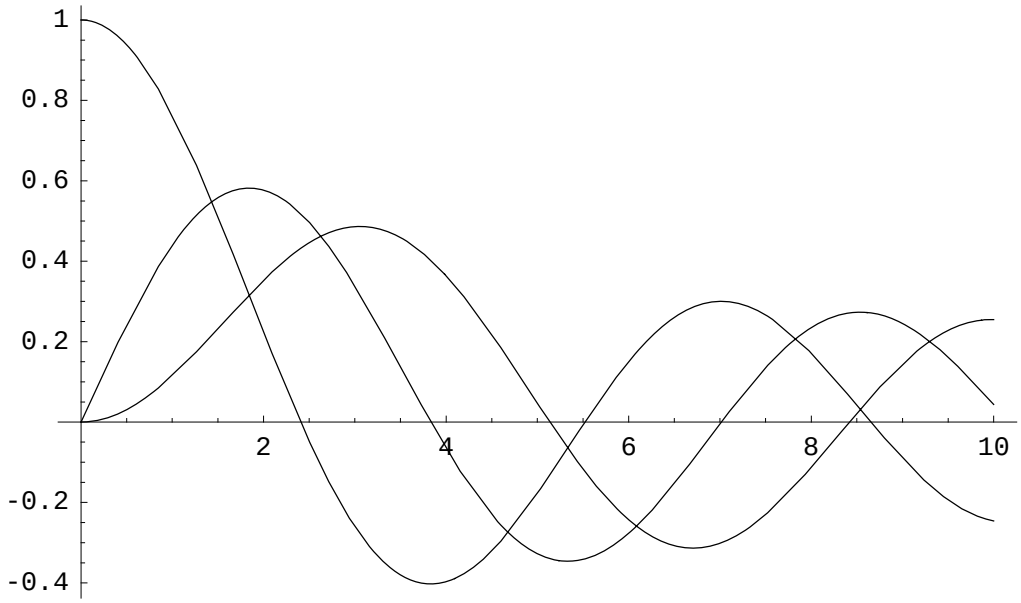
In[1]:=
```
Plot[BesselJ[0, x], {x, 0, 10}];
```



Vediamo come il grafico sia automaticamente scalato nell'asse *y*.

Possiamo anche visualizzare più funzioni nello stesso grafico:

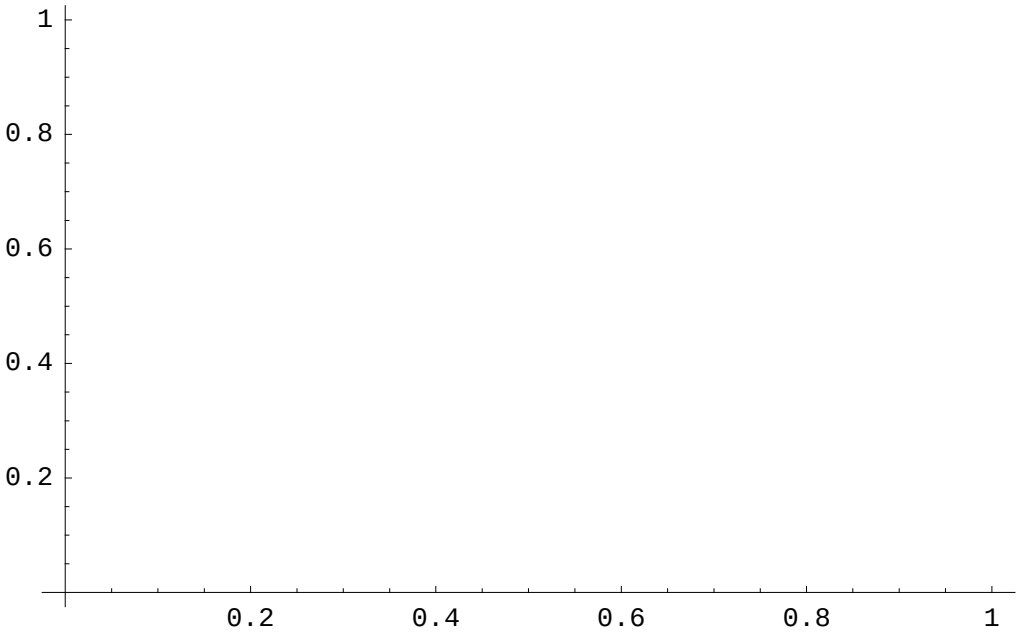
In[2]:=
```
Plot[{BesselJ[0, x], BesselJ[1, x], BesselJ[2, x]}, {x, 0, 10}];
```



Vediamo come le funzioni appaiano sovrapposte fra di loro. Naturalmente, in questo caso, dobbiamo specificare lo stesso dominio per ciascuna funzione, e quindi avremo bisogno di una sola variabile x per tutte e tre le funizioni.

Nel caso si voglia realizzare il grafico di una lista di funzioni definita con Table, le cose cambiano leggermente: supponiamo, infatti, di avere:

```
In[3]:= lista = Table[ChebyshevT[n, x], {n, 8}];

In[4]:= Plot[lista, {x, -1, 1}]

  — Plot::plnr : lista is not a machine-size real number at x = -1.. More…

  — Plot::plnr : lista is not a machine-size real number at x = -0.918866. More…

  — Plot::plnr : lista is not a machine-size real number at x = -0.830382. More…

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



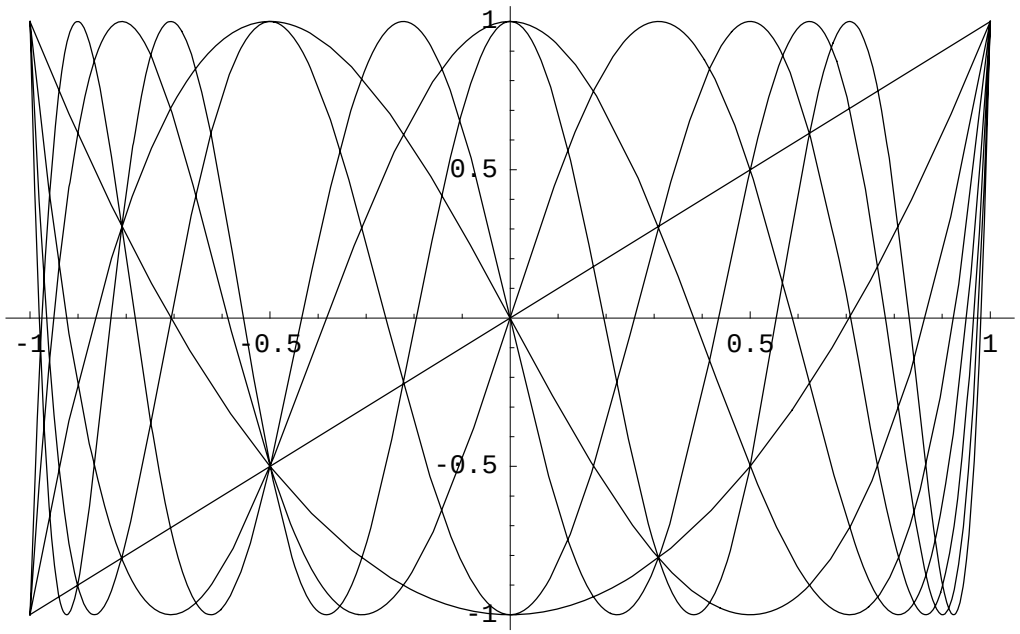

```
Out[4]= - Graphics -
```

In questo caso possiamo vedere che non possiamo visualizzare le funzioni, restituendo degli errori ed un grafico vuoto. Questo perchè, in questa notazione, prima vengono valutati i valori della x per generare il grafico, e dopo viene sostituita alle funzioni: in questo caso, invece è necessario prima valutare la tavola di valori, e poi andare a sostituire i valori di x. Questo si fa con il comando Evaluate:

`Plot[`*f*`, {`*x*`,` x_{min}`,` x_{max}`}]`	*prima* specifica precisi valori di *x*, dopo valuta *f* per ogni valore di *x*
`Plot[Evaluate[`*f*`], {`*x*`,` x_{min}`,` x_{max}`}]`	*prima* valuta *f*, sceglie specifici valori numerici *x*
`Plot[Evaluate[Table[`*f*`, … ]], {`*x*`,` x_{min}`,` x_{max}`}]`	genera una lista di funzioni, e poi li disegna
`Plot[Evaluate[`*y*`[`*x*`] /. solution], {`*x*`,` x_{min}`,`	plotta un'equazione differenziale numerica ottenuta tramite `NDSolve`

Riproviamo ad effettuare, adesso, il plottaggio precedente, ma stavolta con Evaluate:

In[5]:=
```
Plot[Evaluate[lista], {x, -1, 1}]
```



Out[5]=
```
- Graphics -
```

Vediamo come adesso, avendo valutato prima la tavola di funzioni, questa adesso sia correttamente riconosciuta, permettendoci di disegnare i grafici di tutte le funzioni definite. Nel primo caso si mantiene la valutazione simbolica di lista, quindi Table non veniva sviluppata, e di conseguenza *Mathematica* non riconosceva le funzioni, impedendo di poter disegnare i grafici.

Quello che dovete ricordare, quindi, è che quando create una serie di funzioni con Table, dovete usare il comando Evaluate. Supponiamo di trovarci la soluzione numerica di un'equazione differenziale:

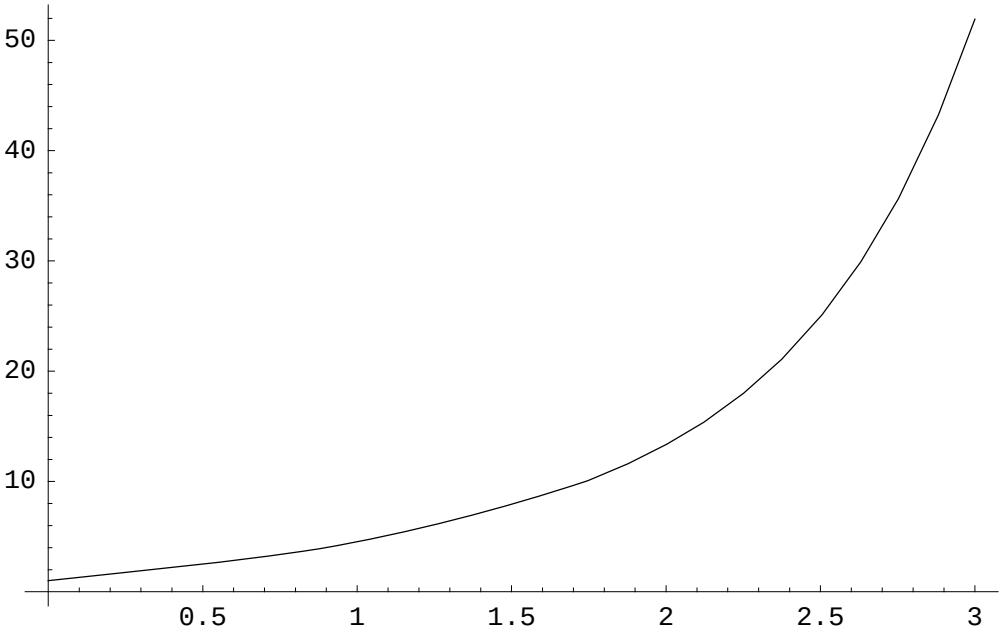
In[6]:=
```
eq = y''[x] + Sin[y'[x]] y[x] == x y[x];
```

In[7]:=
```
NDSolve[{eq, y[0] == 1, y'[0] == 3}, y, {x, 0, 4}]
```

Out[7]=
```
{{y → InterpolatingFunction[{{0., 4.}}, <>]}}
```

Abbiamo visto che la soluzione viene data in forma di funzione pura, quindi, per quello che serve a noi, bisogna prima valutare la funzione sostituita, e poi andare a graficarla:

In[8]:=
```
Plot[Evaluate[y[x] /. %], {x, 0, 3}]
```



Out[8]=
```
- Graphics -
```

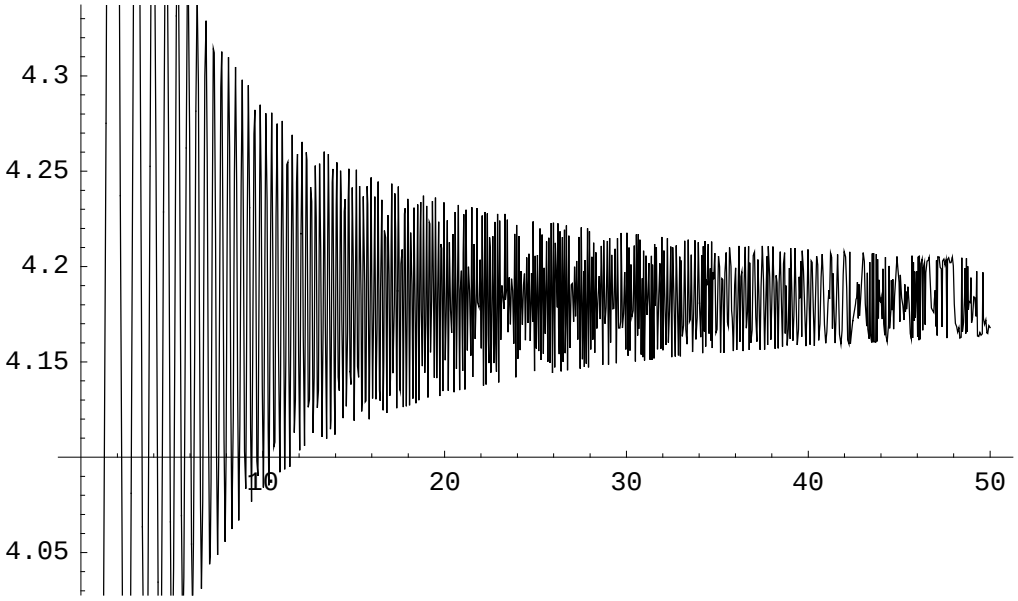
In realtà, avremmo ottenuto lo stesso risultato anche senza Evaluate, ma il tempo impiegato sarebbe diverso;

In[9]:=
```
eq2 = y'[x] / Sqrt[y[x]] == Sin[x^2];
```

In[10]:=
```
NDSolve[{eq2, y[0] == 3}, y, {x, 0, 50}]
```

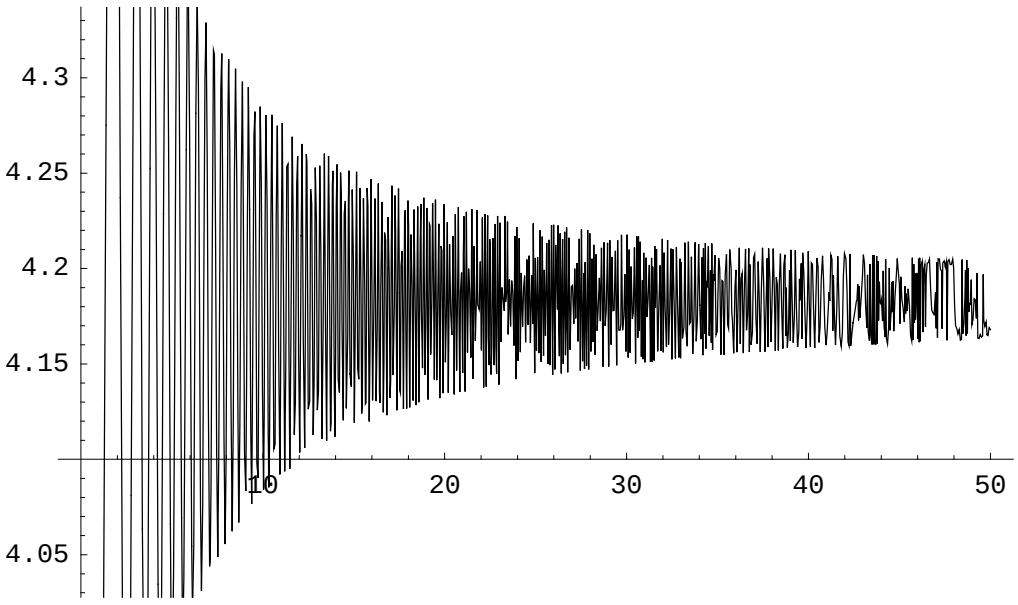
Out[10]=
```
{{y → InterpolatingFunction[{{0., 50.}}, <>]}}
```

In[11]:=
```
Timing[Plot[Evaluate[y[x] /. %], {x, 0, 50}]]
```



Out[11]= {0.016 Second, - Graphics -}

In[12]:=
```
Timing[Plot[y[x] /. %%, {x, 0, 50}]]
```



Out[12]= {0.015 Second, - Graphics -}

Come potete vedere, il risultato grafico è lo stesso. Tuttavia il tempo impiegato per eseguire la funzione è diverso; con Timing, infatti, si calcola il tempo macchina per effettuare l'operazione che ha come argomento, e nel secondo caso abbiamo impiegato un tempo maggiore (sempre poco, effettivamente).

La differenza, in questo caso, sta in questo: nel primo grafico, usando Evaluate, prima abbiamo valutato la funzione, e quindi abbiamo ottenuto la funzione interpolata, e dopo siamo andati a sostituire i valori di *x* alla funzione, andando a trovare i valori e quindi a graficarli. Nel secondo caso, invece, prima andiamo a calcolarci il valore di x in cui valutare la funzione, e poi andiamo a valutare la funzione in quel punto; allora, dato che nella funzione c'è una regola di sostituzione, invece di applicarla solamente una volta, come nel primo caso, la applichiamo per ogni punto x che calcoliamo. andando ad eseguire quindi un tot numero di sostituzioni identiche, una per ogni punto che *Mathematica* usa per creare il grafico. Per funzioni complicate e con molti punti, questo può rallentare notevolmente il processo, specialmente su computer lenti.

Opzioni di disegno

Abbiamo visto come disegnare le funzioni. Tuttavia, a volte è utile poter personalizzare le opzioni di visualizzazione dei grafici, per modificarne lo stile, oppure per evidenziarne alcune caratteristiche particolari. Se andiamo a vedere le opzioni disponibili per il comandio Plot, vediamo che ce ne sono parecchie:

In[13]:=
```
Options[Plot]
```

Out[13]=
```
{AspectRatio → 1/GoldenRatio, Axes → Automatic, AxesLabel → None,
 AxesOrigin → Automatic, AxesStyle → Automatic, Background → Automatic,
 ColorOutput → Automatic, Compiled → True, DefaultColor → Automatic,
 DefaultFont :→ $DefaultFont, DisplayFunction :→ $DisplayFunction,
 Epilog → {}, FormatType :→ $FormatType, Frame → False, FrameLabel → None,
 FrameStyle → Automatic, FrameTicks → Automatic, GridLines → None,
 ImageSize → Automatic, MaxBend → 10., PlotDivision → 30.,
 PlotLabel → None, PlotPoints → 25, PlotRange → Automatic,
 PlotRegion → Automatic, PlotStyle → Automatic, Prolog → {},
 RotateLabel → True, TextStyle :→ $TextStyle, Ticks → Automatic}
```

Alcune sono utili, altre invece non verranno usate quasi mai: vediamo il loro significato:

- **Axes**: definisce se dobbiamo disegnare o no gli assi dei grafici. Se lo impostiamo su False gli assi non verranno disegnati.

- **AxesLabel**: imposta il titolo delle etichette da imporre agli assi cartesiani, in modo da visualizzarli sul grafico; si usa principalmente per evidenziare la grandezza che corrisponde all'ascissa o all'ordinata.

- **AspectRatio**: rappresenta il rapporto fra larghezza ed altezza del grafico: possiamo imporre un valore numerico

- **AxesOrigin**: definisce il centro in cui verranno rappresentati gli assi cartesiani. Di default è impostato su Automatic, che definisce quindi il centro degli assi (0,0), se questo è visibile nel

grafico, altrimeni viene impostato in modo che gli assi si trovino nei margini del grafico. Possiamo comunque impostarlo manualmente per avere l'origine, per esempio, nel massimo di una funzione, conoscendo il punto.

- **AxesStyle**: con questa opzione possiamo imporre lo stile degli assi, per esempio modificandone il colore, oppure lo spessore delle linee.

- **Background**: imposta il tipo di sfondo del grafico, impostando il colore, oppure, mediante scrittura più avanzata, magari un'immagine oppure un gradiente (anche se non ci sono mai riuscito a fare uno sfondo che non sia uniforme, ancora...).

- **ColorOutput**: questa opzione specifica il tipo di output di colore che viene utilizzato per una funzione. Permette, per esempio, di disegnare un gradico in scals di grigi invece che a colori, oppure in colori RGB.

- **Compiled**: specifica se una funzione deve essere compilata dal motore interno di *Mathematica* prima di poter calcolare i valori della funzione per il plottaggio: di solito si imposta di default su True, ma può essere impostato su False per alcuni tipi di grafico, se per esempio è necessaria un'alta precisione numerica, il che a volte con le funzioni compilate possono dare problemi. Credo sia l'opzione che ho usato di meno fra tutte...

- **DefaultColor**: possiamo definire, con questa opzione, il colore standard per disegnare tutto quanto il grafico, nella sua totalità, e non, per esempio, solo il colore della funzione.

- **DefaultFont**: definisce il tipo di font che deve essere usato per visualizzare l'eventuale testo presente nel grafico.

- **DisplayFunction**: non è che abbia mai capito cosa faccia esattamente questa opzione: permette di definire come visualizzare il grafico. Se settato su Identity non è visualizzato... Mah!!!

- **Epilog**: definisce una lista do oggetti grafici (quali punti e cerchi, per esempio), che devono essere aggiunti al grafico dopo che è stata disegnata la funzione principale. Si può usare, per esempio, per visualizzare ed evidenziale punti e zone del grafico.

- **FormatType**: permette di definire lo stile della cella. Per esempio, se scriviamo una formula come titolo del grafico, possiamo, mediante questa opzione, visualizzarla in notazione standard, invece che come la visualizza *Mathematica* sotto forma di testo.

- **Frame**: questa opzione, se settata su True, disegna una cornice intorno al grafico. Mediante le seguenti, opzioni, dopo, possiamo definire lo stile della cornice.

- **FrameLabel**: definisce le etichette da applicare alla cornice. se la impostiamo come {titolo1, titolo2}, appariranno rispettivamente sotto ed a sinistra, mentre se lo impostiamo a {titolo1, titolo2, titolo3, titolo4}, compariranno a partire dal basso, in senso orario.

- ⋆ **FrameStyle**: personalizza lo stile con cui viene disegnata la cornice, definendo colore, stile e spessore delle linee che compongono il frame: ogni lato può avere la sua personalizzazione{{personalizzax},{personalizzay}}.

- ⋆ **FrameTicks**: definisce in questo senso se devono comparire oppure no i marker degli assi nella cornice, definendo anche quali valori devono essere segnati.

- ⋆ **GridLines**: specifica se e come visualizzare la griglia nel grafico. Impostato su Automatic permette a *Mathematica* di creare la griglia automaticamente, altrimenti si possono definire separatamente il tipo di griglia per i due assi, definendo una lista del tipo {{valori x},{valori y}}, in cui compaiono i valori dell'asse x e dell'asse y in cui vogliamo che compaiano le linee della griglia. una di queste iste può essere sostituita da Automatic, se ci interessa solamente un asse e vogliamo che l'altro sia definito automaticamente.

- ⋆ **ImageSize**: specifica le dimensioni in cui sono deve essere visualizzata l'immagine. Si può specificare la larghezza, oppure larghezza ed altezza.

- ⋆ **MaxBend**: definise il massimo angolo di piegatura dell'angolo fra due segmenti. Questo perchè il comando usa un algoritmo adattivo per poter disegnare il grafico, che avvicina i punti a seconda di quest'angolo, in modo da avere una curva risultante smussata e non con punti spigolosi, entro i limiti di PlotDivision. Aumentarlo a volte può servire per visualizzare per intero delle funzioni che *Mathematica*, senza impostare i limiti degli assi, non visualizza interamente all'interno del grafico.

- ⋆ **PlotDivision**: specifica l'ammontare massino di divisione che devono essere impiegate per visualizzare una curva. più alto sarà questo valore, più smussata risulterà la curva, anche se aumenterà il tempo di calcolo. Da aumentare se si vede che la curva è formata da 'spezzate', che indica una divisione non sufficiente.

- ⋆ **PlotLabel**: aggiunge il titolo specificato al grafico, mettendolo sopra il disegno della funzione.

- ⋆ **PlotPoints**: definisce il numero di punti iniziali da usare per disegnare la funzione. E' solo iniziale, perchè l'algoritmo è adattivo, andando a modificarlo in funzione della curvatura della funzione. Come vedremo più avanti, invece, per il grafici a due variabili sarà più importante.

- ⋆ **PlotRange**: definisce l'area di visualizzazione del grafico: settato su All include tutti i punti calcolati, su Automatic aggiusta eliminando i punti troppo estremi, oppure può essere una lista pari a {minimo, massimo} se si vuole limitare solo l'asse y, oppure {{minimox, massimox},{minimoy, massimoy}} se si vuole specificare interamente l'area.

- ⋆ **PlotRegion**: specifica quale area dell'immagine generata dalla funzione plot deve essere riempita con il grafico, lasciando vuoto il resto.

- ⋆ **PlotStyle**: definisce lo stile con cui disegnare il grafico. Se si disegna una sola funzione, è possibile definire l'unico stile, mentre, se si disegnano più funzioni, è possibile decidere se definire lo stile per tutte le funzioni, oppure gestirle separatamente, tramite le liste.

- ⋆ **Prolog**: come Epilog, con la differenza che le forme descritte in Prolog vengono disegnate prima della funzione, e quindi appariranno sotto, invece che sopra.

- ⋆ **RotateLabel**: definisce se l'etichetta dell'asse y deve essere ruotata in modo da essere parallela all'asse, con True, oppure, con False, se deve essere mantenuta orizzontale.

- ⋆ **TextStyle**: specifica lo stile e le opzioni del font del testo che verrà visualizzato.

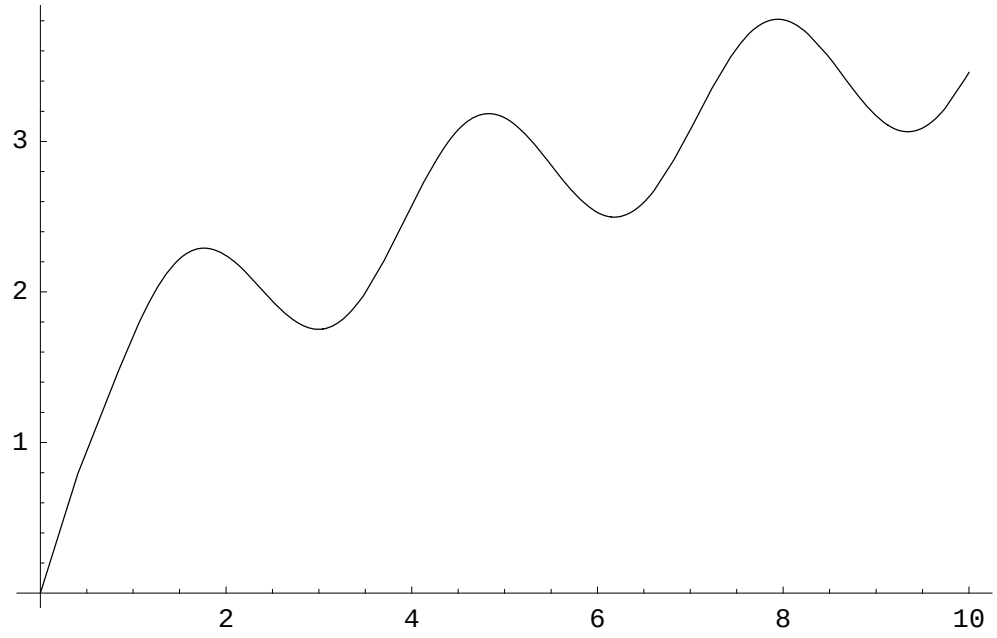
- ⋆ **Ticks**: definisce i marker che devono essere visualizzati negli assi della funzione, se quello usati di default non ci vanno bene. O si definisce la lista solo per l'asse x, oppure si definisce la doppia lista se ci interessa definire entrambi gli assi.

Vediamo, adesso, alcuni esempi su come possiamo utilizzare queste opzioni: supponiamo di definire la seguente funzione:

In[14]:=
```
f[x_] := Sin[x]^2 + Sqrt[x]
```

Adesso, disegnamola tramite il comando Plot:

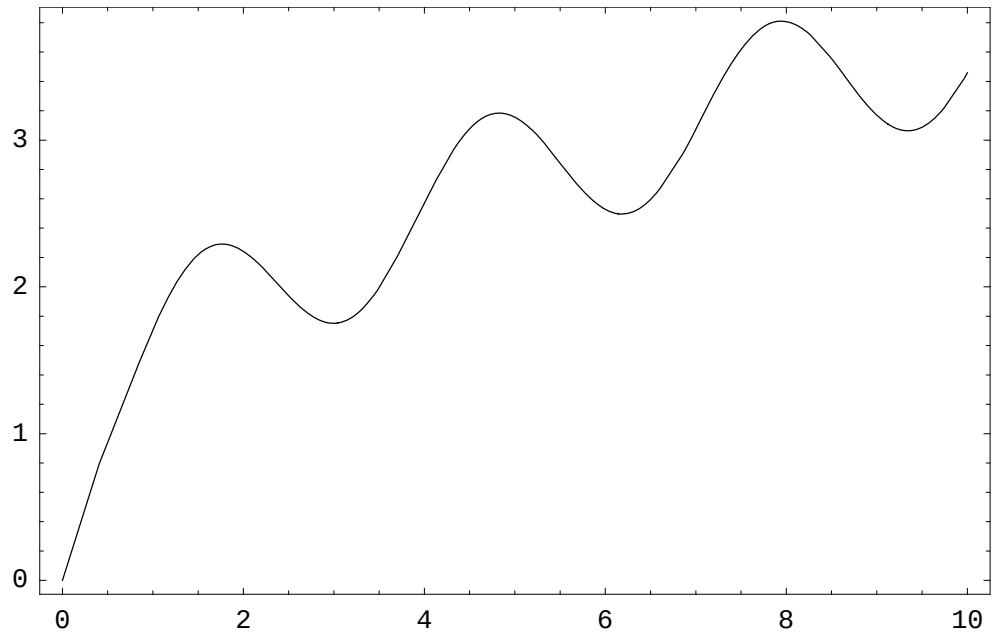
In[15]:=
```
Plot[f[x], {x, 0, 10}]
```



Out[15]=
```
- Graphics -
```

Proviamo ad aggiungere un frame:

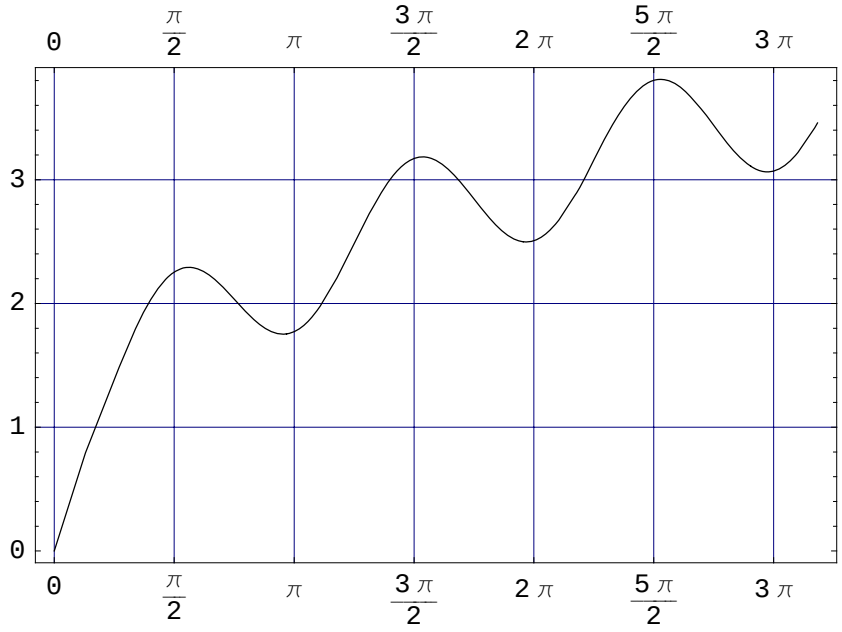
In[16]:=
```
Plot[f[x], {x, 0, 10}, Frame → True]
```



Out[16]= - Graphics -

Visualizziamo, adesso, la griglia, tarata in multipli di π:

In[17]:=
```
Plot[f[x], {x, 0, 10}, Frame → True,
 GridLines → {{0, π / 2, π, 3 / 2 π, 2 π, 5 / 2 π, 3 π}, Automatic},
 FrameTicks -> {{0, π / 2, π, 3 / 2 π, 2 π, 5 / 2 π, 3 π}, Automatic}]
```



Out[17]= - Graphics -

Decidiamo, adesso, di andarci a calcolare il primo massimo della funzione, e visualizzarlo:

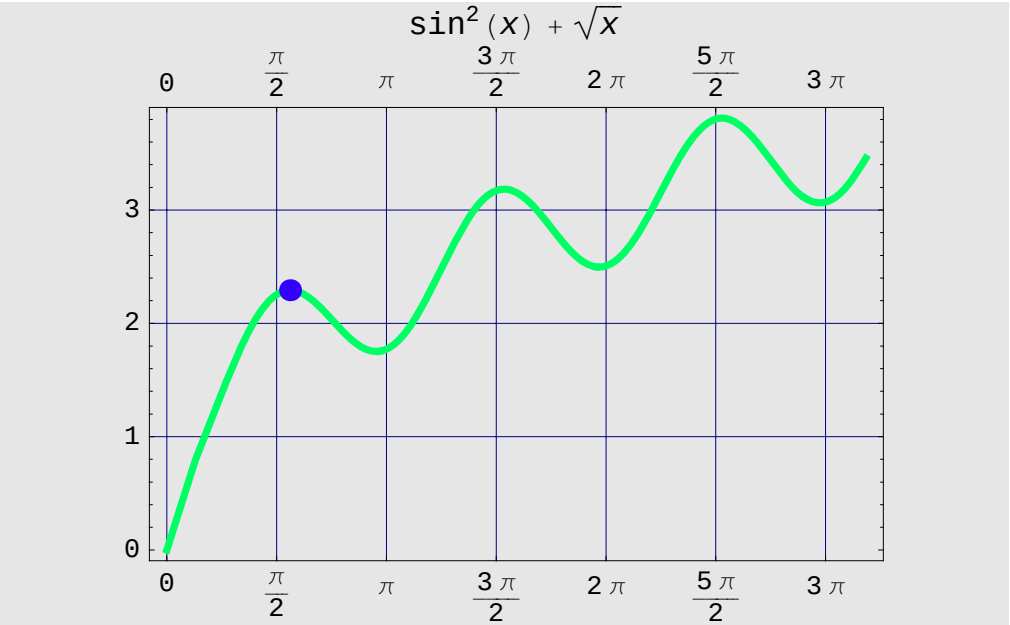
```
In[18]:= FindMaximum[f[x], {x, 1}]

Out[18]= {2.29129, {x → 1.7638}}
```

```
In[19]:= punto = {x, %[[1]]} /. %[[2]]

Out[19]= {1.7638, 2.29129}
```

Notate come ho dovuto giocare un secondo per scrivere il risultato di FindMaximum nella forma che mi serviva, dato che dava prima l'ordinata, e dopo la regola per la sostituzione dell'ascissa. Comunque, se avete seguito la lezioncina sulle liste che abbiamo affrontato prima, non dovreste avere problemi a capire quello che ho fatto.

Avendo adesso trovato questo punto, posso andare a disegnarlo, completando il disegno della funzione per come me l'ero prefissato, aggiungendo qualche altra cosuccia:

```
In[20]:= Plot[f[x], {x, 0, 10},
          Frame → True,
          GridLines → {{0, π / 2, π, 3 / 2 π, 2 π, 5 / 2 π, 3 π}, Automatic},
          FrameTicks -> {{0, π / 2, π, 3 / 2 π, 2 π, 5 / 2 π, 3 π}, Automatic},
          Epilog → {PointSize[0.03], Hue[0.7], Point[punto]},
          PlotLabel →
           StyleForm[TraditionalForm[Sin[x]^2 + Sqrt[x]], FontSize → 12],
          Background → GrayLevel[0.9],
          PlotStyle → {Hue[0.4], Thickness[0.01]}]
```




```
Out[20]= - Graphics -
```

Abbiamo scritto un po' di codice per disegnare questa funzione con questa, chiamiamole, 'formattazione', per cui analizziamola: prima di tutto, abbiamo visualizzato il frame, e dopo le linee della griglia, definendole dove volevamo noi, cioè per punti notevoli trigonomentrici, per l'asse x; dopo, abbiamo definito le etichette dell'asse x nel frame, per far coincidere i valori segnati con la griglia (infatti hanno gli stessi valori). Con Epilog abbiamo aggiunto un punto nelle coordinate che ci eravamo trovati prima, imponendo la dimensione ed il colore. Abbiamo imposto il titolo, scrivendolo in forma tradizionale (come le formule dei libri), ed impostando la dimensione del carattere; per finire, ho imposto un grigio chiaro per lo sfondo, e ho cambiato lo stile della funzione, cambiandone il colore e lo spessore. Notate anche come, a parità di area occupata, le ultime due immagini appaiano più piccole, perchè nello stessa area hanno dovuto trovare posto anche il titolo, per l'ultima, e le etichette, per entrambe. Quindi, dopo aver modificato l'immagine che vi interessava, magari dovete ingrandirla col mouse. Comunque, se sapete trovare il tasto ciccione per accendere il computer e conoscete il nome dell'aggeggio pacioccoso che tenete in mano per sentirvi Guglielmo Tell per puntare la freccia che avete nello schermo, allora sapete anche ridimensionare un'immagine, vero? Detto questo, andiamo a vedere un poco più a fondo la grafica bidimensionele in *Mathematica*.

2D Generale

Abbiamo visto come siamo in grado di visualizzare velocemente e personalizzare in maniera repida ed efficace il layout delle nostre rappresentazioni grafiche. Tuttavia, *Mathematica* non si ferma al mero disegno di funzioni. In effetti, è possibile praticamente disegnare ogni cosa, con i comandi giusti. Per esempio, con quale funzione possiamo andare a disegnare dei grafi di flusso, oppure delle macchine a stati? La rappresentazione di queste figure matematiche bidimensionali va al di là del semplice (ma potentissimo, ed uno dei miei preferiti) comando Plot.

Vediamo innanzitutto come viene visualizzata la grafica in *Mathematica*. Per default, ogni grafico viene visualizzato come disegno eps. Questo formato molto usato nei programmi professionali di grafica ed impaginazione, come Acrobat, perchè rappresenta le immagini in formato vettoriale, e quindi con una qualità indipendente dalla risoluzione. Una volta creata un'immagine, possiamo tranquillamente stamparla su un francobollo come su un poster formato A0, senza alcun decadimento di qualità. Tuttavia, se volete creare immagini abbastanza grandi, dovete ricordare che lo spessore delle linee rimane costante al variare delle dimensioni, quindi magari vi conviene, mediante le opzioni, creare linee un poco più spesse, se devono poter essere viste anche da lontano. Tuttavia con solo un paio di prove sarete in grado di fare quello che volete, e comunque, la personalizzazione è un aspetto specifico, che dovete provare voi.

Adesso, vediamo come sono rappresentate le figure in *Mathematica*. Il programma dispone di una quantità di primitive bidimensionali di default. Le funzioni principali di *Mathematica* sono due, per i grafici bidimensionali:

`Graphics[`*list*`]`	general two-dimensional graphics

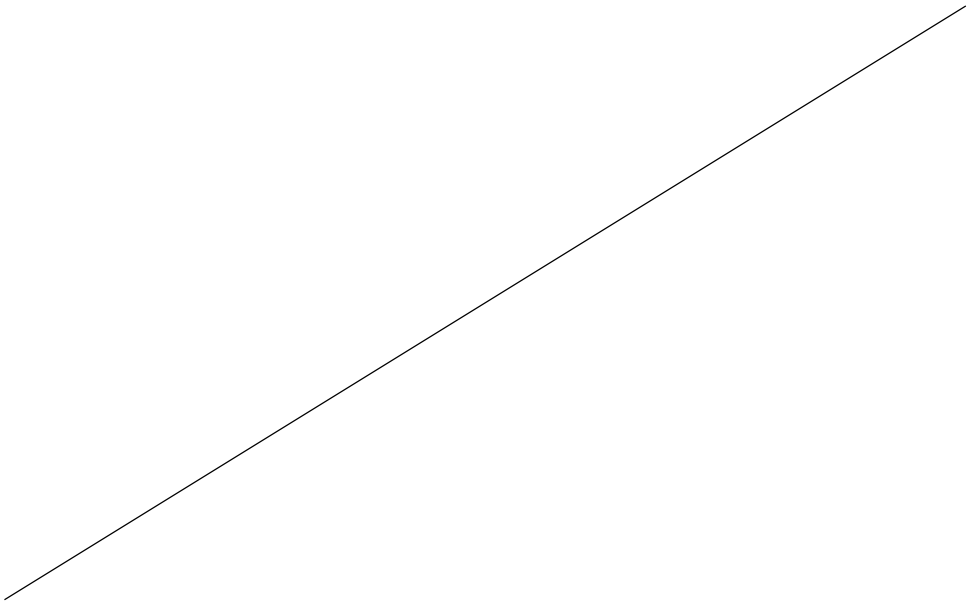
`Show[`g_1`, `g_2`, … ]`	display several graphics objects combined

Con il comando Graphics definiamo le curve e le forme che ci interessano, mentre con Show visualizziamo le figure create: se, per esempio, vogliamo creare una linea, prima dobbiamo definire la primitiva grafica che la rappresenta, e poi dobbiamo visualizzarla:

```
In[21]:= linea = Graphics[Line[{{0, 1}, {2, 4}}]]
Out[21]= - Graphics -
```

Come si può vedere, come output si da un oggetto Graphics. Per visualizzarlo, adesso, basta usare il comando Show:

```
In[22]:= Show[linea]
```



```
Out[22]= - Graphics -
```

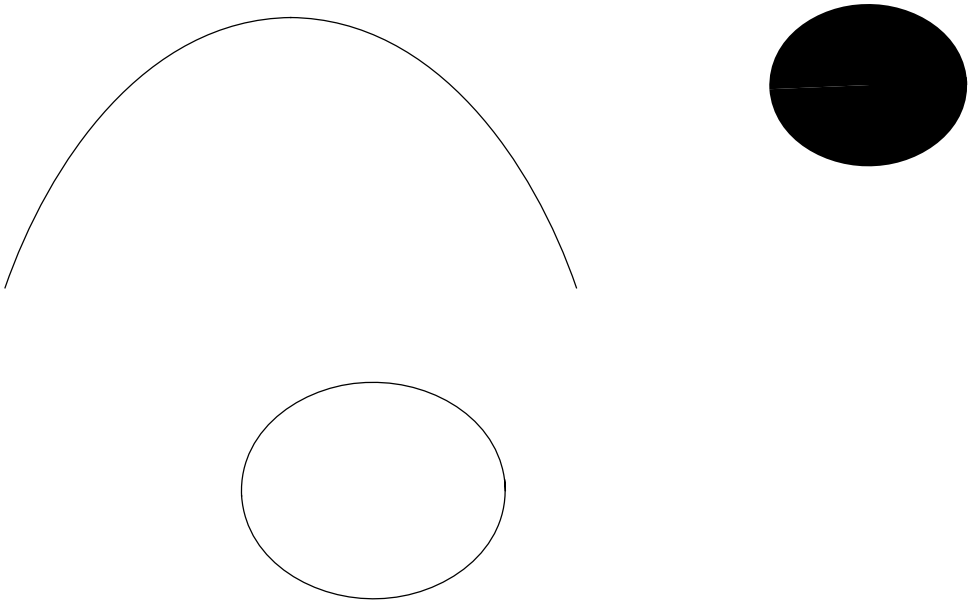
Alcune primitive che siamo in grado di disegnare sono le seguenti:

`Point[{x, y}]`	punto nella posizione *x*, *y*
`Line[{{x1, y1}, {x2, y2}, … }]`	segmento avente gli estremi nelle coordinate {x_1, y_1}, {x_2, y_2}, …
`Rectangle[{xmin, ymin}, {xmax, ymax}]`	rettangolo pieno, con angolo inferiore sinistro e superiore destro specificati
`Polygon[{{x1, y1}, {x2, y2}, … }]`	poligono pieno con le specificate coordinate per i vertici
`Circle[{x, y}, r]`	circonferenza di raggio *r* centrata in *x*, *y*
`Disk[{x, y}, r]`	cerchio di raggio *r* ccentrato *x*, *y*
`Raster[{{a11, a12, … }, {a21, … }, … }]`	array rettangolare con gli elementi $a_{i,j}$ rappresentanti un valore di scala di grigio compreso fra `0` e `1`
`Text[expr, {x, y}]`	il testo di *expr*, centrato in *x*, *y*
`Circle[{x, y}, {rx, ry}]`	ellisse con semiassi r_x e r_y
`Circle[{x, y}, r, {theta1, theta2}]`	arco circolare con gli angoli specificati
`Circle[{x, y}, {rx, ry}, {theta1, theta2}]`	arco ellittico
`Raster[array, {{xmin, ymin}, {xmax, ymax}}, {zmin,`	array di elementi di grigio fra z_{min} e z_{max} disegnato nel rettangolo fra {x_{min}, y_{min}} e {x_{max}, y_{max}}
`RasterArray[{{g11, g12, … }, {g21, … }, … }]`	array rettangolare in cui ogni cella è colorata usando le direttive grafiche g_{ij}

Possiamo, quindi, anche definire più figure assieme, nello stesso comando Show:

In[23]:=
```
cerchio = Graphics[Circle[{0, 0}, 0.8]];
arco =
  Graphics[Circle[{-0.5, -0.5}, {2, 4}, {30 Degree, 150 Degree}]];
disco = Graphics[Disk[{3, 3}, .6]];
```

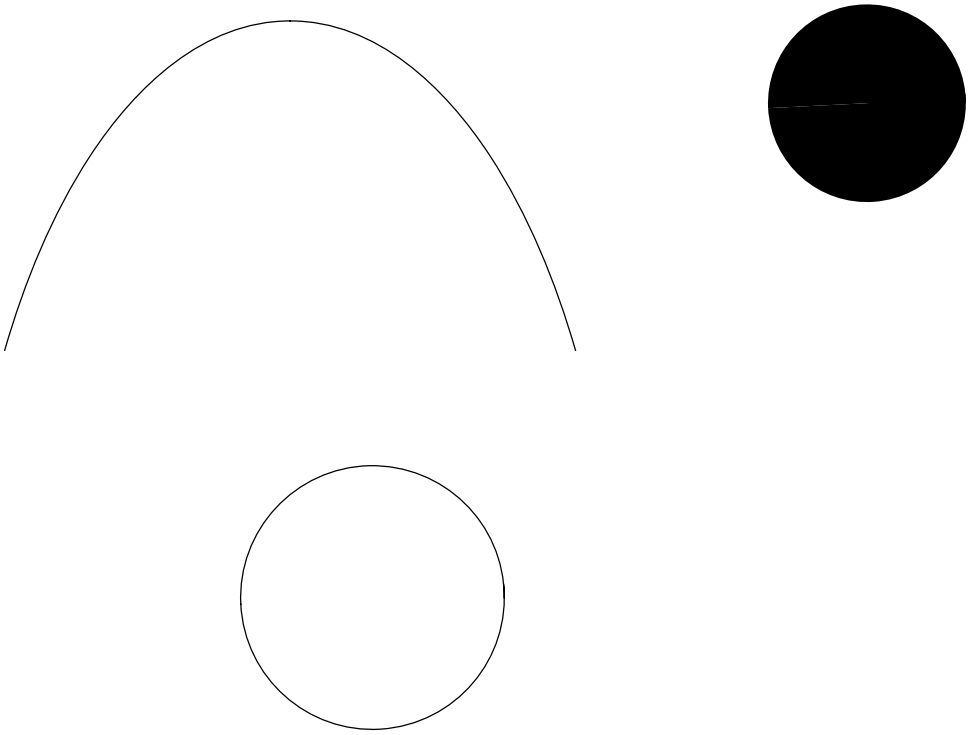
In[26]:=
```
Show[{cerchio, arco, disco}]
```



Out[26]= - Graphics -

Possiamo anche impostare le opzioni per il comando Show; per esempio, vediamo che nella figura di prima non abbiamo le stesse proporzioni per i due assi; possiamo correggere questo con l'apposita opzione:

In[27]:=
```
Show[%, AspectRatio → Automatic]
```



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

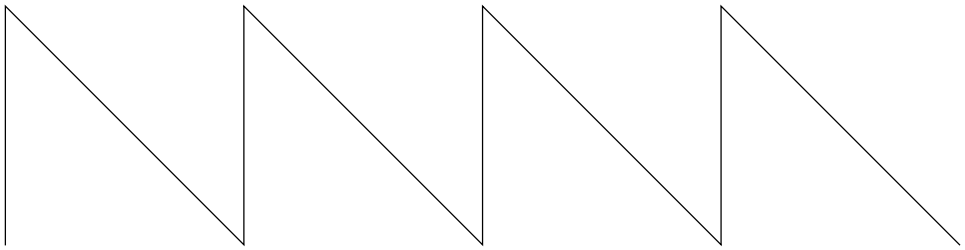
In questo modo, abbiamo impostato la stessa scala sia per le ascisse che per le ordinate.

Una volta saputo questo, e vedendo che effettivamente possiamo creare le primitive semplicemente definendo delle semplici liste di numeri, viene naturale, nelle applicazioni pratiche, usare valori dei risultati dei nostri calcoli per poter usare efficacemente queste direttive. Per esempio, possiamo mostrare semplicemente un dente di sega:

In[28]:=
```
dentedisega = Line[Table[{Mod[n, 2] + n, (-1)^n}, {n, 9}]]
```

Out[28]=
```
Line[{{2, -1}, {2, 1}, {4, -1},
  {4, 1}, {6, -1}, {6, 1}, {8, -1}, {8, 1}, {10, -1}}]
```

In[29]:=
```
Show[Graphics[dentedisega], AspectRatio → Automatic]
```



Out[29]= `- Graphics -`

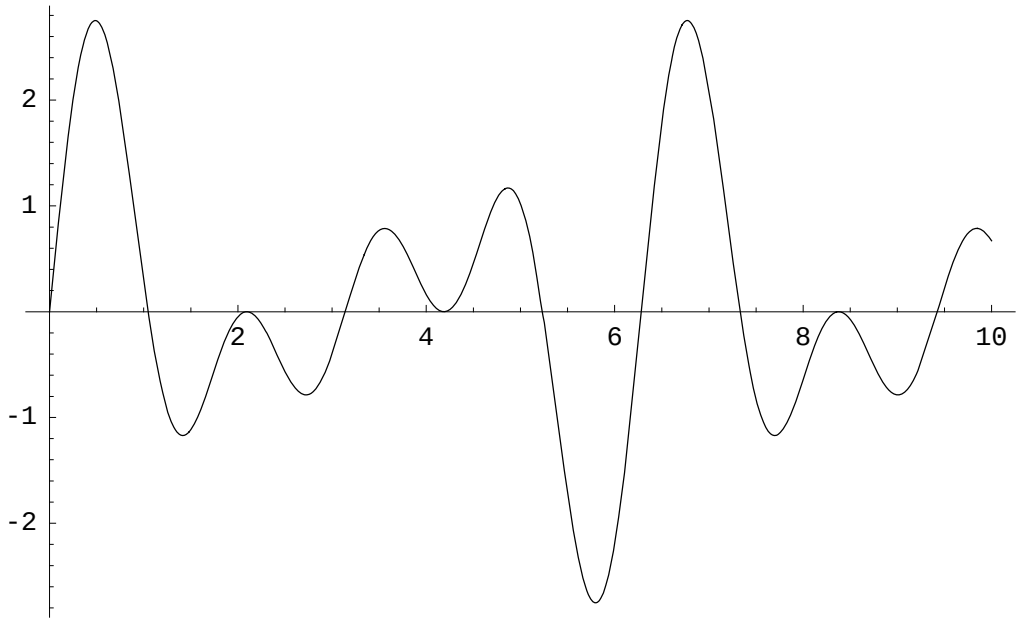
In questo esempio, abbiamo prima creato il dente di sega nel seguente modo; abbiamo creato una lista di coppie di valori con Table, in cui il primo valore, che rappresenta l'ascissa del punto, varia ogni due valori di n, usando il comando Mod, che restituisce il resto della divisione, sommato ad n. In questo caso, se per *n* il resto è 1, per *n* + 1 il resto è zero, e quindi rappresentano lo stesso valore, sommati al numero stesso. Per le ordinate, invece, si è posto alternativamente -1 ed 1. Una volta creata la lista, è stato semplice, poi, andare a visualizzare i valori, dove Line, in questo caso, contiene più valori estremi, e rappresenta a tutti gli effetti una linea spezzata, invece di invocarla per ogni segmento.

Anche il comando Plot restituisce un oggetto grafico visualizzabile tramite Show. Si può capire perchè, oltre all'immagine, entrambi restituiscono il valore Graphics, in modo da far capire che si tratta, per *Mathematica*, entrambi di oggetti grafici, e possono essere trattati allo stesso modo. Supponiamo, per esempio, di utilizzare la seguente funzione:

In[30]:=
```
f[x_] := Sin[2 x] + Sin[3 x] + Sin[4 x]
```

Possiamo disegnare il grafico, assegnandogli un nome come per qualsiasi altro oggetto in *Mathematica*:

In[31]:= **grafico = Plot[f[x], {x, 0, 10}]**



Out[31]= **- Graphics -**

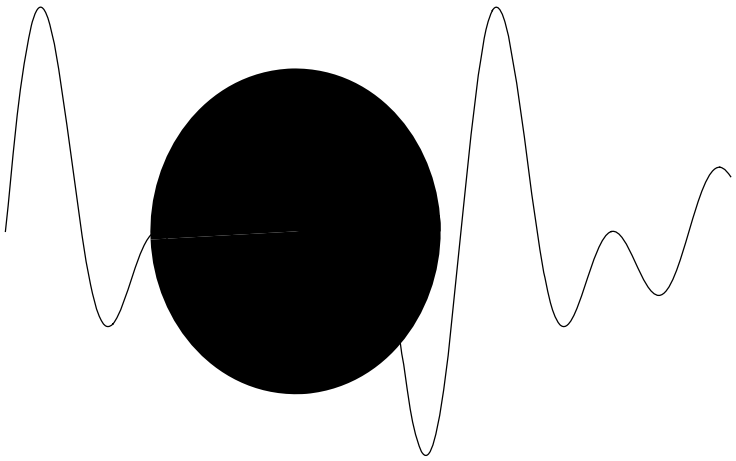
Adesso, definiamo una primitiva cerchio:

In[32]:= **cerchio = Graphics[Disk[{4, 0}, 2]]**

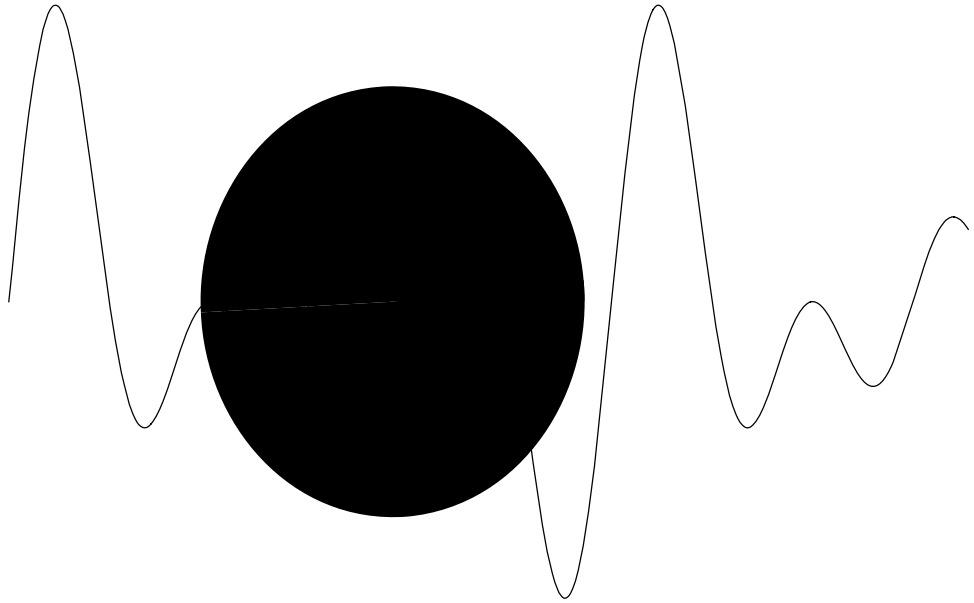
Out[32]= **- Graphics -**

E visualizziamole assieme:

In[33]:= **Show[cerchio, grafico]**



Out[33]= **- Graphics -**



- Graphics -

Possiamo vedere come, in questo caso, la funzione sia trattata come una primitiva grafica come le altre, e priva, quindi, di assi cartesiani oppure polari.

Possiamo, tramite le direttive grafiche, cambiare anche il colore delle primitive, definendo all'interno di Graphics, oltre alla figura da visualizzare, anche anche il colore che deve assumere. Possiamo, per esempio, definire una spirale di cerchi: prima di tutto definiamo la lista dei centri dei cerchi di questa spirale:

In[34]:=
```
centri = Table[{Sin[n], Cos[n]} n, {n, 1, 20, Pi / 6}];
```

Ho semplicemente creato le coordinate dei centri utilizzando le proprietà trigonometriche del cerchio, e per ogni punto ho aumentato il raggio moltiplicandolo per n.

Dopo aver definito i centri, definiamo i raggi dei cerchi:

In[35]:=
```
raggi = Table[n / 4, {n, 1, 20, Pi / 6}];
```

Notate come il numero dei valori della lista sia uguale a prima, e questo perchè dobbiamo creare dopo una relazione biunivoca fra questi elementi.

Allo stesso modo, adesso, creiamo la lista dei colori che deve assumere ogni singolo cerchio:

In[36]:=
```
colori = Table[Hue[n / 40], {n, 1, 20, Pi / 6}];
```

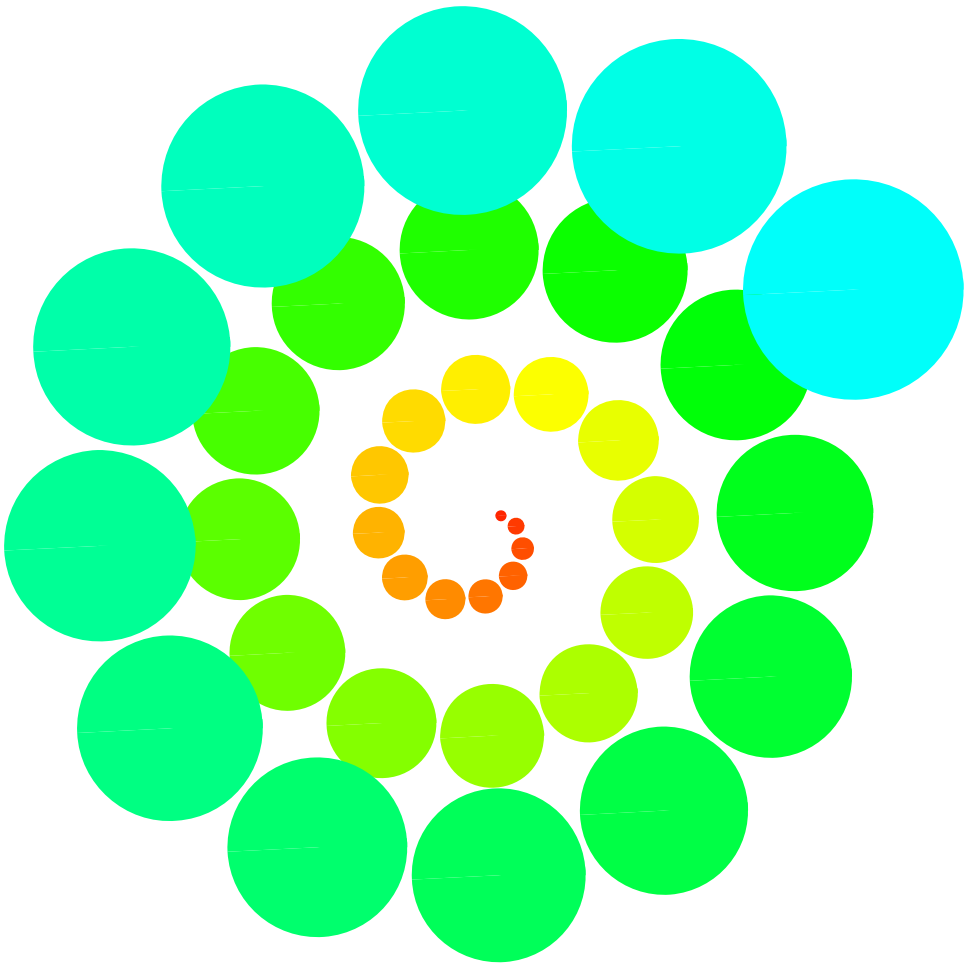
Abbiamo utilizzato la funzione Hue, ma potevamo, allo stesso modo, utilizzare RGBColor oppure GrayLevel. Ho usato Hue perchè permette, come RGBColor, di avere un colore e non un grigio ma, come GrayLevel, permette di crearlo utilizzando un solo argomento, e non tre come RGBColor. Tuttavia, dovrete poi utilizzare la funzione che meglio serve per rappresentare i vostri dati. Adesso, dopo aver creato le liste, andiamo a creare la lista di punti che andremo a visualizzare con il comando Graphics:

```
In[37]:= punti = Table[{{colori[[n]], Disk[centri[[n]], raggi[[n]]]}}, {n, 37}];

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

In questo modo, ho creato una lista di punti, utilizzando lo stesso indice per ognuno degli elementi, in modo da associarli nel modo corretto; per ogni elemento di Table, ho usato un elemento Disk ed un elemento colori. Notate anche come la funzione che definisce il colore, nella lista, debba essere definito prima del comando che crea la primitiva.

Una volta creati i punti, non ci resta che visualizzarli tramite il comando Show:

In[38]:=
```
Show[Graphics[punti], PlotRange → All, AspectRatio → Automatic]
```



Out[38]=
```
- Graphics -
```

Ah, com'è elegante...

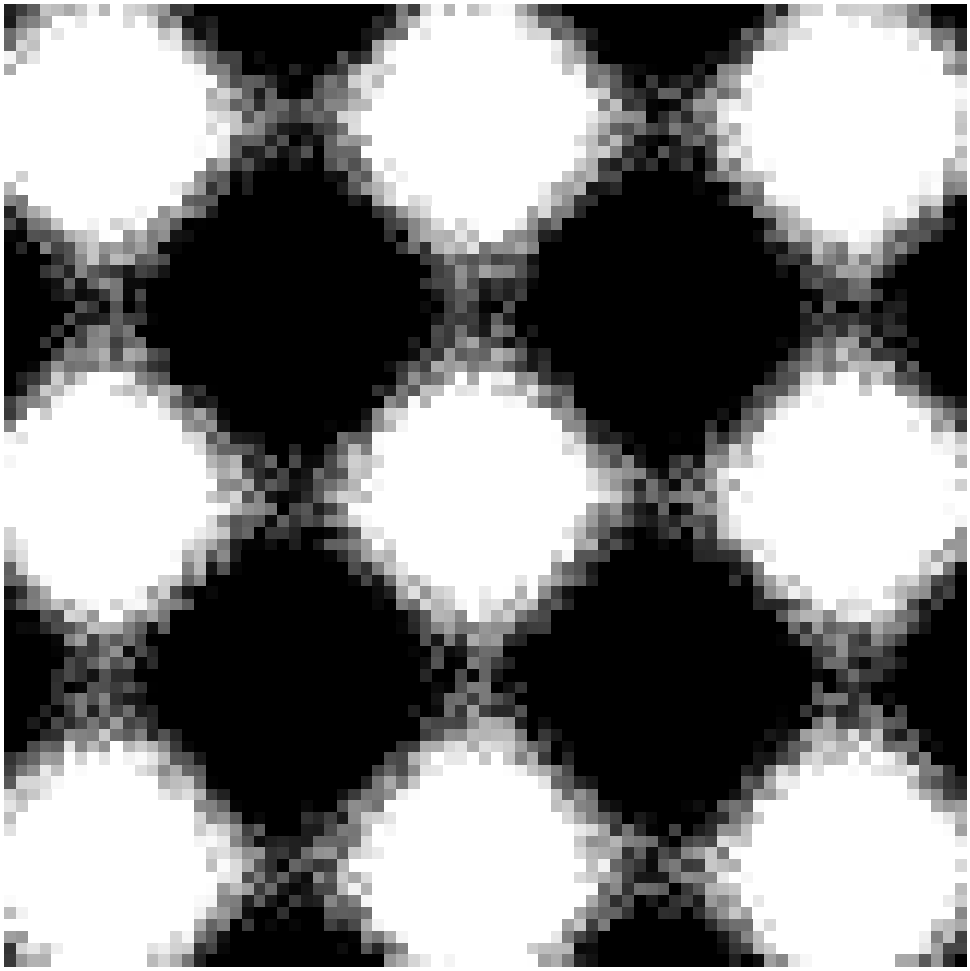
Ammetto, come avete capito, che all'inizio è un poco difficile utilizzare la grafica in modo adeguato in *Mathematica*; questo principalmente perchè non esistono pulsanti per le opzioni, e per qualsiasi cosa vogliamo fare, dobbiamo scrivere un poco di codice. Questo può essere seccante le prime volte, ma, una volta imparati quei pochi comandi e opzioni essenziali per ognuno di noi, vi rendererete conto che scrivere permette un livello di personalizzazione molto più elevato rispetto alle opzioni grafiche di altri programmi, e questo perchè scrivete voi stessi quello che volete, e non siete limitati a seguire il modo di pensare di chi ha disegnato l'interfaccia grafica di quel particolare programma.

Un altro comando utile, specialmente per visualizzare facilmente in modo grafico il contenuto di matrici, è il comando Raster. Possiamo, per fare un esempio, definire una matrice nel seguente modo:

In[39]:=
```
matrice =
  Table[Sin[x / 5] + Sin[y / 5] + Random[] / 2, {x, 0, 80}, {y, 0, 80}] // N;
```

Adesso, visualizziamola con Graphics:

In[40]:=
```
Show[Graphics[Raster[matrice]], AspectRatio → Automatic]
```



Out[40]=
```
- Graphics -
```

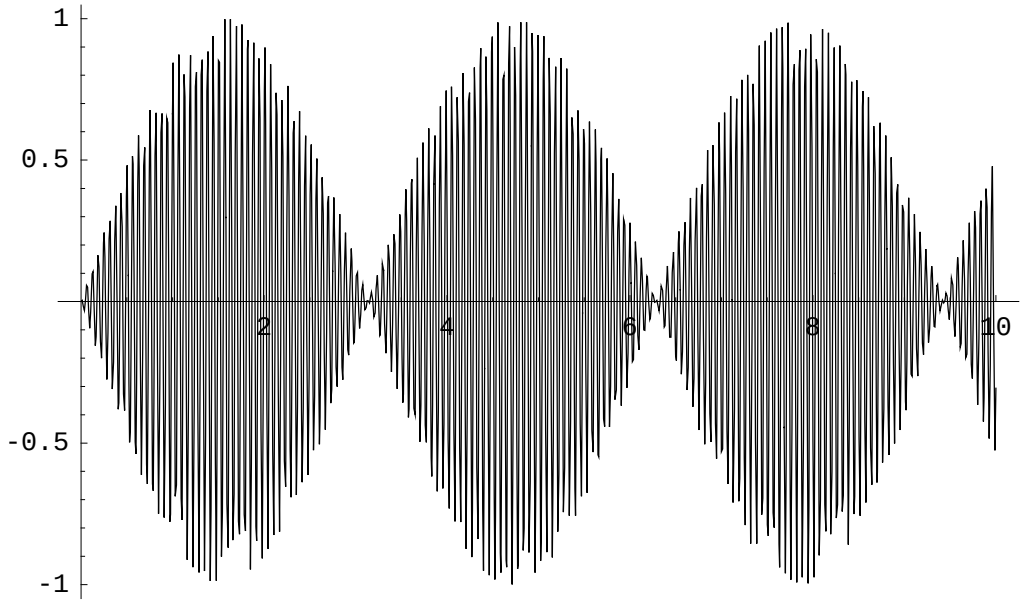
Vedete come è più facile guardare il contenuto di questa matrice mediante la rappresentazione grafica, piuttosto che andare a leggerne gli elementi.

Un altro elemento utile per visualizzare la grafica è Rectangle. Abbiamo già visto che può servire per rappresentare rettangoli, ma è più utile sapere che, all'interno di questo rettangolo, possiamo andare a metterci un'altra direttiva grafica: in questo modo possiamo, per esempio, andare ad annidare due grafici. Un possibile utilizzo consiste nel visualizzare un grafico assieme ad un suo ingrandimento: Consideriamo, per esempio, la seguente funzione:

In[41]:=
```
fun[x_] := Sin[x] Cos[100 x]
```

Proviamo a disegnarla:

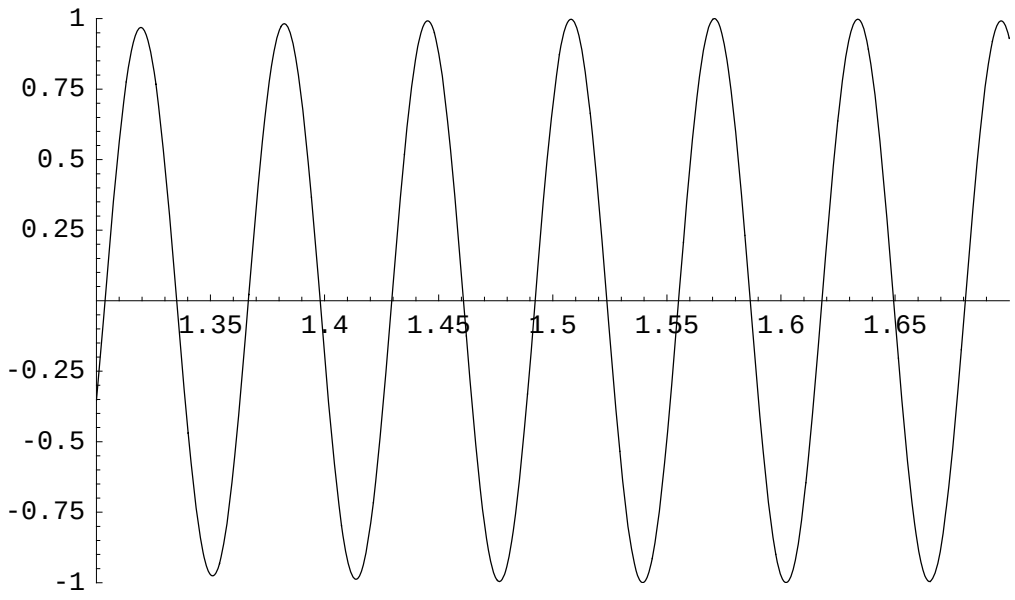
In[42]:=
```
globale = Plot[fun[x], {x, 0, 10}, MaxBend → 1]
```



Out[42]=
```
- Graphics -
```

Come possiamo vedere, non è molto chiara; vediamo che c'è un inviluppo, ma non si vede bene che la funzione modulata è effettivamente un seno. Proviamo adesso a visualizzare un particolare:

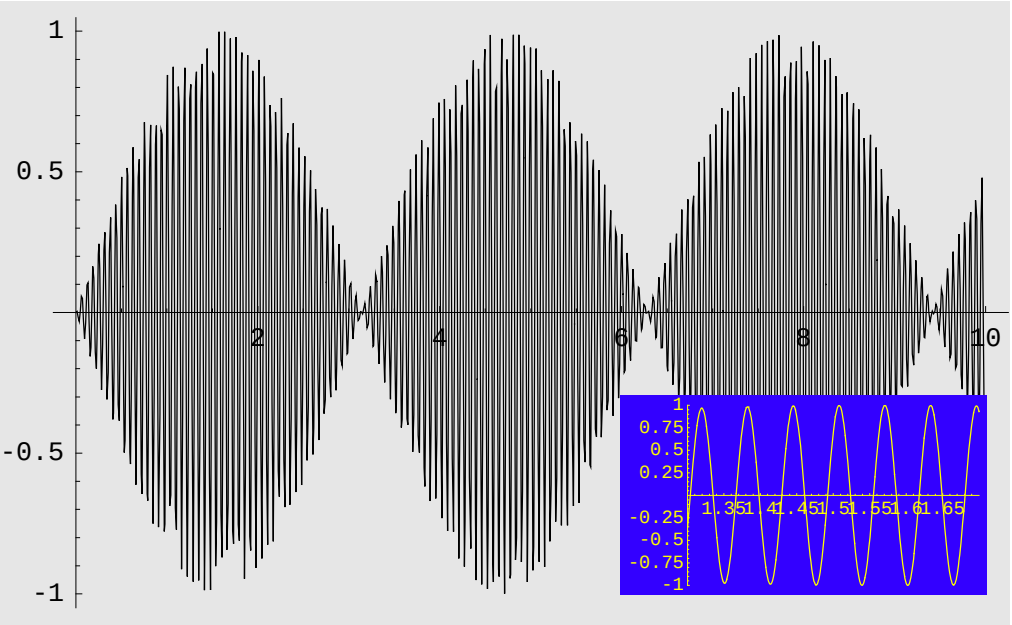
In[43]:=
```
particolare = Plot[fun[x], {x, 0, 3}, MaxBend → 0,
  PlotRange → {{1.3, 1.7}, {-1, 1.}}, PlotPoints → 300]
```



Out[43]= - Graphics -

Qua possiamo vedere che, opportunamente scalato l'asse x, si vede la funzione modulata, che effettivamente è una sinusoide. Possiamo, adesso, combinare entrambi i grafici in modo da ottenerne uno unico:

In[44]:=
```
Show[globale,
 Epilog ->
  Rectangle[{6, -1}, {10, -0.3},
   Show[
    particolare,
    Background → Hue[0.7],
    DefaultFont → {"Courier", 7},
    DisplayFunction → Identity
   ]
  ],
 Background → GrayLevel[0.9],
 DisplayFunction → $DisplayFunction
]
```



Out[44]= - Graphics -

Possiamo vedere ed apprezzare la formattazioe che ha permesso la creazione di questa immagine: dopo aver mostrato il grafico principale, abbiamo disegnato il rettangolo, e, dopo aver espresso le sue coordinate in cui viene disegnato, invece di essere riempito con un colore è stato riempito con un altro grafico, formattato a sua volta. Ho anche scalato il font del grafico più piccolo, altrimenti avrebbe avuto la stessa dimensione di quello più grande, mostrando un brutto effetto. Le opzioni di DisplayFunction sono necessarie perchè, in caso contrario, si sarebbero ottenuti delle immagini distinte, invece che annidate, e non era l'effetto che volevamo.

Inoltre, il comando Show, ha anche la potenzialità notevole di formattare un grafico senza doverlo ricalcolare. Quando creiamo un grafico con Plot, *Mathematica* valuta la funzione per calcolarsi i punti della funzione, e poi applica la formattazione che abbiamo deciso. Se vogliamo cambiare lo

stile del grafico, con Plot siamo costretti a rieseguire il comando, ricalcolandoci tutti i punti, e questo può farci perdere tempo se la funzione è complicata. Invece, memorizzando il grafico in una variabile, come abbiamo fatto prima per visualizzarli con Show, possiamo anche modificare la formattazione senza doverci ricalcolare la funzione, che in effetti è già memorizzata nella variabile: questo consente di rendere più veloce il nostro lavoro e, soprattutto, permette di memorizzare le formattazioni che preferiamo in delle funzioni, e poi usarle a nostro piacimento per tutte le funzioni che vogliamo, senza dover riscrivere ogni volta il risultato. Questo, però, lo lascio fare a voi, per vedere se ci riuscite!!!! Mica devo servirvi tutto su un piatto d'argento! E dove siamo, dico io?!?!?!?

Funzioni a due variabili

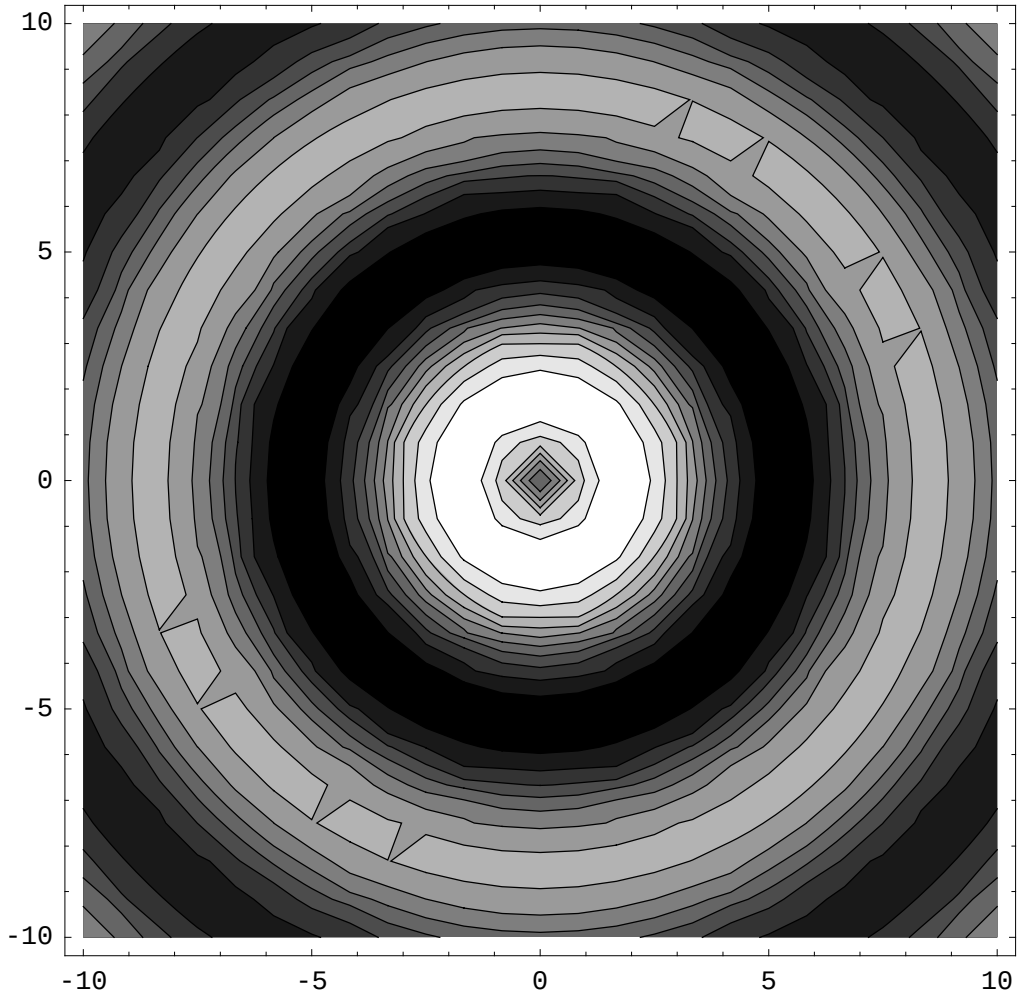
Anche se probabilmente verrà subito voglia di effettuare, per queste funzioni, il grafico tridimensionale, che comunque faremo subito dopo, è interessante notare che *Mathematica* permette di visualizzare questo tipo di funzioni anche nel campo 2D, usando differenti tipi di comandi rispetto a Plot, anche se sono effettivamente simili:

`ContourPlot[`*f*`, {`*x*`,` x_{min}`,` x_{max}`}, {`*y*`,` y_{min}`,` y_{max}`}]`	crea un grafico con contorni di *f* in funzione di *x* ed *y*
`DensityPlot[`*f*`, {`*x*`,` x_{min}`,` x_{max}`}, {`*y*`,` y_{min}`,` y_{max}`}]`	crea un grafico di densità *f*

Rappresentano su un piano la superfice, dove a diversi valori corrispondono diversi colori, un poco come succede nelle cartine geografiche fisiche, possiamo vedere il seguente esempio, per capirne il funzionamento:

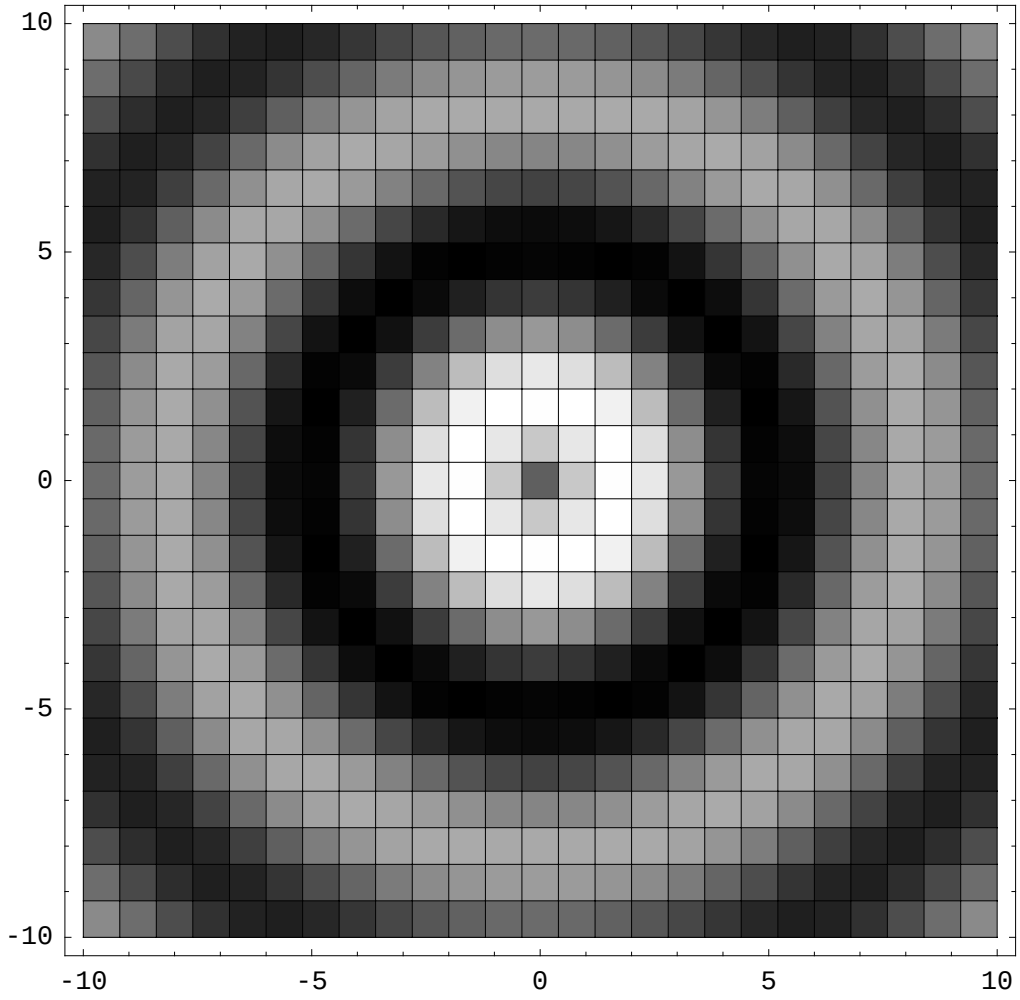
In[45]:=
```
f[x_, y_] := BesselJ[1, Sqrt[x^2 + y^2]]
```

In[46]:=
```
ContourPlot[f[x, y], {x, -10, 10}, {y, -10, 10}]
```



Out[46]=
```
- ContourGraphics -
```

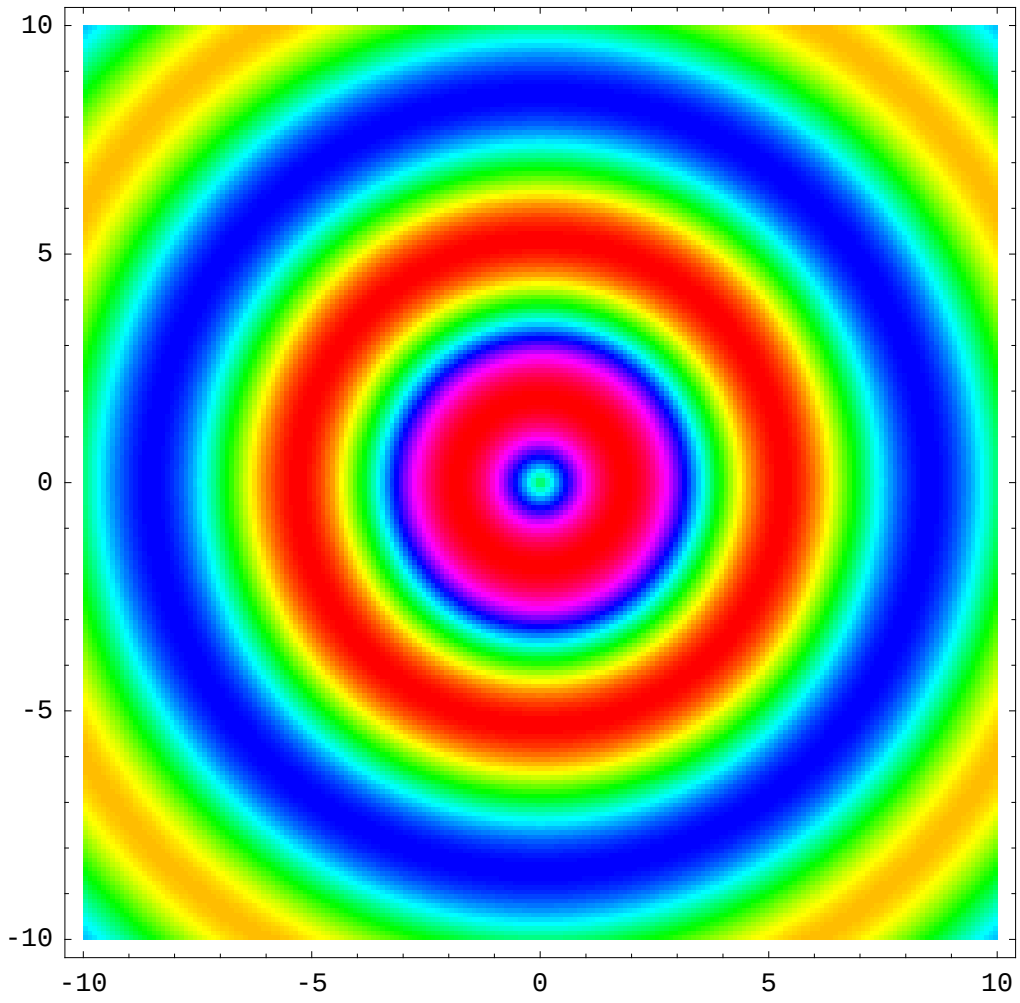
In[47]:=
```
DensityPlot[f[x, y], {x, -10, 10}, {y, -10, 10}]
```



Out[47]= - DensityGraphics -

Possiamo vedere come, nei due casi, le funzioni rappresentino i dati in modo diverso, anche se simile; nel primo caso effettivamente vengono create delle curve a valori costanti per la funzione. Per poter creare questo tipo di grafico, la funzione viene calcolata in maniera 'iterativa' da *Mathematica*. Nel secondo caso, invece, il programma si limita a calcolare la funzione al centro delle coordinate dei quadratini, e li rappresenta come farebbe il comando Raster, con la differenza che in questo caso, di default, viene anche visualizzata la griglia, cosa che comunque possiamo tranquillamente evitare mediante l'opportuna opzione. Inoltre, sempre di default, *Mathematica* colora i grafici con tonalità di scala di grigi, con i valori più chiari corrispondenti ai valori più elevati. Ed, anche in questo caso, siamo in grado di cambiare le cose giocando con le opzioni. Per esempio, possiamo manipolare il secondo grafico, per poter ottenere un'immagine, secondo me, più accattivante:

In[48]:=
```
DensityPlot[f[x, y], {x, -10, 10}, {y, -10, 10},
 Mesh → False,
 PlotPoints → 200,
 ColorFunction → Hue
]
```



Out[48]= - DensityGraphics -

Quello che è stato fatto, in questo caso, è stato nascondere le linee della griglia, con Mesh → False; poi abbiamo aumentato il numero di quadrati della griglia, rendendoli più piccoli e quindi l'immagine risulta più definita, anche se naturalmente aumenta il tempo e la memoria impiegati; questo con l'opzione PlotPoints → 200. Ho anche modificato, mediante l'opzione ColorFunction → Hue, anche il tipo di colorazione, che quindi non è più in scala di grigi, ma in tonalità di colore (ricordo che Hue definisce un colore con i tre valori tonalità, luminosità e saturazione). Le opzioni, in questo caso, sono simili, fra di loro e con Plot, tranne alcuni come Mesh, appunto. Quindi vedetevi il manuale e giocateci un poco, perchè a volte permettono una rappresentazione, magari meno scenografica, ma più chiara e diretta dei grafici tridimensionali. Non sempre, però!!! :-)

Introduzione al 3D

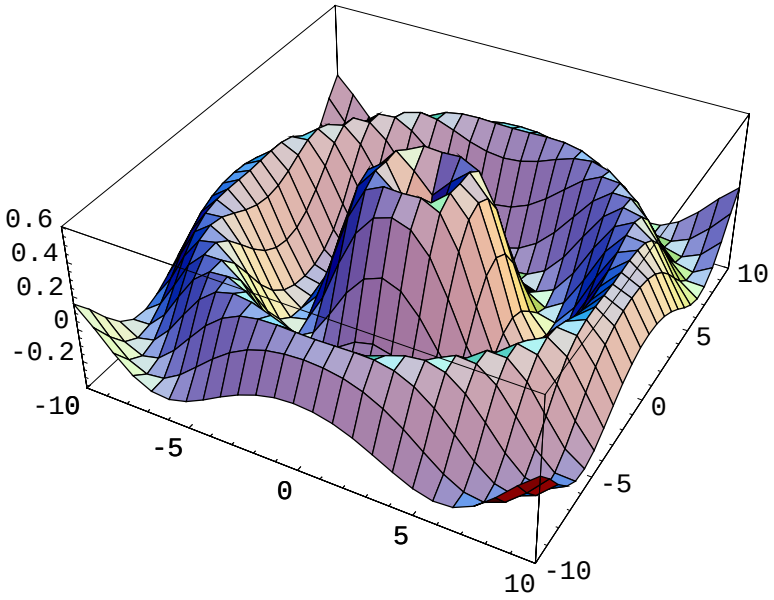
Quasi tutto quello che abbiamo applicato al 2D può essere applicato anche, con le opportune modifiche ed osservazioni, anche al caso 3D. Per tracciare grafici tridimensionali, il comando principale da usare è Plot3D:

`Plot3D[`f`, {`x`, `x_{min}`, `x_{max}`}, {`y`, `y_{min}`, `y_{max}`}]`	crea un grafico tridimensionale di f in funzione delle variabili x ed y

In questo caso, la funzione è molto simile a quella già vista Plot, con l'accortezza di tener conto anche della seconda variabile:

In[49]:=
```
f[x_, y_] := BesselJ[1, Sqrt[x^2 + y^2]]
```

In[50]:=
```
Plot3D[f[x, y], {x, -10, 10}, {y, -10, 10}]
```



Out[50]=
```
- SurfaceGraphics -
```

Come possiamo vedere, la sintassi è effettivamente molto simile: anche Plot3D ha un considerevole numero di opzioni. Ne ha anche qualcuna in più, per tener conto, per esempio, del punto di vista del grafico e delle luci in gioco:

```
In[51]:= Options[Plot3D]

Out[51]= {AmbientLight → GrayLevel[0], AspectRatio → Automatic, Axes → True,
  AxesEdge → Automatic, AxesLabel → None, AxesStyle → Automatic,
  Background → Automatic, Boxed → True, BoxRatios → {1, 1, 0.4},
  BoxStyle → Automatic, ClipFill → Automatic, ColorFunction → Automatic,
  ColorFunctionScaling → True, ColorOutput → Automatic, Compiled → True,
  DefaultColor → Automatic, DefaultFont :→ $DefaultFont,
  DisplayFunction :→ $DisplayFunction, Epilog → {}, FaceGrids → None,
  FormatType :→ $FormatType, HiddenSurface → True, ImageSize → Automatic,
  Lighting → True, LightSources → {{{1., 0., 1.}, RGBColor[1, 0, 0]},
    {{1., 1., 1.}, RGBColor[0, 1, 0]}, {{0., 1., 1.}, RGBColor[0, 0, 1]}},
  Mesh → True, MeshStyle → Automatic, Plot3Matrix → Automatic,
  PlotLabel → None, PlotPoints → 25, PlotRange → Automatic,
  PlotRegion → Automatic, Prolog → {}, Shading → True,
  SphericalRegion → False, TextStyle :→ $TextStyle,
  Ticks → Automatic, ViewCenter → Automatic,
  ViewPoint → {1.3, -2.4, 2.}, ViewVertical → {0., 0., 1.}}
```

Anche in questo caso ci sono numerose opzioni, e adesso le elenchiamo come prima, stavolta evitando di dover spiegare quelle già viste, e concentrandosi più su quelle nuove. Se volete sapere pure il sognificato di quelle di cui ometterò la spiegazione, andate a rivedervi le opzioni di Plot:

- ⋆ **AmbientLight**: definisce la quantità di illuminazione ambiente di un grafico tridimensionale, oltre al colre di questa luce. Sicuramente chi avrà fatto almeno una volta grafica 3D conoscerà bene questo concetto. Si usa definendo una delle funzioni colore come Hue o RGBColor e specificando un colore, che verrà usato come luce ambiente.

- ⋆ **AspectRatio**: vedi Plot

- ⋆ **Axes**: vedi Plot

- ⋆ **AxesEdge**: definisce in quali lati della 'scatola' che contiene il grafico tridimensionale devono essere disegnati gli assi; si definisce la lista $\{\{dir_y, dir_z\}, \{dir_x, dir_z\}, \{dir_x, dir_y\}\}$, dove i valori, che possono essere + o -1, specificano se gli assi devono essere disegnati negli spigoli con valori più grandi o più piccoli della coordinata corrispondente. Un elemento della lista può essere sostituito da Automatic.

- ⋆ **AxesLabel**: vedi Plot

- ⋆ **AxesStyle**: vedi Plot

- ⋆ **Background**: vedi Plot

- ⋆ **Boxed**: specifica se il grafico deve oppure no essere contenuto in una scatola, visualizzandone gli spigoli. Può essere, naturalmente, True oppure False.

- ⋆ **BoxRatios**: definisce le proporzioni fra le varie coordinate *x, y, z* del grafico, deformando la scatola in cui è contenuto. Si usa definendo una lista di tre valori che specificano il rapporto fra le tre dimensioni.

- ⋆ **BoxStyle**: specifica lo stile con ciu devono essere disegnati gli spigoli della scatola. Di solito gli si assegna un colore e/o uno spessore.

- ⋆ **ClipFill**: con questa opzione si decide se, quando il grafico non viene contenuto interamente dall'asse z, le zone in cui il grafico è tagliato perchè è fuori dai margini, deve essere lasciato vuoto, oppure riempito con zone. Si utilizza None, oppure una lista di due colori, che definiscono il colore del grafico tagliato in alto ed in basso.

- ⋆ **ColorFunction**: vedi Plot

- ⋆ **ColorFunctionScaling**: è un parametro booleano che definisce se la funzione dei colori del grafico deve essere scalata in modo da essere contenuta fra 0 ed 1, in modo da utilizzare tutta la scala di colori della funzione.

- ⋆ **ColorOutput**: vedi Plot

- ⋆ **Compiled**: vedi Plot

- ⋆ **DefaultColor**: vedi Plot

- ⋆ **DefaultFont:** vedi Plot

- ⋆ **DisplayFunction**: vedi Plot

- ⋆ **Epilog**: vedi Plot

- ⋆ **FaceGrids**: specifica se devono essere visualizzate o meno le griglie nelle facce della scatola che contiene il grafico. Settato su All definisce tutte le griglie, altrimenti bisogna specificare le facce. Inoltre è possibile personalizzare la griglia come nel caso del comando Plot.

- ⋆ **FormatType**: vedi Plot

- ⋆ **HiddenSurface**: con questa opzione decidiamo se visualizzare o meno le facce nascoste; settato su False, permette di rendere il grafico 'trasparente'.

- ⋆ **ImageSize**: vedi Plot

- ⋆ **Lighting**: con questa opzione booleana decidiamo se visualizzare o meno le luci per la superfice.

- ⋆ **LightSources**: definisce una lista di luci che verranno usate nel grafico. ogni elemento è una lista a sua volta, conenente come primo elemento una lista di tre numeri che definiscono le coordinate della

luce, e come secondo elemento il colore stesso della luce. Questa opzione è una delle prime da cambiare, se volete modificare l'aspetto del grafico.

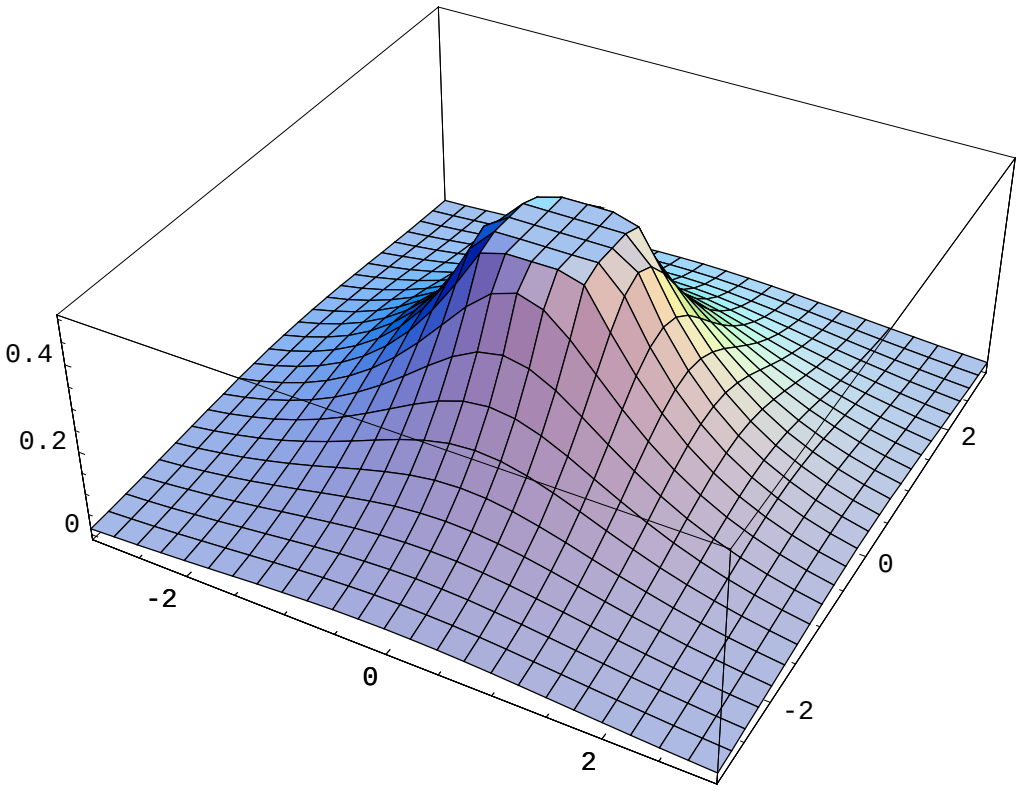
- **Mesh**: definisce se visualizzare o meglio le linee che compongono la mesh del grafico tridimensionale.
- **MeshStyle**: definisce lo stile delle linee della mesh del grafico
- **Plot3Matrix**: non usare questa opzione. Si mantiene solo per compatibilità con le vecchissime versioni di *Mathematica*. Neanch'io so cosa faccia questa opzione, e vivo benissimo senza saperlo.
- **PlotLabel**: vedi Plot
- **PlotPoints**: vedi Plot, ma qui ricordo che definisce il numero di punti in cui viene diviso ognuno dei due assi x e y in cui viene calcolata la funzione. Aumentare questo valore aumenterà la definizione del grafico.
- **PlotRange**: vedi Plot
- **PlotRegion**: vedi Plot
- **Prolog**: vedi Plot
- **Shading**: definisce se la superfice debba essere o meno illuminata.
- **SphericalRegion**: questa opzione specifica se l'immagine finale debba essere scalata in modo che una sfera che contenga interamente la funzione sia visibile (teoricamente) nella figura. Alquanto inutile...
- **TextStyle**: vedi Plot
- **Ticks**: vedi Plot
- **ViewCenter**: scala l'immagine in modo che il punto selezionato appaia al centro dell'immagine stessa. Questa opzione scala interamente il grafico, con assi e tutto quanto.
- **ViewPoint**: definisce il punto di vista nello spazio tridimensionale dal quale si guarda il grafico. rappresenta un'altra della opzioni più utilizzate per modificare l'aspetto del grafico.
- **ViewVertical**: specifica, mediante una lista di tre valori, qual'è la direzione nello spazio del grafico che deve apparire verticale nell'immagine finale. permette, in poche parole, di ruotare il grafico.

Vediamo di fare un esempio concreto di qualcuna di queste opzioni. Definiamo una funzione sempilce, per adesso:

In[52]:=
```
f[x_, y_] := Exp[-Sqrt[x^2 + y^2]]
```

Adesso, visualizziamola:

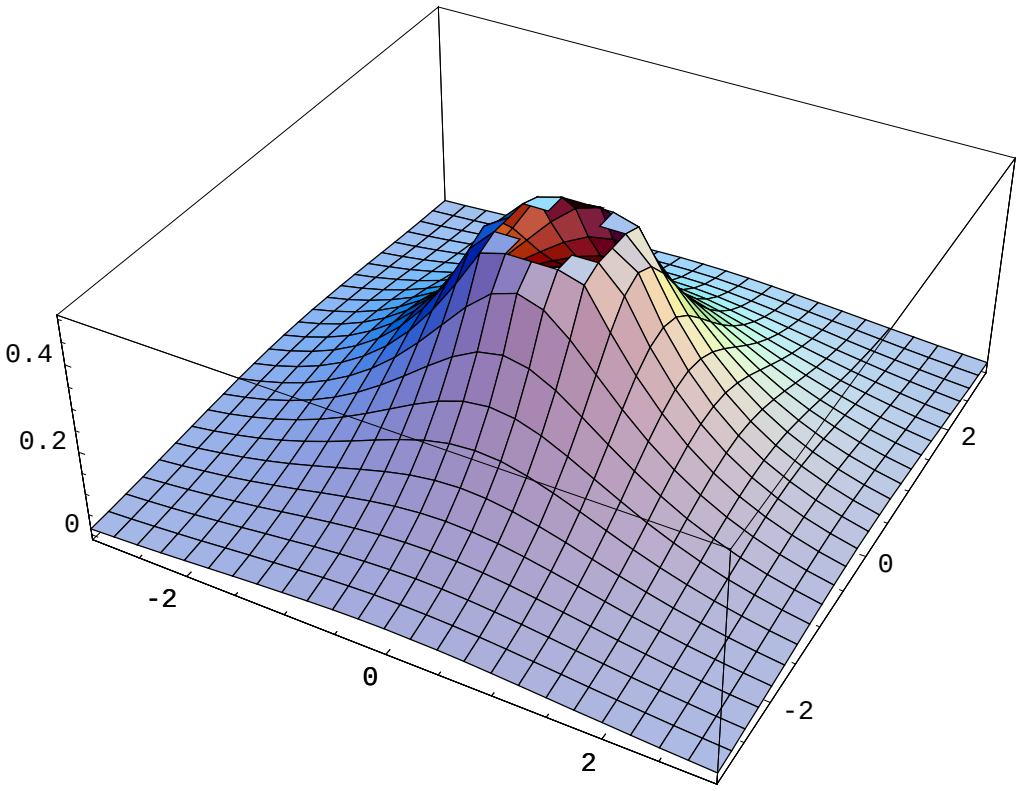
In[53]:=
```
Plot3D[f[x, y], {x, -3, 3}, {y, -3, 3}]
```



Out[53]=
```
- SurfaceGraphics -
```

Come possiamo vedere, abbiamo la funzione che risulta 'tagliata' sopra. Possiamo vederlo meglio se usiamo l'opzione ClipFill:

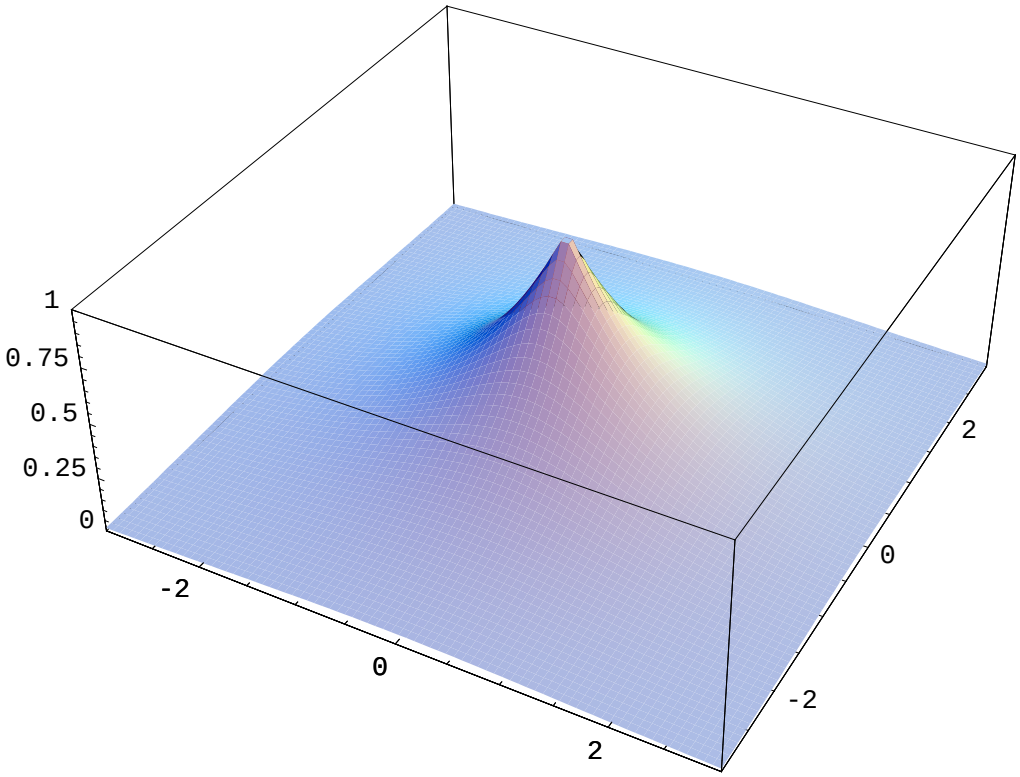
In[54]:= **Show[%, ClipFill → None]**



Out[54]= **- SurfaceGraphics -**

Ah! Adesso si vede che effettivamente c'è un buco!!! E vedete anche come posso ridisegnare il grafico semplicemente usando il comando Show, evitando di dover calcolare tutti i punti. Adesso, voglio visualizzarlo completamente, ed, in aggiunta, voglio eliminare la mesh e rendere il grafico più dettagliato:

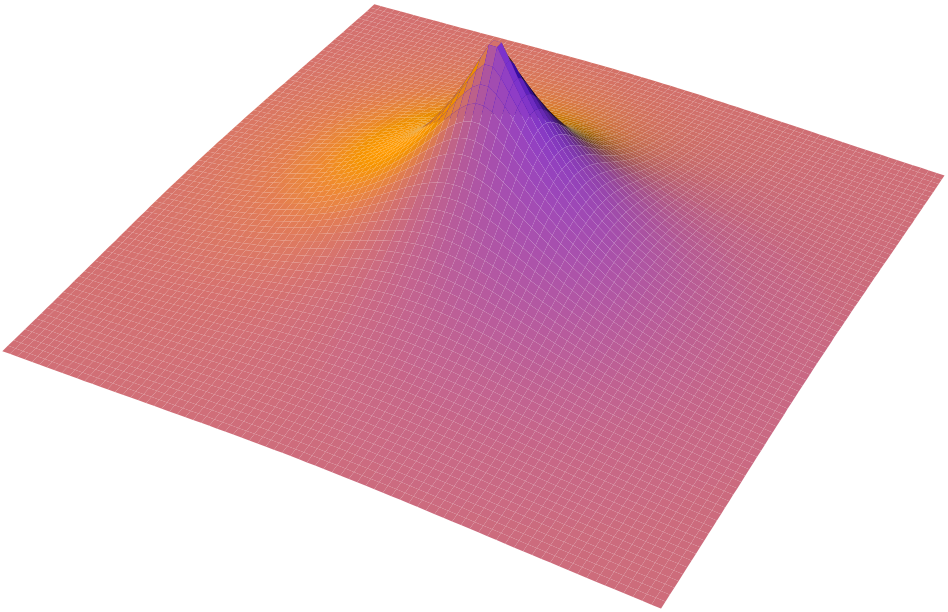
In[55]:=
```
Plot3D[f[x, y], {x, -3, 3}, {y, -3, 3},
 Mesh → False,
 PlotPoints → 80,
 PlotRange → {Automatic, Automatic, {0, 1}}]
```



Out[55]= `- SurfaceGraphics -`

Adesso si vede che, effettivamente, la curva è più ripida di quanto ci si aspettasse ad una prima occhiata del primo grafico tagliato. Proviamo a cambiare le luci, ed ad eliminare la scatola che lo racchiude. Inoltre, cambieremo anche punto di vista

```
In[56]:= Show[%,
          LightSources → {{{-3, 4, 3}, Hue[0.1]}, {{3, -3, 4}, Hue[0.7]}},
          Boxed → False,
          Axes → None
           ViewPoint -> {-0.894, 3.078, 1.083}
         ]
```



```
Out[56]= - SurfaceGraphics -
```

Come possiamo vedere, anche se si tratta della stessa funzione, abbiamo ottenuto un aspetto alquanto diverso da quello di partenza, e questo giocando soltanto con alcune delle opzioni disponibili.

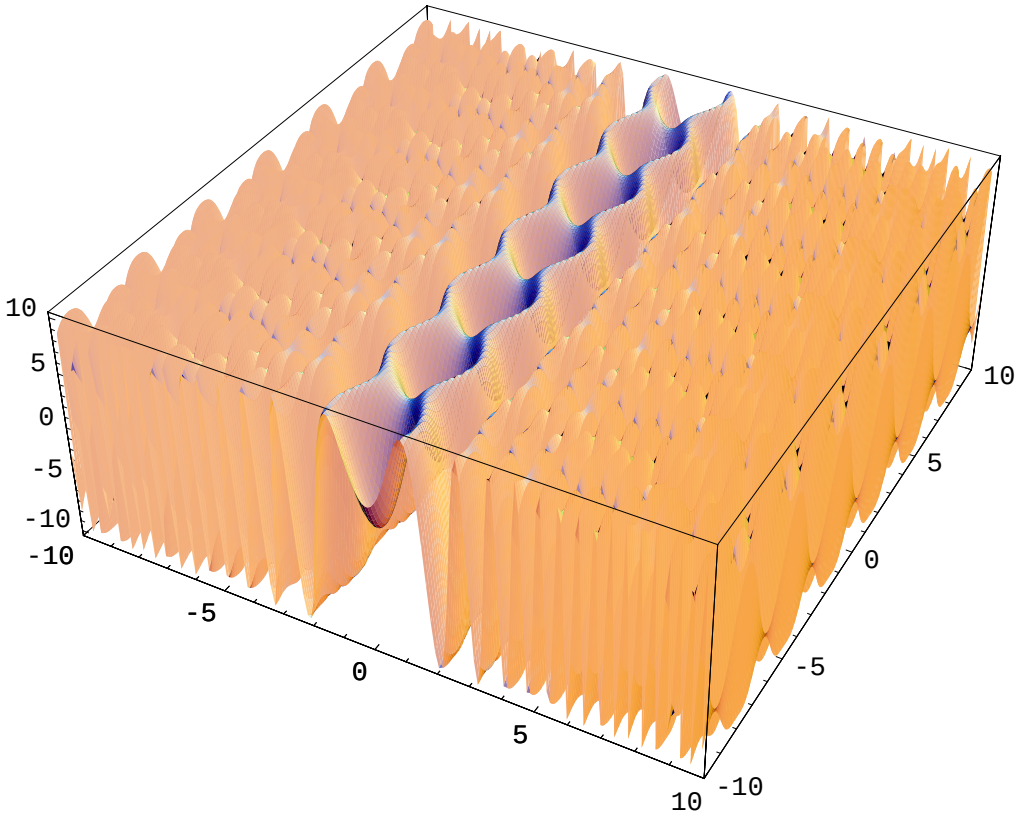
In generale è possibile, come nel caso bidimensionale, andare a disegnare contemporaneamente più superfici ma questo, a meno di casi specifici, è sconsigliabile, o almeno io lo sconsiglio, perchè se non si sa bene quello che si vuole ottenere, si rischia di ottenere un'immagine confusa e non chiara, soprattutto a causa di porzioni di superfici che rimangono invariabilmente scoperte. Anche perchè, in questo caso, non sempre è possibile visualizzare correttamente i risultati.

Qua, infatti, sento personalmente una delle mancanze di questo programma rispetto a Matlab, ovvero la possibilità di ruotare in tempo reale, col mouse, un grafico 3D. Questo ha un suo perchè, ovviamente. *Mathematica* genera, una volta calcolati i dati, non un'immagine tridimensionale vera e propria, ma un'immagine bidimensionale in formato Encapsulated PostScript, che rappresenta il grafico tridimensionale. Tuttavia, l'immagine rimane bidimensionale a tutti gli effetti, con l'ovvia

conseguenza dell'impossibilità di rotazione senza ricalcolo. Tuttavia questo permette di elaborare le immagini, sovrapporle e modificarne l'aspetto con i semplici e potenti comandi che abbiamo imparato, magari con un notevole risparmio di tempo e con la possibilità, per esempio, di sovrapporre un'immagine bidimensionale ad una tridimensionale. Questo permette di ottenere un layout più professionale e più personalizzabile. Ma se solo ci fosse pure la rotazione in tempo reale!!!! Ah, non mi accontento mai...

Bisogna anche vedere come, effettivamente, avendo memorizzato il grafico tridimensionale in una variabile, i valori calcolati sono disponibili anche per le altre visualizzazioni. Per esempio, posso avere:

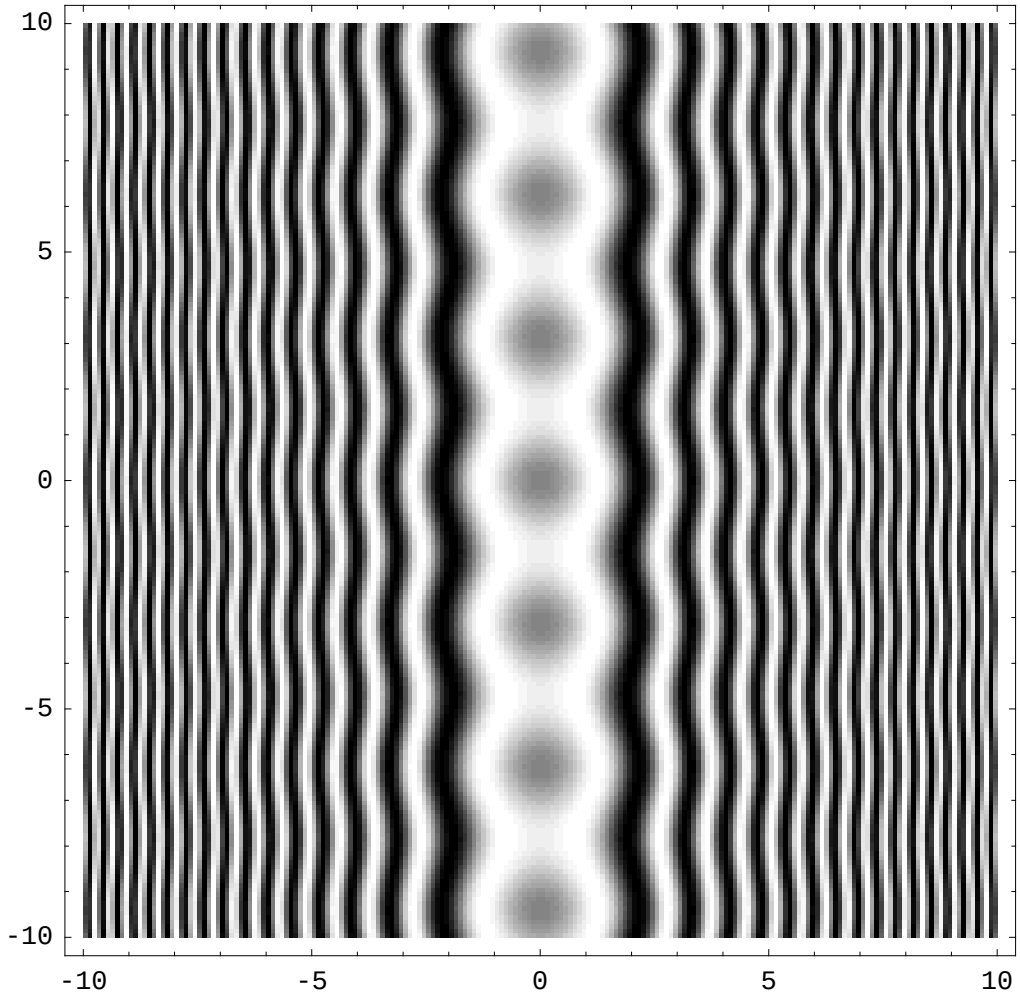
In[57]:=
```
graf = Plot3D[10 Sin[x^2 + Sin[y]^2],
  {x, -10, 10}, {y, -10, 10}, PlotPoints → 200, Mesh → False]
```



Out[57]= `- SurfaceGraphics -`

Adesso, possiamo anche visualizzare il grafico di densità, che mostra il grafico in maniera più chiara di quanto possa fare il grafico tridimensionale, pur modificando punto di vista e colori:

In[58]:= **Show[DensityGraphics[graf]]**



Out[58]= **-** DensityGraphics **-**

che, secondo me, da un'idea pioù chiara del comportamento della funzione. Una volta impratichiti con le opzioni e le visualizzazioni, trovare il modo più elegante ed efficace di visualizzare dati sarà una bella sfida, e non vi accontenterete mai, cercando sempre di meglio. D'altronde, qua è dove la scienza finisce e dove inizia la sensibilità e la forma artistica e personale di ognuno, dove ci possiamo esprimere individualmente al meglio. Non commettete l'errore di essere asettici. L'obiettività è importante, ma l'aridità rende difficile e poco interessante anche il più utile dei lavori!

Visualizzazione dati

Quando si tratta di visualizzare funzioni, si tratta semplicemente di inserire le formule che ci interessano, e vedere come si comportano. Dobbiamo invece agire diversamente (ma non di molto), quando si tratta di visualizzare dati numerici. Infatti, capita molto spesso di dover interpretare dei dati sperimentali visualizzandoli in maniera opportuna. Questi devono usare, quindi, diversi comandi rispetto a quelli per il tracciamento delle funzioni, anche se, in verità, sono molto simili:

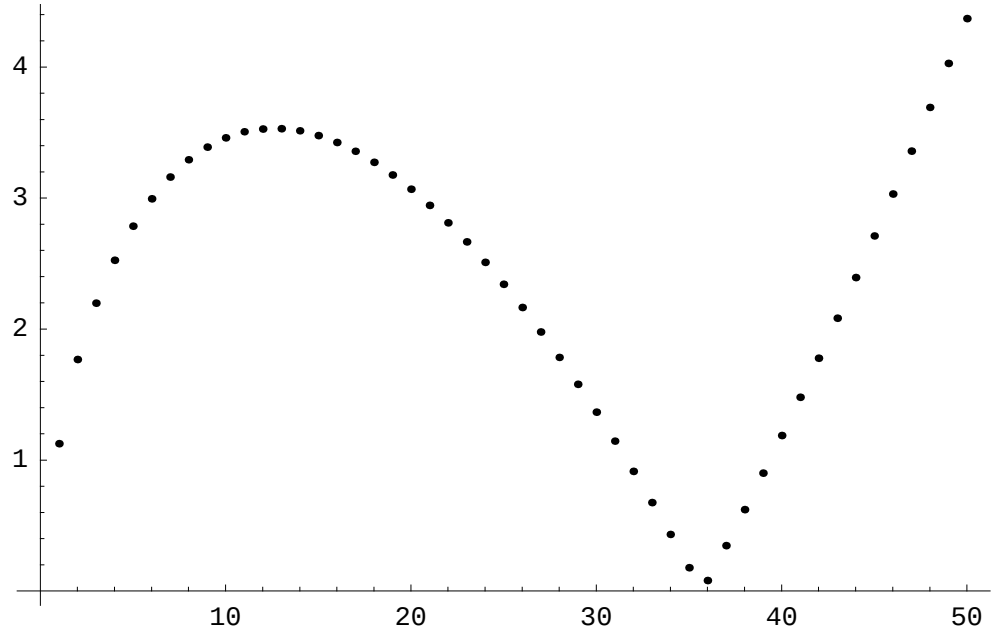
`ListPlot[{`y_1`, `y_2`, … }]`	disegna il grafico dei dati y_1, y_2, ... per valori di x pari a 1, 2, ...
`ListPlot[{{`x_1`, `y_1`}, {`x_2`, `y_2`}, … }]`	disegna i punti (x_1,y_1), ...
`ListPlot[`*list*`, PlotJoined -> True]`	unisce i punti con delle linee
`ListPlot3D[{{`z_{11}`, `z_{12}`, … }, {`z_{21}`, `z_{22}`, … }, … }]`	crea un grafico tridimensionale dove la mesh rappresenta i valori z_{yx} della matrice
`ListContourPlot[`*array*`]`	crea un grafico di contorno con curve di livello
`ListDensityPlot[`*array*`]`	crea un grafico di densità

Naturalmente, per ora creeremo sempre le nostre liste a partire da funzioni. Vedremo più avanti come fare per importare dati esterni nel programma. Possiamo cominciare a vedere come si rappresenta una lista di dati:

In[59]:=
```
lista = Table[Abs[RiemannSiegelTheta[x / 2]], {x, 50}];
```

Una volta creata la lista di dati, possiamo visualizzarla:

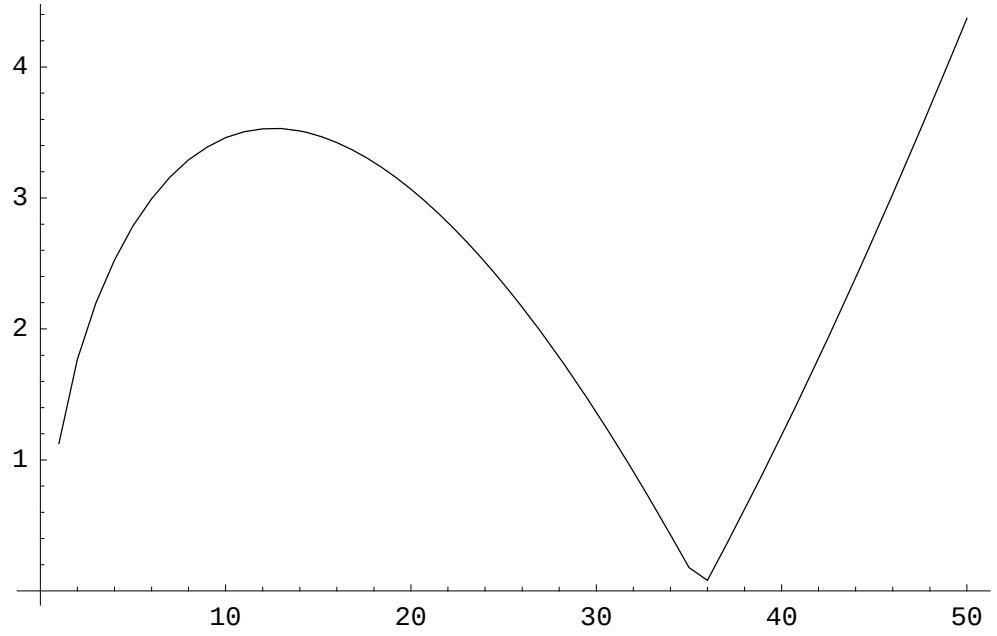
In[60]:= **ListPlot[lista]**



Out[60]= **- Graphics -**

Come possiamo vedere, in questo caso vediamo solamente dei punti che rappresentano i dati. Se volessimo un grafico a linea, dovremmo utilizzare l'opzione corretta, che abbiamo visto sopra:

In[61]:= **grafico1 = ListPlot[lista, PlotJoined → True]**

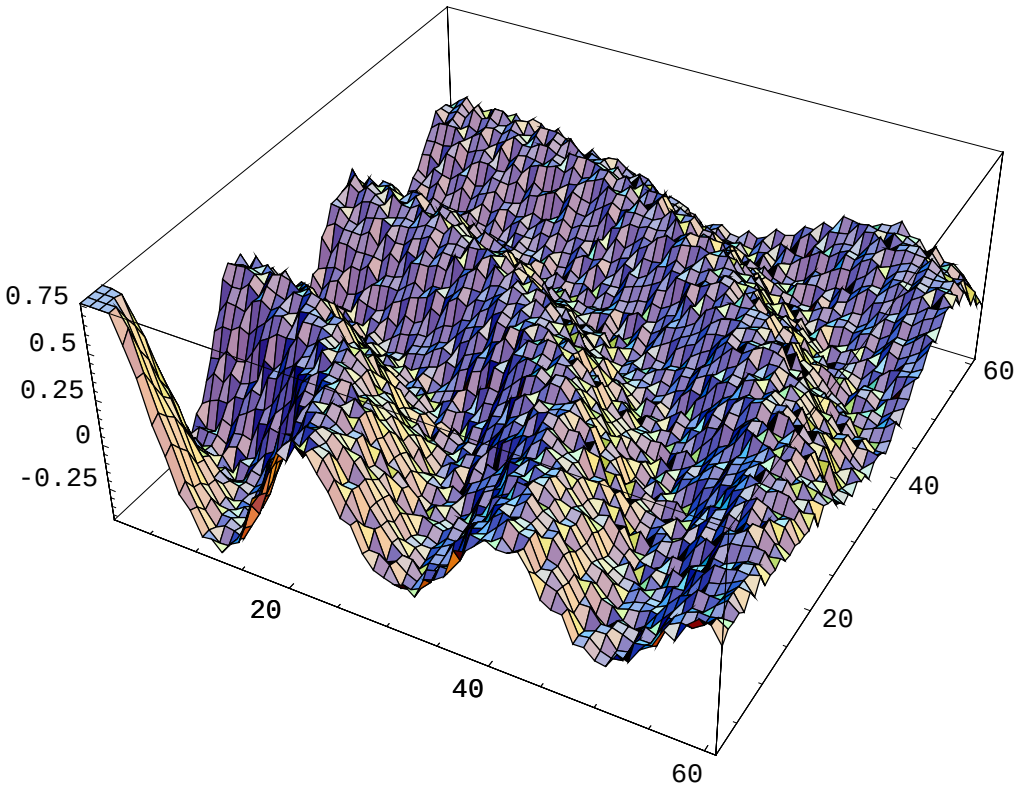


Out[61]= **- Graphics -**

Possiamo vedere come sia relativamente semplice visualizzare dati ed, inoltre, questi comandi hanno le stesse opzioni di quelli usati per le funzioni. Possiamo vedere anche come si comportano i grafici di dati tridimensionali, e per farlo occorre definire una matrice di valori che devono essere visualizzati:

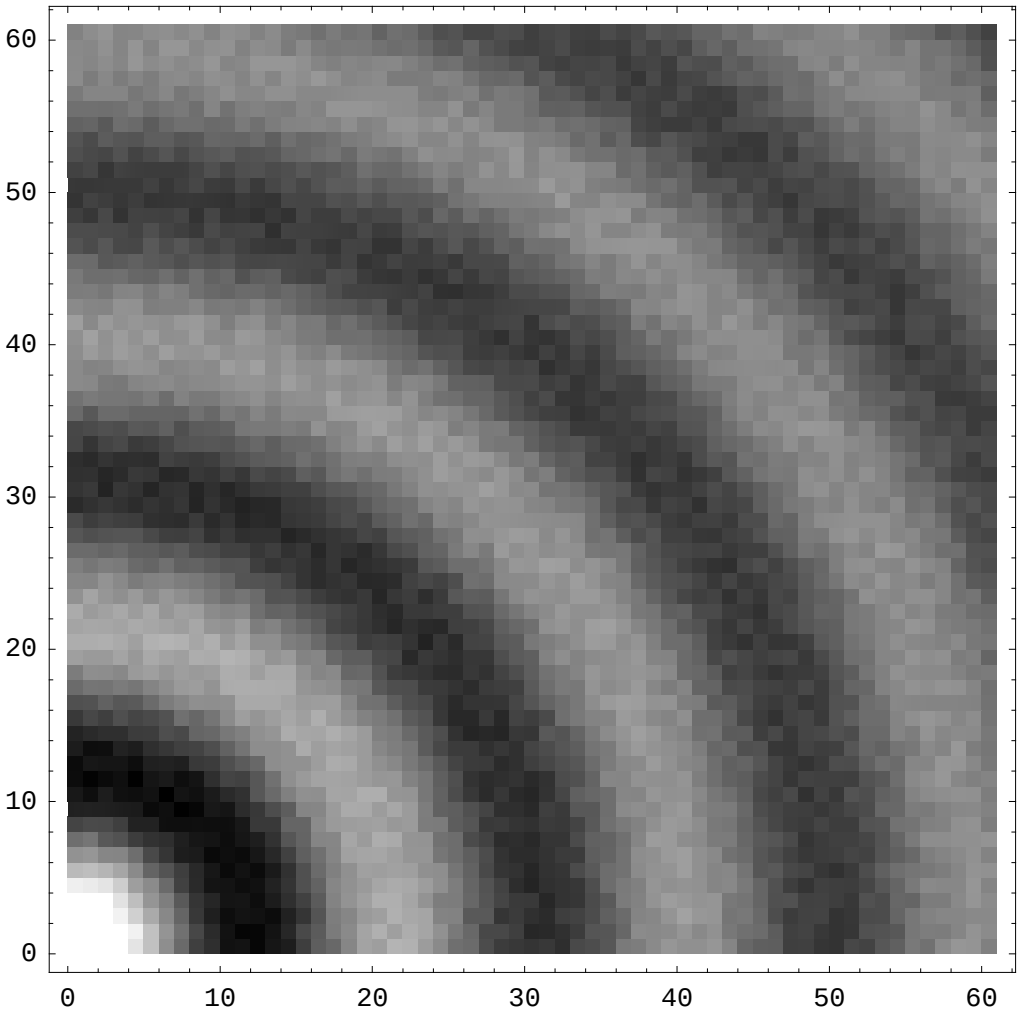
In[62]:=
```
lista3D = Table[BesselJ[0, Sqrt[(x / 3)^2 + (y / 3)^2]] + Random[] / 10,
   {x, 0, 60}, {y, 0, 60}];
```

In[63]:=
```
ListPlot3D[lista3D]
```



Out[63]=
```
- SurfaceGraphics -
```

In[64]:= **Show[DensityGraphics[%], Mesh → False]**



Out[64]= **-** DensityGraphics **-**

Come potete vedere, la visualizzazione di dati non è niente di complicato, ammesso che si sia capito come funzionano i comandi corrispondenti per il tracciamento delle funzioni. Si tratta solo di sperimentare e, una volta che vi siete impratichiti, di provare ad elaborare dei dati reali, magari di cui conoscete già risultati et similia, per essere sicuri di avere i risultati corretti.

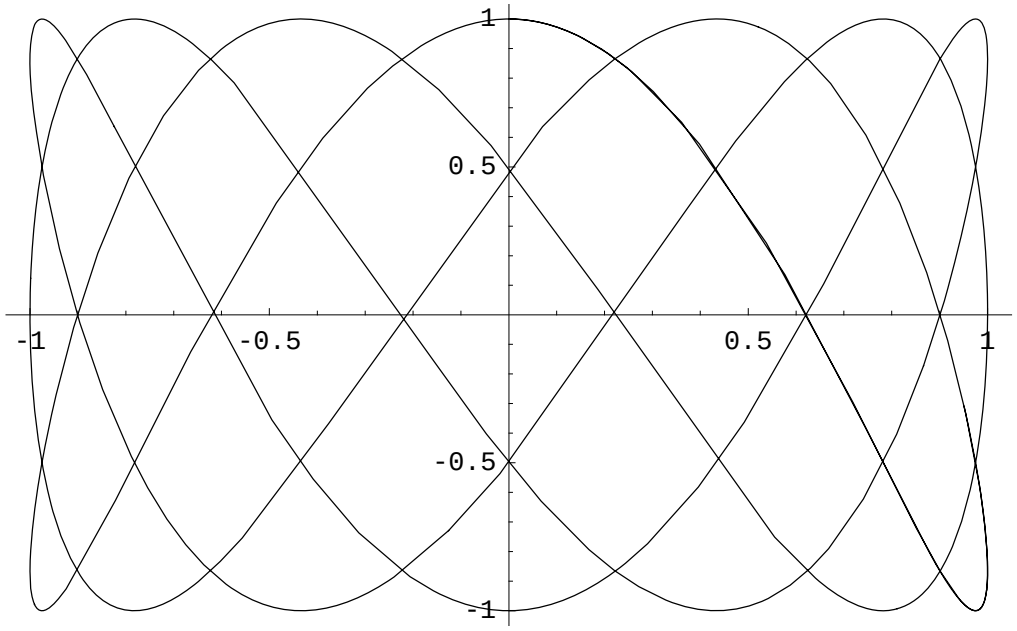
Grafici parametrici

A volte non è sufficiente usare i comandi sopra elencati, perchè alcune tipologie di grafico, come appunto questi parametrici, non sono visualizzabili direttamente come funzioni ad una oppure a due variabili, e questo perchè dipendono da essa (o da esse), sia i valori dell'ascissa che quelli dello'ordinata. Per questi tipi di grafici esistono appositi comandi:

`ParametricPlot[{`f_x`,` f_y`}, {`t`,` t_{min}`,` t_{max}`}]`	crea un grafico parametrico
`ParametricPlot[{{`f_x`,` f_y`}, {`g_x`,` g_y`}, … }, {`t`,` t_{min}`,` t_{max}`}]`	disegna assieme diversi grafici parametrici
`ParametricPlot[{`f_x`,` f_y`}, {`t`,` t_{min}`,` t_{max}`}, AspectRatio -> Automatic]`	conserva le proporzioni della funzione parametrica
`ParametricPlot3D[{`f_x`,` f_y`,` f_z`}, {`t`,` t_{min}`,` t_{max}`}]`	crea il grafico di una superfice parametrica
`ParametricPlot3D[{`f_x`,` f_y`,` f_z`}, {`t`,` t_{min}`,` t_{max}`}, {`u`,` u_{min}`,` u_{max}`}]`	indovia indovinello
`ParametricPlot3D[{`f_x`,` f_y`,` f_z`,` s`}, … ]`	disegna quella parte di superfice concorde con *l'* equazione *s*
`ParametricPlot3D[{{`f_x`,` f_y`,` f_z`}, {`g_x`,` g_y`,` g_z`}, … }, …`	disegna assieme diverse superfici

Vediamo un esempio classico di curve parametriche, ovvero le figure di Lissajouis:

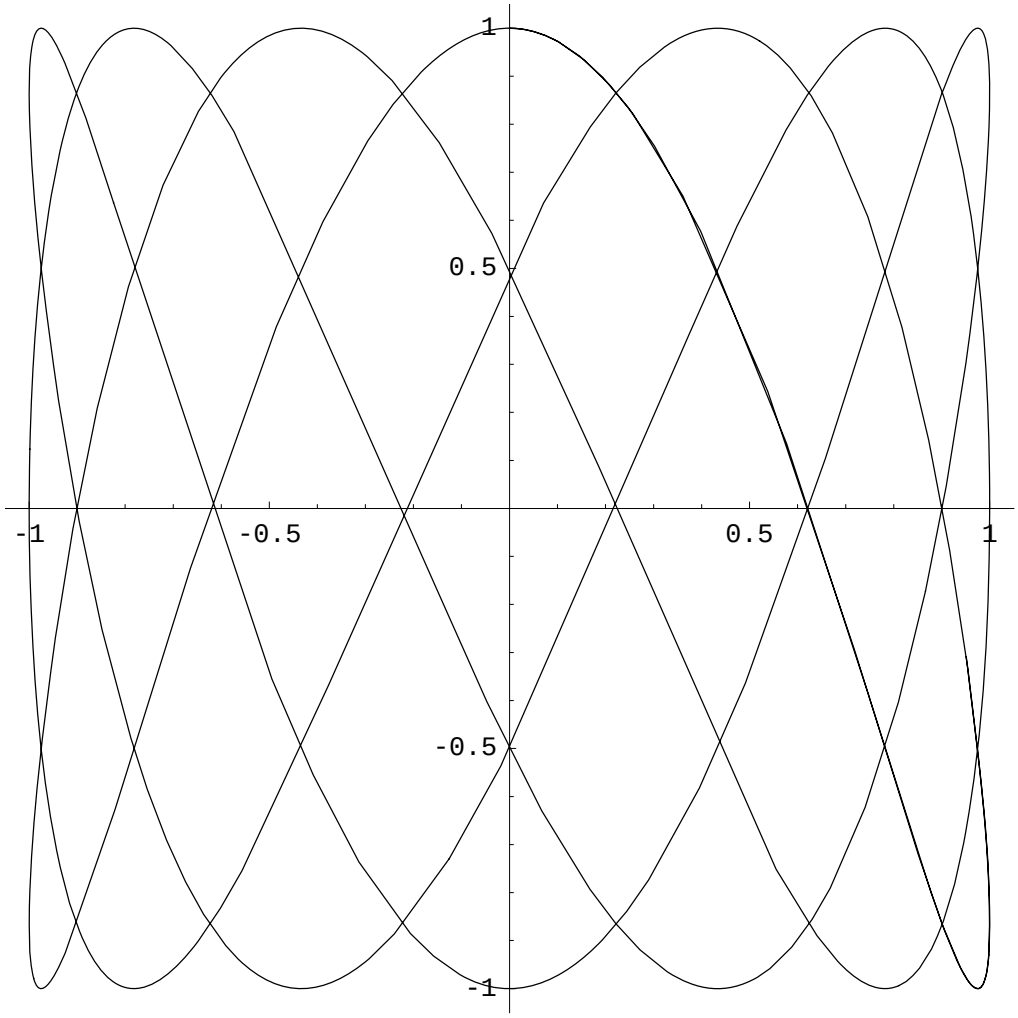
In[65]:=
```
lissajouis = ParametricPlot[{Sin[3 x], Cos[7 x]}, {x, 0, 2.2 Pi}]
```



Out[65]=
```
- Graphics -
```

Potete vedere come abbiamo definito la lista delle due funzioni, e poi si siano disegnati i punti e la funzione mediante il parametro *x*: dalla scala del grafico possiamo vedere come, in realtà, il rapporto non sia 1:1. Possiamo correggerlo o ridisegnando la curva, oppure usando Show:

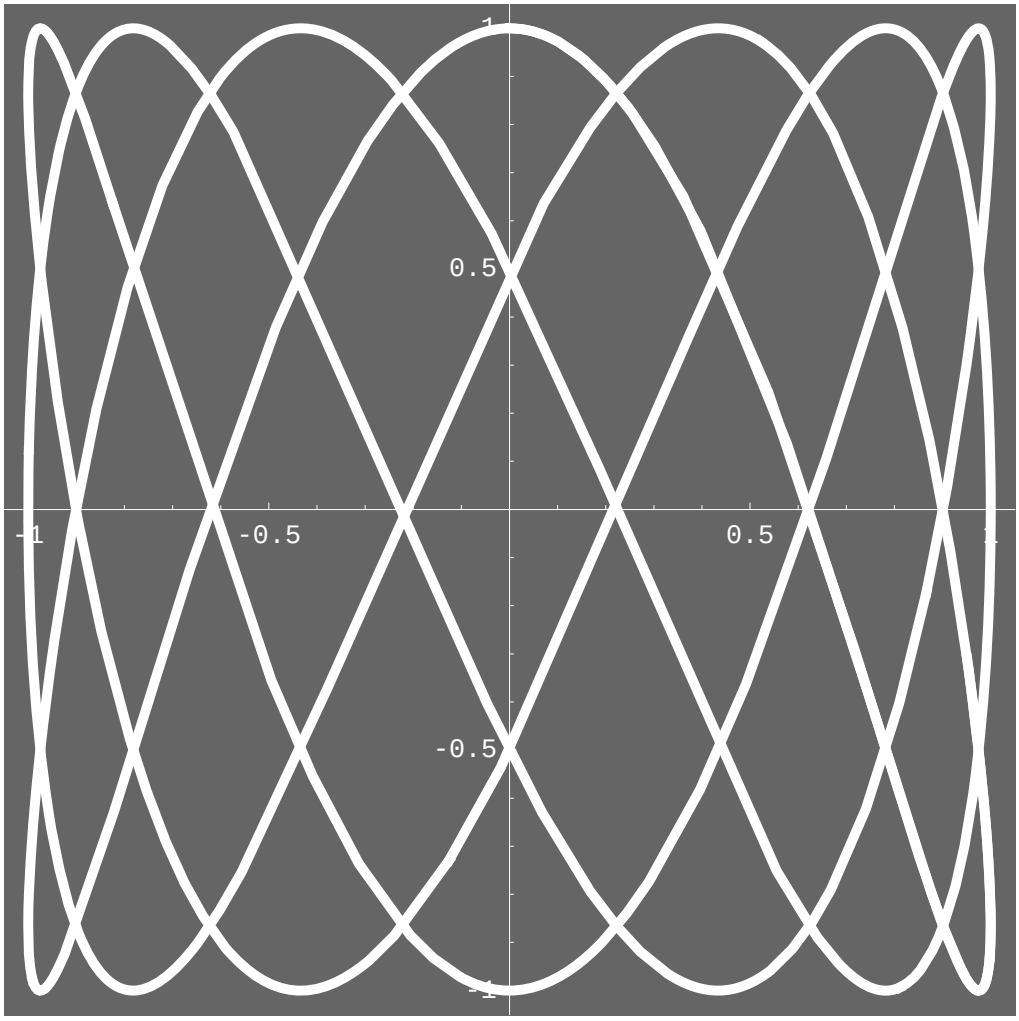
In[66]:= **Show[%, AspectRatio → Automatic]**



Out[66]= **- Graphics -**

Vediamo come ora le proporzioni siano corrette: dato che in questo caso entrambi gli assi sono funzione di un parametro, la scala uguale o definita da voi in modo specifio può essere di maggiore aiuto rispetto al caso di funzione normale, per capirne l'andamento e quello che sta dietro ad esso. Anche in questo caso possiamo formattare la funzione come più ci aggrada:

In[67]:=
```
ParametricPlot[{Sin[3 x], Cos[7 x]}, {x, 0, 2.2 Pi},
 Background → GrayLevel[0.4],
 PlotStyle → Thickness[.01],
 AspectRatio → Automatic
]
```

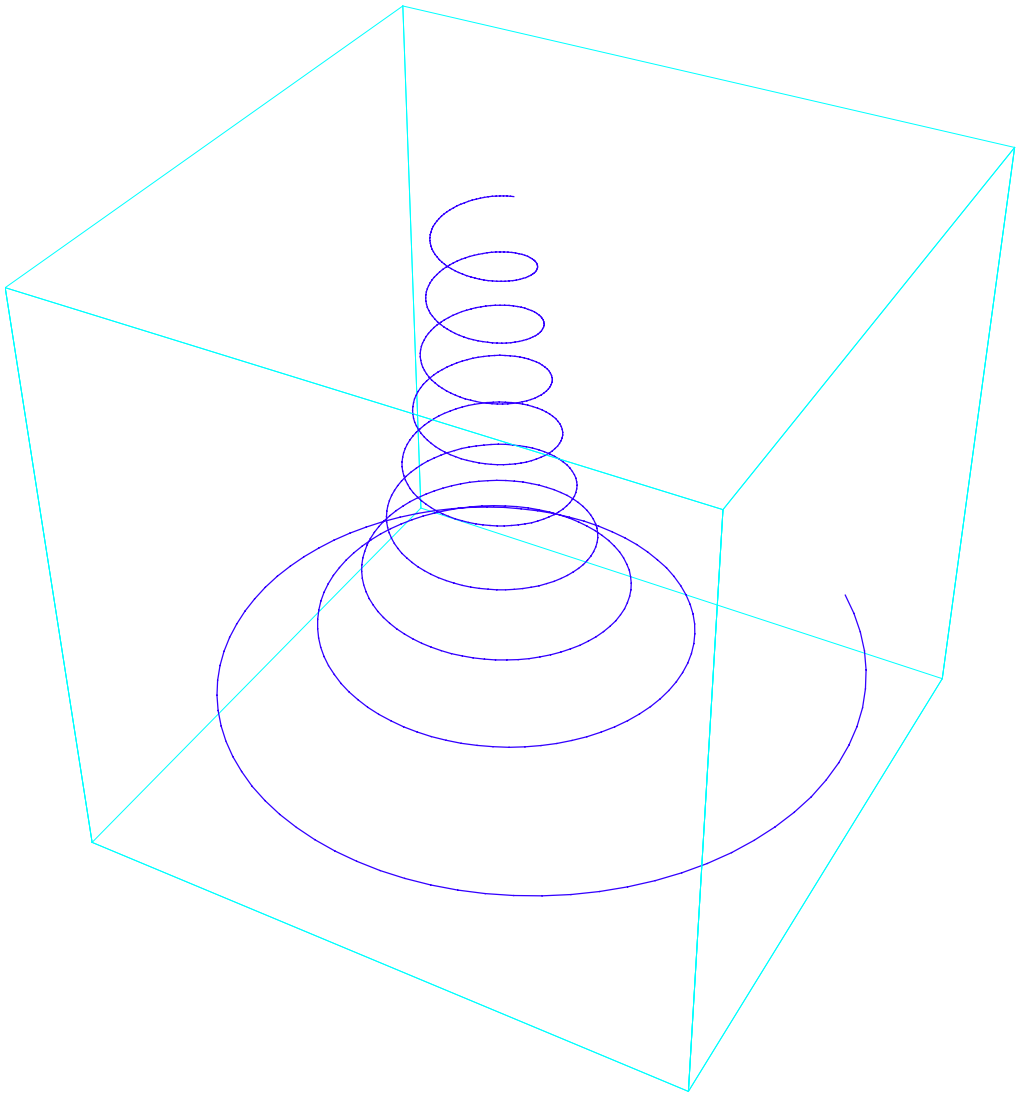


Out[67]= - Graphics -

Ormai siete talmente bravi e 'pumatusi' da non dovervi spiegare come abbia ottenuto il risultato, vero? Scommetto che lo sapete fare anche ad occhi chiusi, mangiando una pizza con una mano e con il computer alle spalle, come ho fatto io...

Per i grafici parametridi 3D le cose sono molto simili, considerando anche il fatto che ci sono due tipi di grafici: le curve parametriche e le superfici parametriche. La differenza principale consiste nel fatto che nel primo caso occorre soltanto un parametro, mentre nel secondo ne occorrono (indovinate?) due: possiamo definire, per cominciare, una curva parametrica:

In[68]:=
```
ParametricPlot3D[{Sin[7 x] / x, Cos[7 x] / x, x / 6}, {x, 1, 9},
 PlotPoints → 700,
 DefaultColor → Hue[.7],
 BoxStyle → RGBColor[0, 1, 1],
 Axes → None
]
```

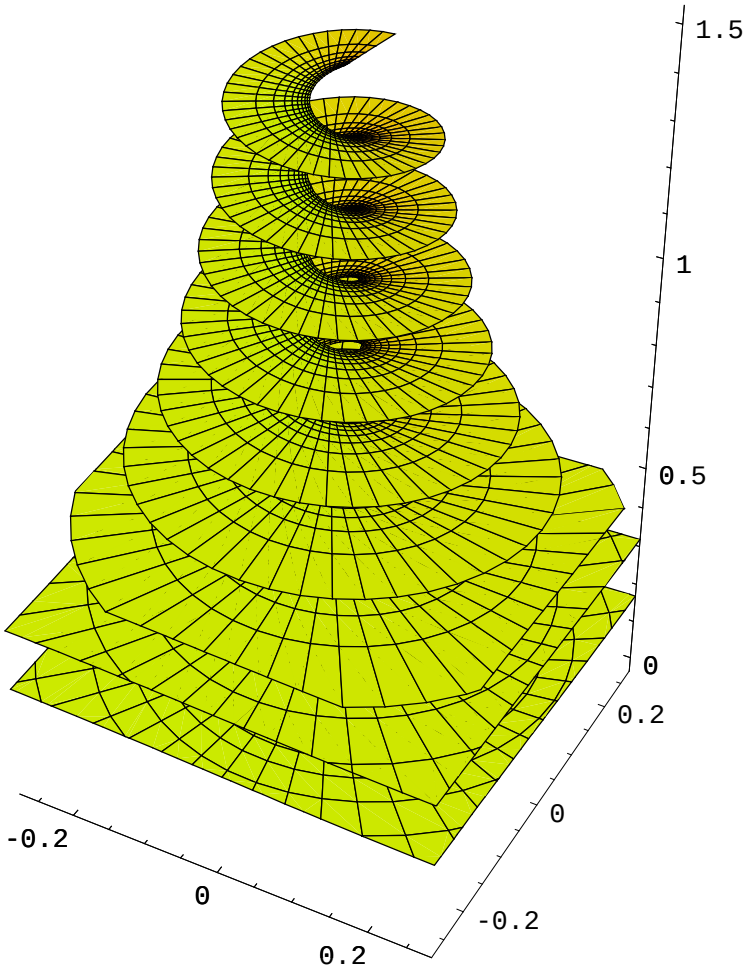


Out[68]= - Graphics3D -

Notate come abbia anche applicato una formattazione: ormai i grafici puramente neri non sono più così attraenti, una volta imparate le opzioni!!!

Se, invece, abbiamo bisogno di disegnare una superfice, dobbiamo considerare anche l'altro parametro, e da curva si ottiene una superfice:

In[69]:=
```
ParametricPlot3D[
 {Sin[7 x] / (x y), Cos[7 x] / (x y), x / 6}, {x, 1, 9}, {y, 1, 5},
 PlotPoints → {400, 10},
 BoxRatios → {1, 1, 1.7},
 LightSources → {{{-3, 4, 3}, Hue[0.1]}, {{3, -3, 4}, Hue[0.3]}},
 Boxed → False,
 AxesEdge → {{-1, -1}, {1, -1}, {1, 1}}
]
```



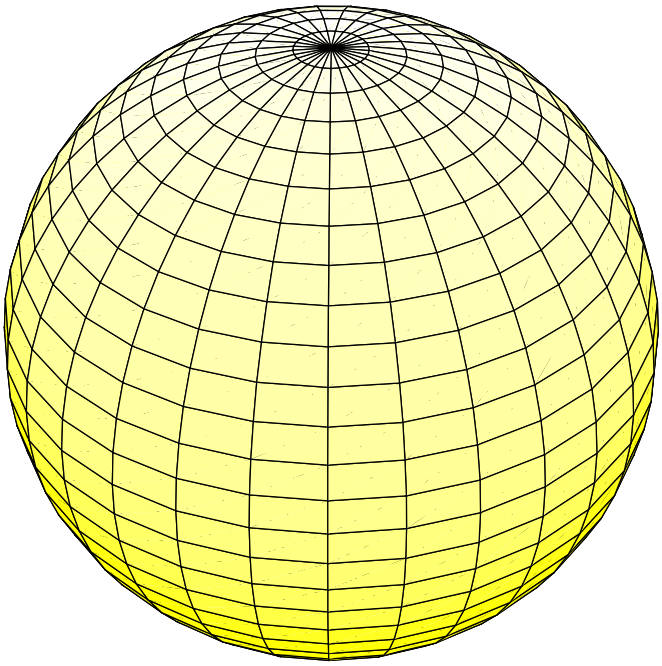
Out[69]= - Graphics3D -

Come possiamo vedere, abbiamo ottenuto la curva parametrica che volevamo in quattro e quattr'otto (d'altronde, la matematica non è un opinione). Quello che succede nel disegno di curve parametriche è il seguente: si fissa il secondo parametro, in questo caso *y*, e si disegna la curva parametrica corrispondente in *x*: dopo averla disegnata, si incrementa il valore *y*, e si disegna la seconda curva parametrica corrispondente. A questo punto, si uniscono i punti aventi lo stesso

parametro *x*, per avere una striscia di poligoni, e poi si procede ad oltranza, fino ad avere la rappresentazione completa della curva. Questo è il modo di disegnarle, semplice ed efficace. Notate anche come abbia dovuto incrementare il numero di punti per avere una rappresentazione dettagliata, e come, dato che ho bisogno di due parametri, i valori dei punti di disegno in PlotPoints sia definito da una lista di due valori, uno per il primo parametro, e l'altro per il secondo.

Possiamo parametrizzare anche i solidi, come, per esempio, la classica e sottovalutata sfera:

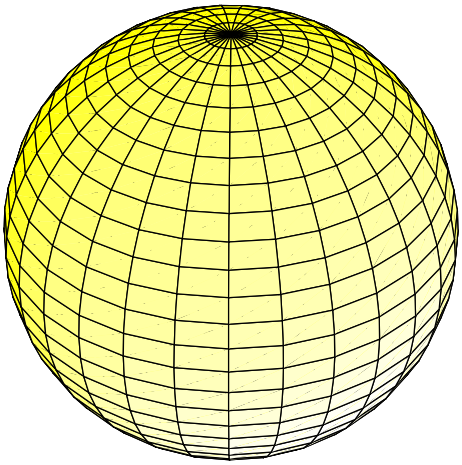
In[70]:=
```
sfera = ParametricPlot3D[{Cos[x] Cos[y], Sin[x] Cos[y], Sin[y]},
  {x, 0, 2 Pi}, {y, -Pi / 2, Pi / 2},
  Boxed → False,
  Axes → None,
  AmbientLight → Yellow
 ]
```

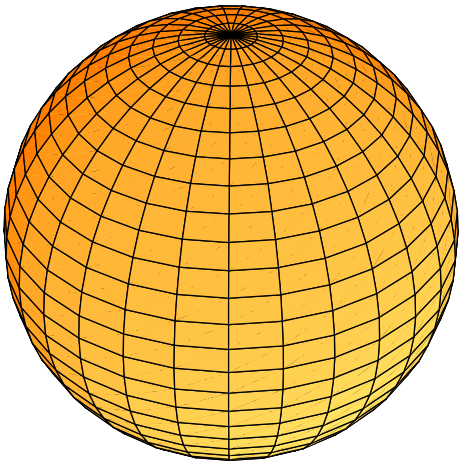


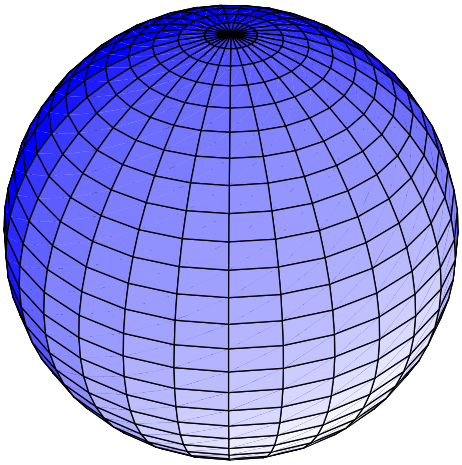
Out[70]= - Graphics3D -

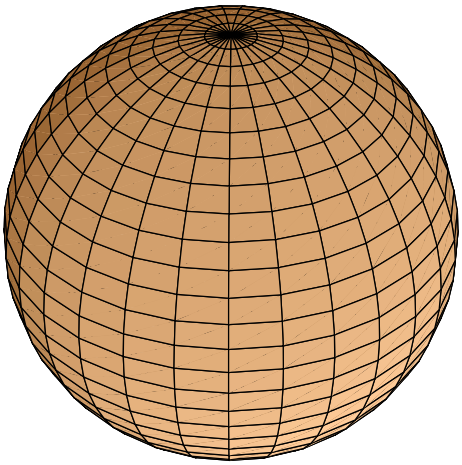
Qua potete notare un'altra cosuccia interessante, fra le opzioni: riuscite a vederla? *Mathematica* ha nel suo delicato e complicato pancino delle costanti con nomi di colore, che rappresentano appunto il colore specificato, senza doverlo andare a scrivere come funzione di Hue oppure di RGBColor. Sono presenti tutti i colori più comuni, considerando sempre i loro nomi inglesi.

In[71]:=
```
sfera1 = ParametricPlot3D[{Cos[x] Cos[y], Sin[x] Cos[y], Sin[y]},
   {x, 0, 2 Pi}, {y, -Pi / 2, Pi / 2},
   Boxed → False,
   Axes → None,
   AmbientLight → Yellow,
   LightSources → {{{1.3, -2.4, 2.}, RGBColor[1, 1, 1]}}
  ];

sfera2 =
  sfera = ParametricPlot3D[{Cos[x] Cos[y], Sin[x] Cos[y], Sin[y]},
    {x, 0, 2 Pi}, {y, -Pi / 2, Pi / 2},
    Boxed → False,
    Axes → None,
    AmbientLight → Orange,
    LightSources → {{{1.3, -2.4, 2.}, RGBColor[.4, .4, .4]}}
   ];

sfera3 =
  sfera = ParametricPlot3D[{Cos[x] Cos[y], Sin[x] Cos[y], Sin[y]},
    {x, 0, 2 Pi}, {y, -Pi / 2, Pi / 2},
    Boxed → False,
    Axes → None,
    AmbientLight → Blue,
    LightSources → {{{1.3, -2.4, 2.}, RGBColor[1, 1, 1]}}
   ];

sfera4 =
  sfera = ParametricPlot3D[{Cos[x] Cos[y], Sin[x] Cos[y], Sin[y]},
    {x, 0, 2 Pi}, {y, -Pi / 2, Pi / 2},
    Boxed → False,
    Axes → None,
    AmbientLight → Brown,
    LightSources → {{{1.3, -2.4, 2.}, RGBColor[.4, .4, .4]}}
   ];
```

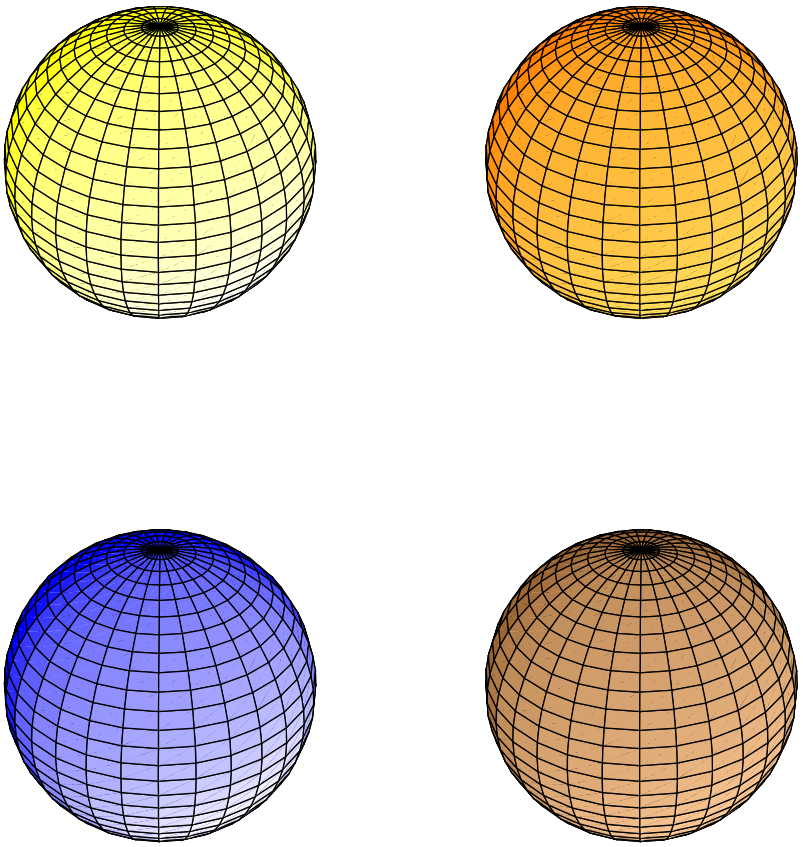








In[75]:=
```
Show[GraphicsArray[{{sfera1, sfera2}, {sfera3, sfera4}},
  GraphicsSpacing → 0]]
```



Out[75]=
```
- GraphicsArray -
```

Vediamo qua le sfere colorate in modo diverso dall'AmbientLight. Notate come, per far notare meglio il colore, sia stato necessario ridefinire le luci di default di *Mathematica*, che sono colorate e quindi falsificavano il vero colore della luce ambiente. Le figure, di default, non hanno colorazione di superfice, per cui in questo caso il colore effettivo visualizzato corrisponde a quello della luce ambiente. Non vi ho spiegato prima GraphicsArray, ma credo che non ci sia bisogno di spiegazioni. D'altronde, come vi ho detto innumerevoli volte, il manuale on-line è indispensabile, quindi leggetelo e, ogni volta che lo farete, scoprirete sempre cose nuove... Altro che "La Storia Infinita"!!!

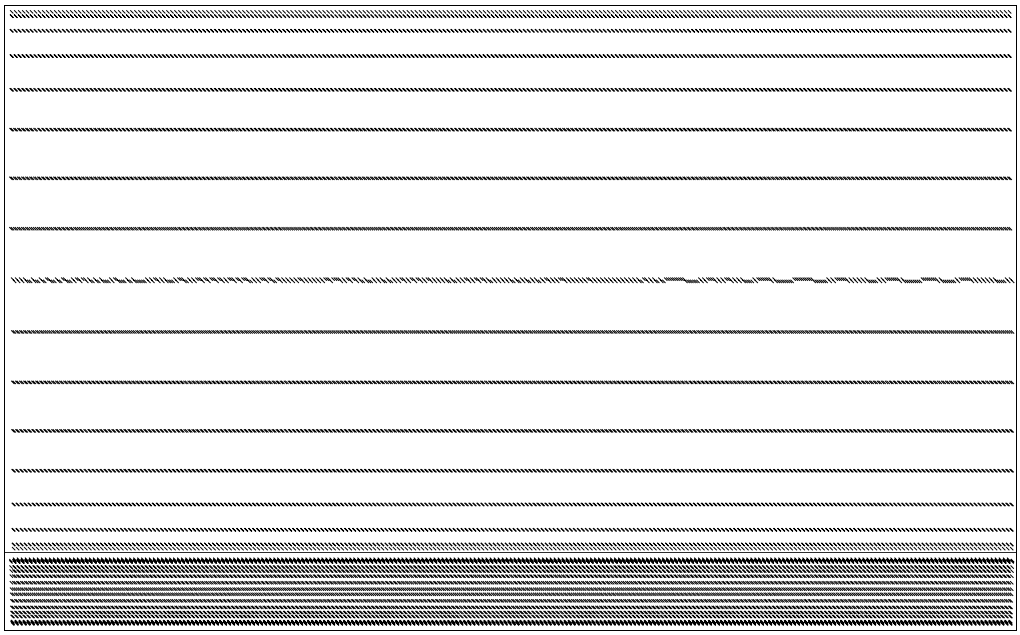
Suoni

La parte riguardante i suoni, lo ammetto, è una di quelle che conosco meno in assoluto di *Mathematica*, perchè personalmente non ho mai avuto modo di usarla. Quindi, quello che dirò saranno solamente le nozioni base. Prima di tutto, come per il caso dei grafici, possiamo decidere se creare un suono da una funzione matematica, oppure da una lista di dati campionati. Per i due casi, ci sono funzioni distinte, anche se simili:

`Play[`f`, {`t`, 0, `t_{max}`}]`	genera un suono di ampiezza f come funzione del tempo t espresso in secondi
`ListPlay[{`a_1`, `a_2`, … }, SampleRate ->` r`]`	genera un suono a partire dalla lista di dati campionati, con il Rate specificato

Possiamo molto facilmente definire delle armoniche, usando le funzioni sinusoidali:

In[76]:=
```
Play[Sin[500 Pi t], {t, 0, 1}]
```

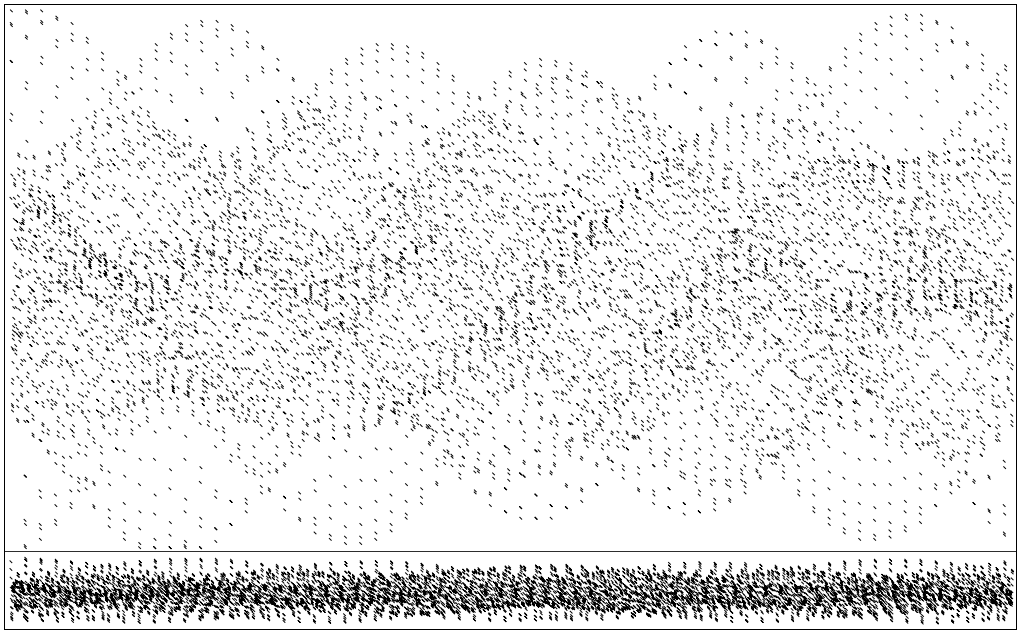


Out[76]=
```
- Sound -
```

Possiamo vedere come, oltre al suono (che naturalmente non potete sentire, ma se eseguite il comando lo sentirete), viene rappresentato anche un grafico con il campionamento del suono. Notate come, nell'indicatore di cella a lato, compare una piccola frecca nella parte superiore. In questo modo, facendo doppio clic sulla freccia, allora il suono riparte, senza bisogno di dover rieseguire il comando.

Possiamo anche definire deli suoni più complicati come, in questo caso, l'accordo di do:

In[77]:=
```
Play[
 Sin[526 π t] +
  Sin[526 2^(1/3) π t] +
  Sin[526 2^(7/12) π t] +
  Sin[1052 π t],
 {t, 0, 1}
]
```



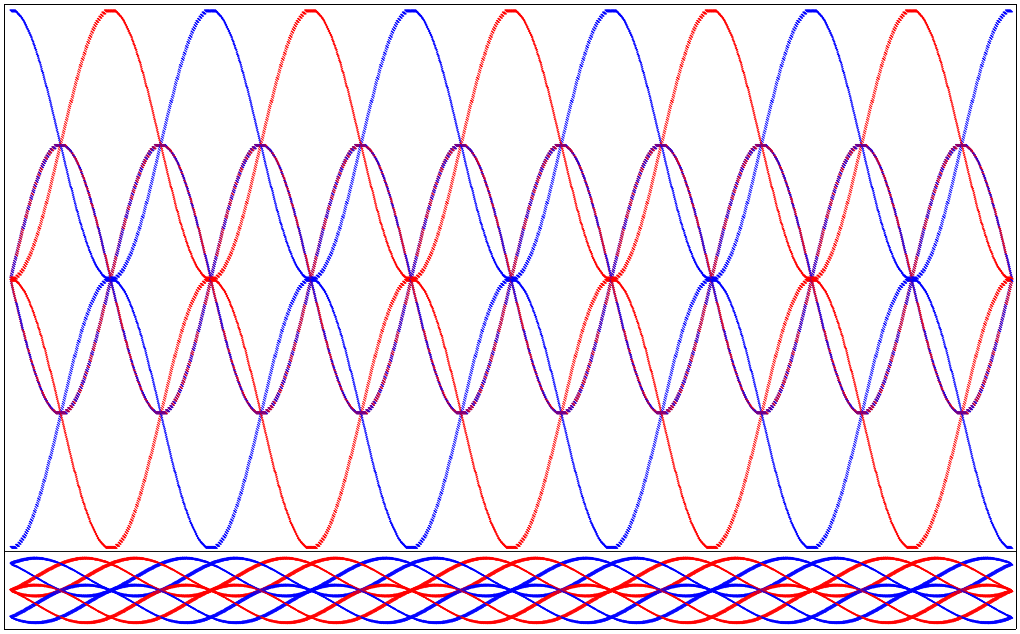
Out[77]= - Sound -

Un altro aspetto interessante è il fatto che *Mathematica* è in grado anche di generare audio multicanale, generando una lista di funzione, una per ogni canale:

In[78]:=
```
suono1[t_] := Sin[4000 Pi t] + Sin[3990 Pi t]
```

In[79]:=
```
suono2[t_] := Sin[4000 Pi t] + Sin[3990 Pi t + Pi]
```

In[80]:=
```
Play[{suono1[t], suono2[t]}, {t, 0, 1}]
```



Out[80]= - Sound -

Possiamo vedere come, in questo caso, oltre al campionamento si vede che ogni singolo canale è rappresentato da un colore diverso. Non ho sinceramente provato il suono con più canali, ma magari voi siete tanto gentili da provare e farmi sapere, vero? Chi lo sa se con gli opportuni comandi *Mathematica* è pure in grado di gestire suono surround...