New Plotting Framework for SymPy

Structure of the Module

This module implements a new plotting framework for SymPy. The central class of the module is the `Plot` class that connects the data representations (subclasses of `BaseSeries`) with different plotting backends. It's not imported by default for backward compatibility with the old module.

Then there is the `plot()` function that has a less stricter requirements for its input and is better suited for interactive work.

Docstrings

In [1]:
```
help(plot)
```

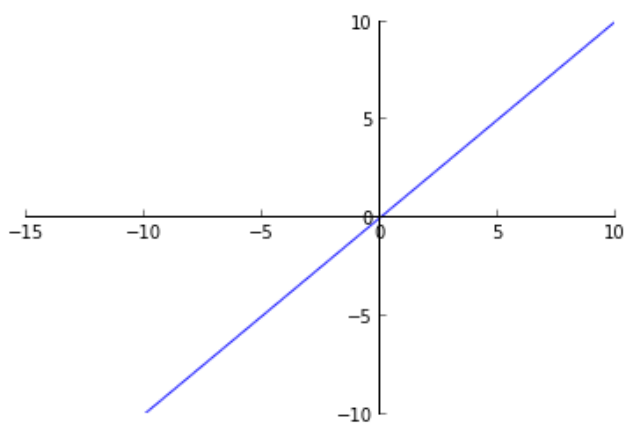
```
Help on function plot in module sympy.plotting.plot:

plot(*args, **kwargs)
    A plot function for interactive use.

    It implements many heuristics in order to guess what the user wants on
    incomplete input.

    There is also the 'show' argument that defaults to True (immediately
    showing the plot).

    The input arguments can be:
      - lists with coordinates for ploting a line in 2D or 3D
      - the expressions and variable lists with ranges in order to plot any of
        the following: 2d line, 2d parametric line, 3d parametric line,
        surface, parametric surface
         - if the variable lists do not provide ranges a default range is used
         - if the variables are not provided, the free variables are
           automatically detected
         - if neither variables nor ranges are provided, both are guessed
         - if multiple expressions are provided in a list all of them are
           plotted
      - an instance of BaseSeries() subclass
      - another Plot() instance
      - tuples containing any of the above mentioned options, for plotting them
        together

    Examples:
    ---------

    Plot expressions:
    >>> from sympy import plot, cos, sin, symbols
    >>> x,y,u,v = symbols('x y u v')
    >>> p1 = plot(x**2, show=False) # with default [-10,+10] range
    >>> p2 = plot(x**2, (0, 5), show=False) # it finds the free variable itself
    >>> p3 = plot(x**2, (x, 0, 5), show=False) # fully explicit

    Fully implicit examples (finding the free variable and using default
    range). For the explicit versions just add the tuples with ranges:
    >>> p4 = plot(x**2, show=False) # cartesian line
    >>> p5 = plot(cos(u), sin(u), show=False) # parametric line
    >>> p6 = plot(cos(u), sin(u), u, show=False) # parametric line in 3d
    >>> p7 = plot(x**2 + y**2, show=False) # cartesian surface
    >>> p8 = plot(u, v, u+v, show=False) # parametric surface

    Multiple plots per figure:
    >>> p9 = plot((x**2, ), (cos(u), sin(u)), show=False) # cartesian and
parametric lines

    Set title or other options:
    >>> p10 = plot(x**2, title='second order polynomial', show=False)

    Plot a list of expressions:
    >>> p11 = plot([x, x**2, x**3], show=False)
    >>> p12 = plot([x, x**2, x**3], (0,2), show=False) # explicit range
    >>> p13 = plot([x*y, -x*y], show=False) # list of surfaces

    And you can even plot a Plot or a Series object:
    >>> a = plot(x, show=False)
    >>> p14 = plot(a, show=False) # plotting a plot object
```

In [2]:
```
help(Plot) # This is from the old module!
```

```
Help on function Plot in module sympy.plotting.proxy_pyglet:

Plot(*args, **kwargs)
    A temporary proxy for an interface under deprecation.

    This proxy is the one imported by `from sympy import *`.

    The Plot class will change in future versions of sympy to use the new
    plotting module. That new plotting module is already used by the
    plot() function (lowercase). To write code compatible with future versions
    of sympy use that function (plot() lowercase). Or if you want to use the
    old plotting module just import it directly:
    `from sympy.plotting.pygletplot import PygletPlot`

    To use Plot from the new plotting module do:
    `from sympy.plotting.plot import Plot`

    In future version of sympy you will also be able to use
    `from sympy.plotting import Plot` but in the current version this will
    import this proxy object. It's done for backward compatibility.

    The old plotting module is not deprecated. Only the location will
    change. The new location is sympy.plotting.pygletplot.
```

General examples

In [3]:
```
p = plot(x)
```



In [4]:
```
p # the Plot object
```

Out[4]:
```
                    Plot object containing:
[0]: cartesian line: x for x over (-10.0, 10.0)
```

In [5]:
```
p[0] # one of the data series objects
```

Out[5]:
```
cartesian line: x for x over (-10.0, 10.0)
```

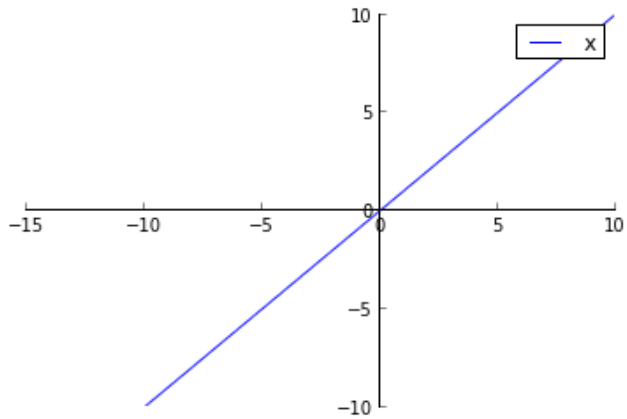
In [6]:
```
p[0].label # an option of the data series
```

Out[6]: x

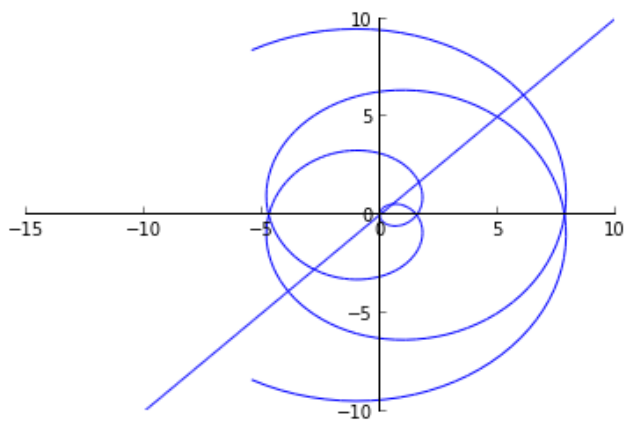
In [7]:
```
p.legend # a global option of the plot
```

Out[7]: $False$

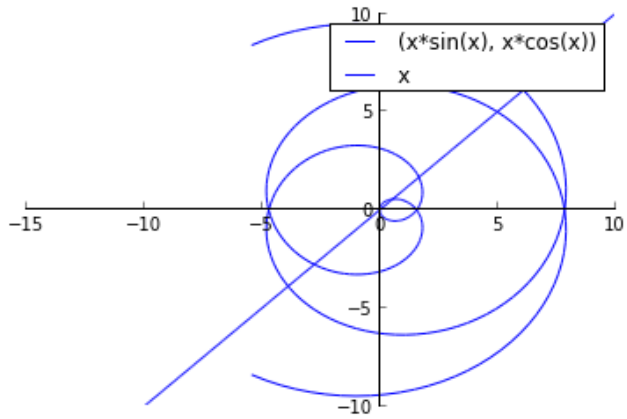
In [8]:
```
p.legend = True
p.show()
```



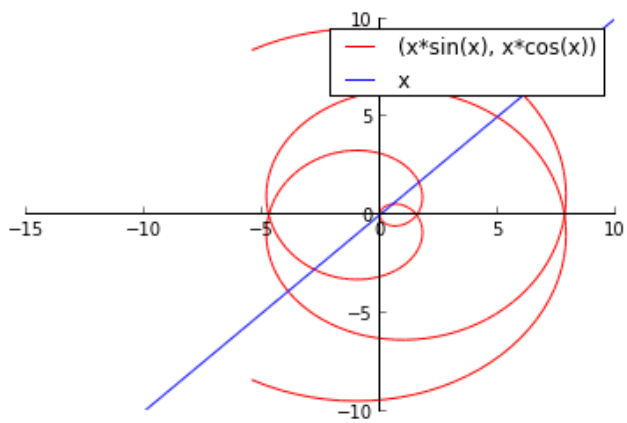
In [9]:
```
p1 = plot(x*sin(x),x*cos(x), show=False)
p1.extend(p) # Plot objects are just like lists.
p1.show()
```



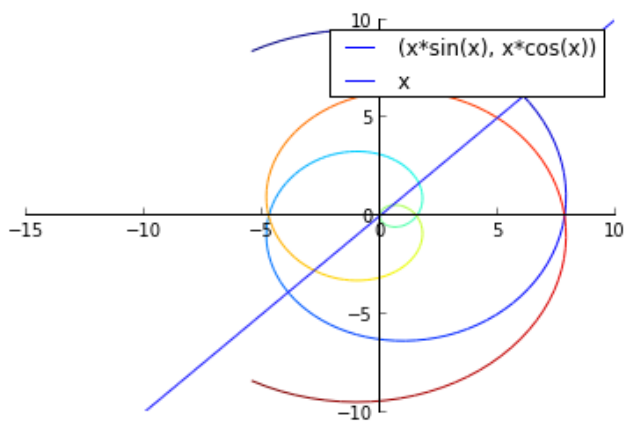
In [10]:
```
p1.legend = True
p1.show()
```



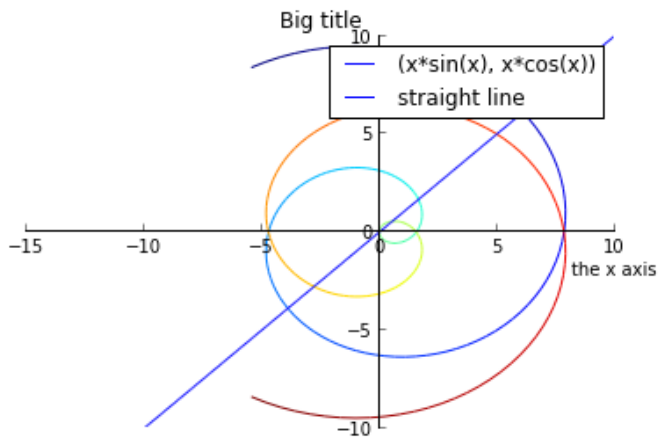
In [11]:
```
p1[0].line_color='r'
p1[1].line_color='b' # a constant color
p1.show()
```



In [12]:
```
p1[0].line_color = lambda a : a # color dependent on the parameter
p1.show()
```



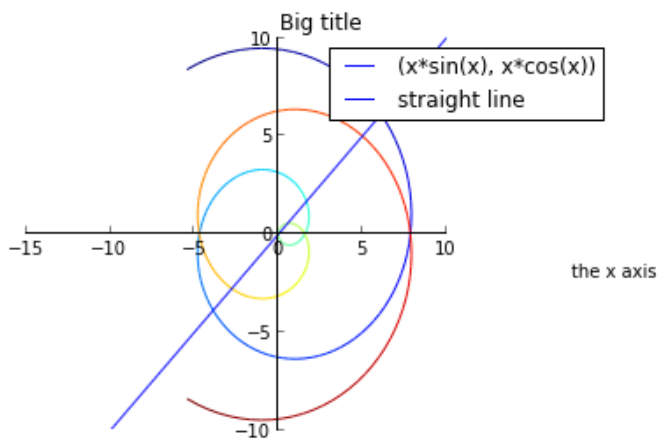
In [13]:
```
p1.title = 'Big title'
p1.xlabel = 'the x axis'
p1[1].label = 'straight line'
p1.show()
```



In [14]:
```
p1.aspect_ratio
```

Out[14]: auto

In [15]:
```
p1.aspect_ratio = (1,1)
p1.xlim = (-15,20)
p1.show()
```



Hm, `xlim` does not work in the notebook. Hopefully it works in IPython.

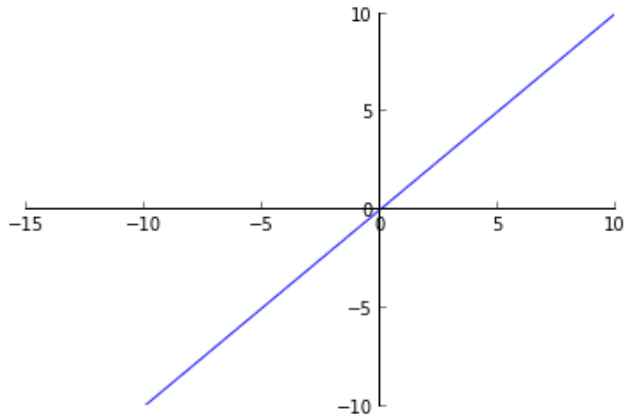
In [16]:
```
p1._backend.ax.get_xlim()
```

Out[16]: *[Math Processing Error]*

Yeah, the backend got the command, but the `inline` backend does not honour it.

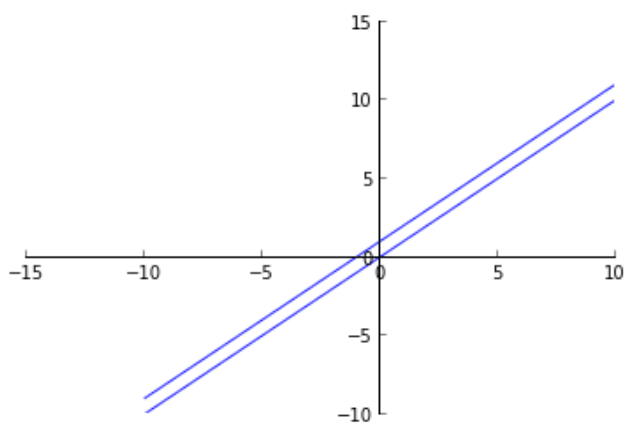
Adding expressions to a plot

In [17]:
```
p = plot(x)
p
```



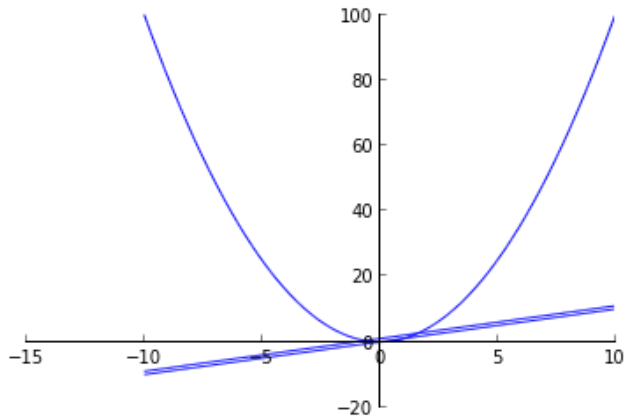
Out[17]:
```
                    Plot object containing:
[0]: cartesian line: x for x over (-10.0, 10.0)
```

In [18]:
```
p.extend(plot(x+1, show=False))
p.show()
p
```



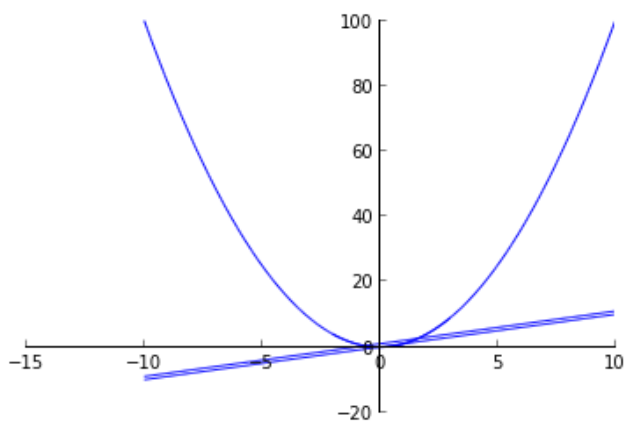
Out[18]:
```
                    Plot object containing:
  [0]: cartesian line: x for x over (-10.0, 10.0)
[1]: cartesian line: x + 1 for x over (-10.0, 10.0)
```

In [19]:
```
p.append(plot((x+3,),(x**2,), show=False)[1])
p.show()
p
```



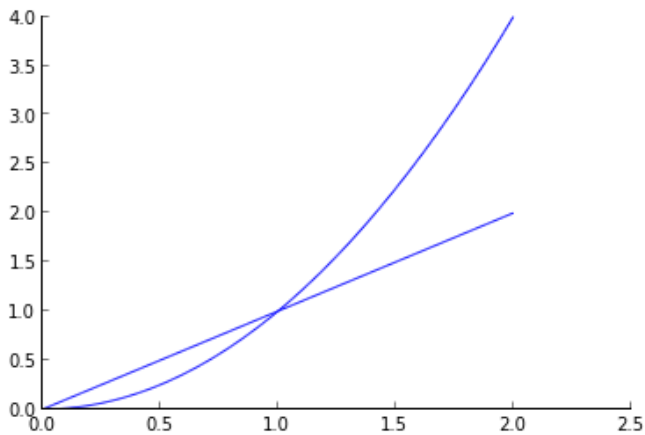
Out[19]:
```
                    Plot object containing:
  [0]: cartesian line: x for x over (-10.0, 10.0)
[1]: cartesian line: x + 1 for x over (-10.0, 10.0)
 [2]: cartesian line: x**2 for x over (-10.0, 10.0)
```

In [20]:
```
p[2] = x**2, (x, -2, 3)  # you must be explicit when not using plot()
p.show() # and call show() yourself
```

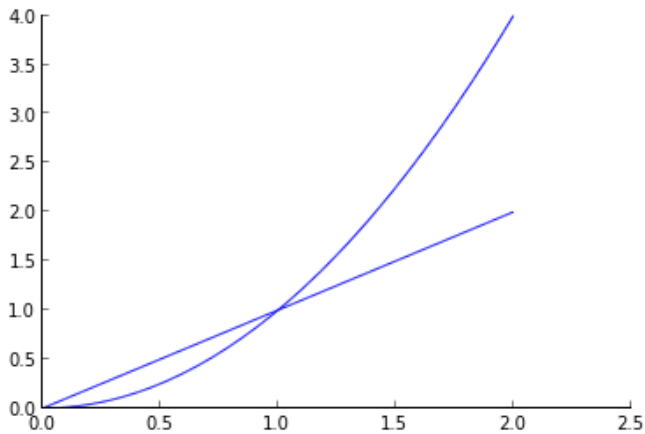


Or even sending a `Plot` or a `Series` object to the `plot` function.

In [21]:
```
a = plot((x,(0,2)), (x**2,(0,2)))
```

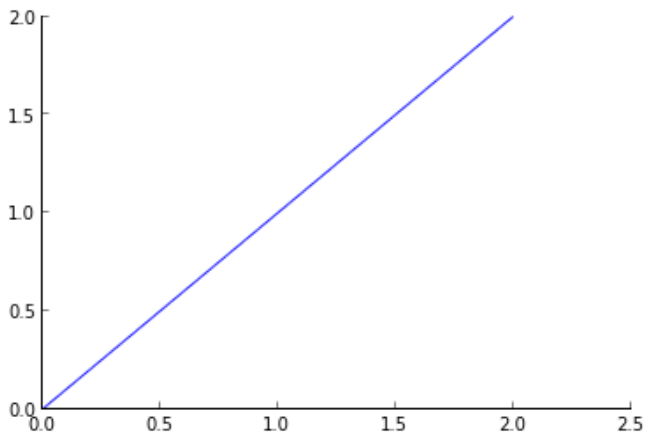


In [22]:
```
plot(a)
```



Out[22]:
```
                    Plot object containing:
  [0]: cartesian line: x for x over (0.0, 2.0)
[1]: cartesian line: x**2 for x over (0.0, 2.0)
```

In [23]:
```
plot(a[0])
```



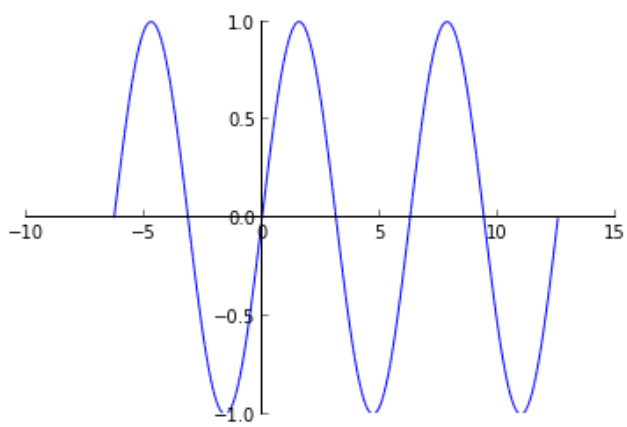
Out[23]:
```
                Plot object containing:
[0]: cartesian line: x for x over (0.0, 2.0)
```

Ambiguous input in `plot()`

`plot()` is capable of working with more ambiguous input than `Plot()`. The later needs explicit free variables and range while the first finds the free variables and has default ranges.

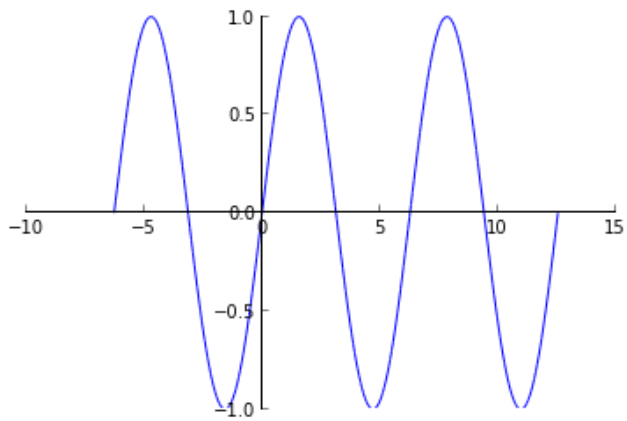
The number of expression determines the type of the plot. The arguments to `plot()` can also be tuples, every tuples containing a new expression to be plotted.

In [24]:
```
plot(sin(x),(x,-2*pi,4*pi)) # explicit 2D line
```



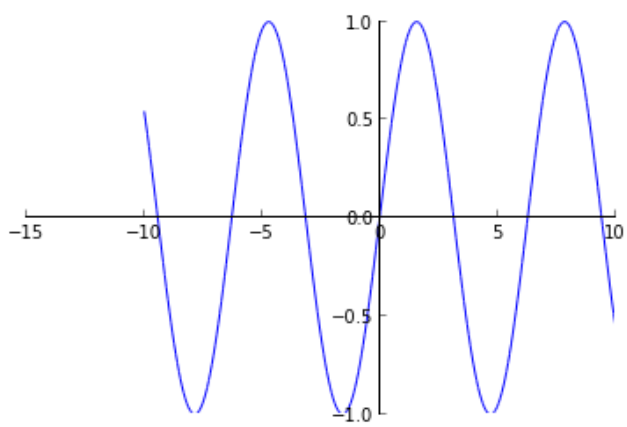
Out[24]:
```
                                Plot object containing:
[0]: cartesian line: sin(x) for x over (-6.283185307179586, 12.566370614359172


)
```

In [25]:
```
plot(sin(x),(-2*pi,4*pi)) # implicit free variable
```



Out[25]:
```
                                Plot object containing:
[0]: cartesian line: sin(x) for x over (-6.283185307179586, 12.566370614359172


)
```

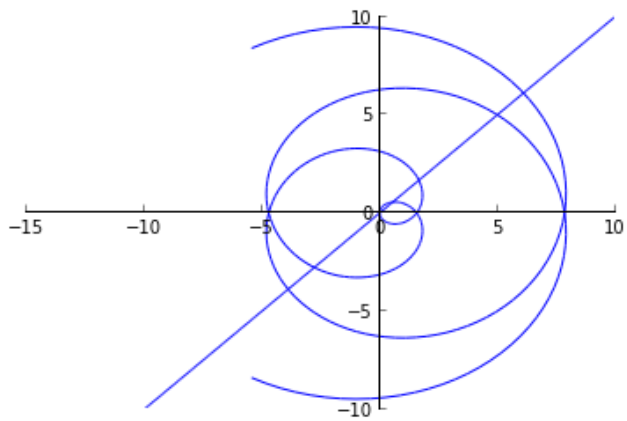
In [26]:
```
plot(sin(x)) # and default range
```



Out[26]:
```
                Plot object containing:
[0]: cartesian line: sin(x) for x over (-10.0, 10.0)
```

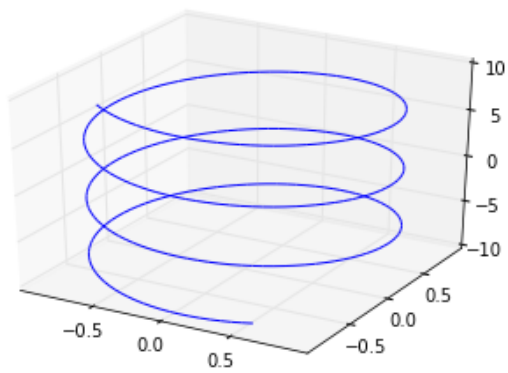
Different types of plots

In [27]:
```
plot((x,), (x*sin(x), x*cos(x))) # cartesian line and a parametric line
```



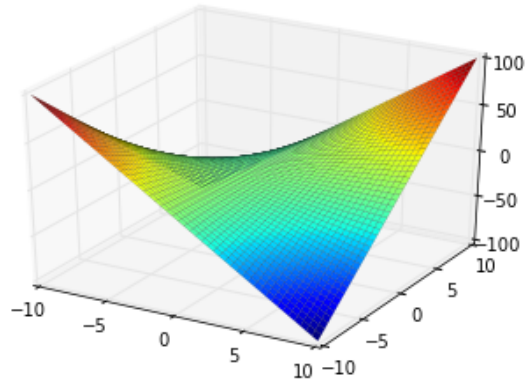
Out[27]:
```
                    Plot object containing:
[0]: cartesian line: x for x over (-10.0, 10.0)
[1]: parametric cartesian line: (x*sin(x), x*cos(x)) for x over (-10.0, 10.0)
```

In [28]:
```
plot(sin(x),cos(x),x) # 3D parametric line
```



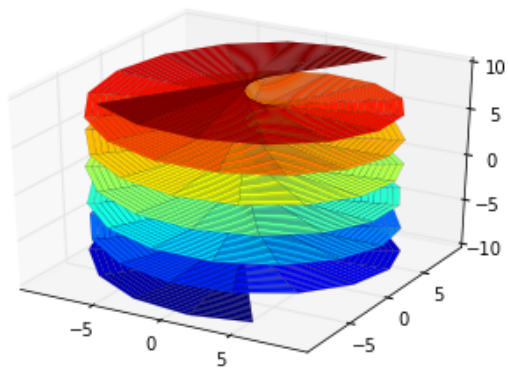
Out[28]:
```
                    Plot object containing:
[0]: 3D parametric cartesian line: (sin(x), cos(x), x) for x over (-10.0, 10.0


)
```

In [29]:
```
plot(x*y) # a surface
```



Out[29]:
```
                                    Plot object containing:
[0]: cartesian surface: x*y for x over (-10.0, 10.0) and y over (-10.0, 10.0)
```

In [30]:
```
plot(x*sin(z),x*cos(z),z) # parametric surface
```

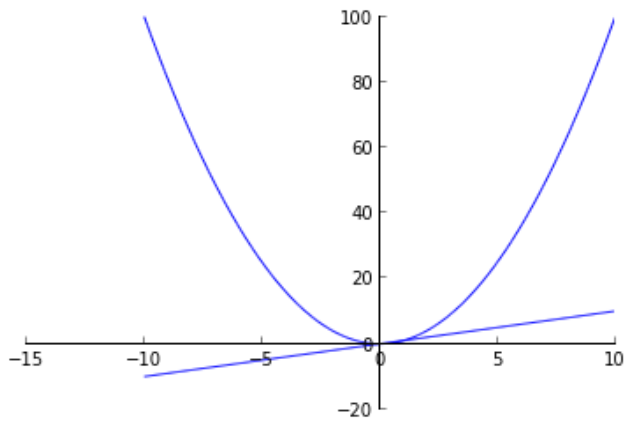


Out[30]:
```
                                                    Plot object containing:
[0]: parametric cartesian surface: (x*sin(z), x*cos(z), z) for x over (-10.0,

10.0) and z over (-10.0, 10.0)
```

List of expressions as an argument

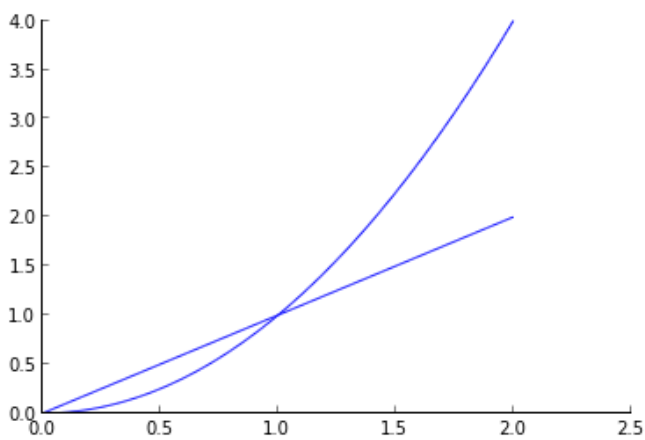
Especially useful when plotting the output of `solve`.

In [31]:
```
plot([x,x**2])
```



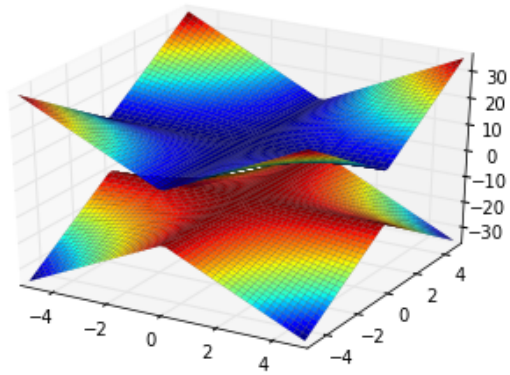
Out[31]:
```
                    Plot object containing:
[0]: cartesian line: x for x over (-10.0, 10.0)
[1]: cartesian line: x**2 for x over (-10.0, 10.0)
```

In [32]:
```
plot([x,x**2], (0,2))
```



Out[32]:
```
                    Plot object containing:
[0]: cartesian line: x for x over (0.0, 2.0)
[1]: cartesian line: x**2 for x over (0.0, 2.0)
```

In [33]:
```
plot([-abs(x*y)-10, abs(x*y)+10], (-5,5), (-5,5))
```



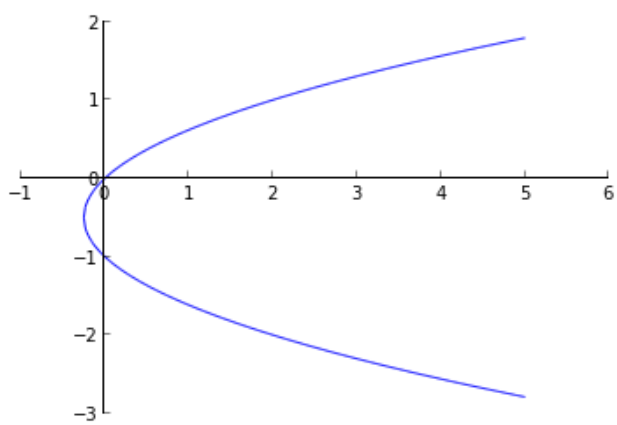
Out[33]:
```
                                    Plot object containing:
[0]: cartesian surface: -Abs(x*y) - 10 for x over (-5.0, 5.0) and y over (-5.0
[1]: cartesian surface: Abs(x*y) + 10 for x over (-5.0, 5.0) and y over (-5.0,


, 5.0)
 5.0)
```

In [34]:
```
solutions = solve(x**2+x-y,x)
```

In [35]:
```
solutions
```

Out[35]: *[Math Processing Error]*

In [36]:
```
plot(solutions, (-0.25,5))
```



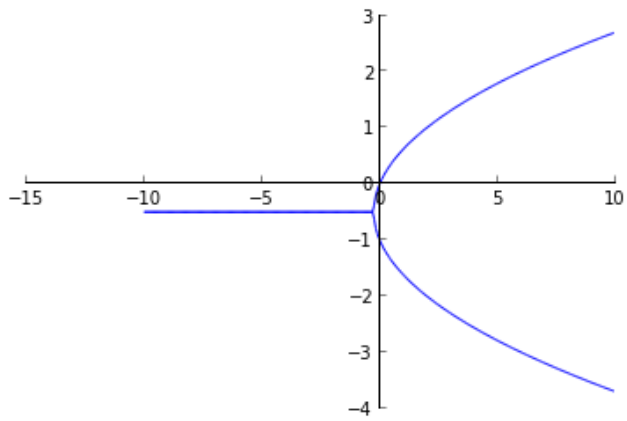
Out[36]:
```
                              Plot object containing:
[0]: cartesian line: -sqrt(4*y + 1)/2 - 1/2 for y over (-0.25, 5.0)
 [1]: cartesian line: sqrt(4*y + 1)/2 - 1/2 for y over (-0.25, 5.0)
```

Complex values

If complex values are encountered a warning is raised and only the real parts are plotted.

In [37]:
```
plot(solutions)
```

```
/home/stefan/scientific_python_stack/sympy/sympy/plotting
/experimental_lambdify.py:129: UserWarning: Complex values as arguments to numpy
functions encountered.
  warnings.warn('Complex values as arguments to numpy functions encountered.')
```



Out[37]:
```
                        Plot object containing:
[0]: cartesian line: -sqrt(4*y + 1)/2 - 1/2 for y over (-10.0, 10.0)
[1]: cartesian line: sqrt(4*y + 1)/2 - 1/2 for y over (-10.0, 10.0)
```

Textplot

There is also the textplot backend that permits plotting in the terminal.

In [38]:
```
pt = plot(sin(x),show=False)
```

In [39]:
```
pt.backend = plot_backends['text']
```

In [40]:
```
pt.show()
```

```
0.99935 |               ..               ...              . .
        |              .  .             .
        |                                   .            .   .
        |             .    .
        | .                            .                     .
        |                                  .          .
        |            .      .
        |  .                          .                       .
      0 | -----------------------------------.--------.---------
        |           .        .
        |   .                        .                          .
        |                                     .      .
        |          .          .
        |    .                      .
        |                                    .    .
        |     .   .             .
        |                            .             .  .
-0.9993 |      ...               ...               ..
          -10                     0                          10
```