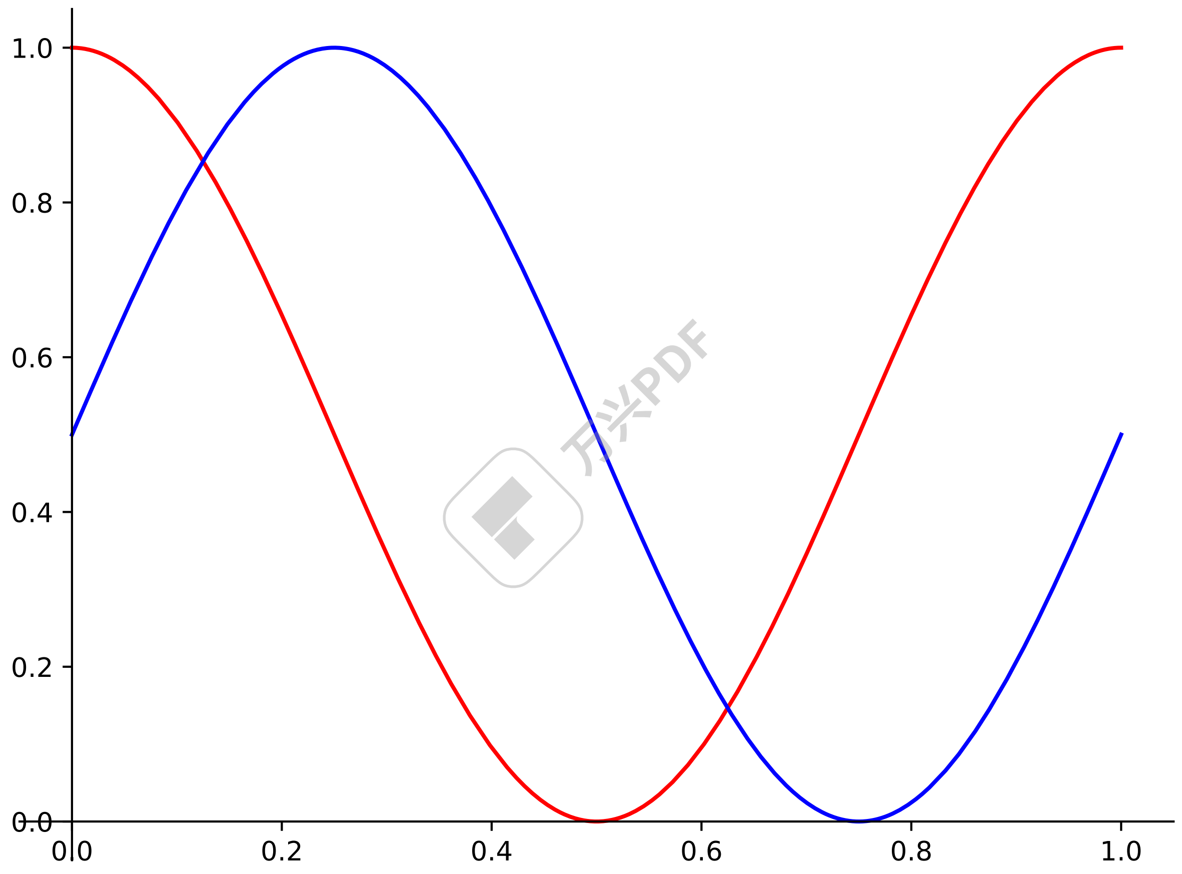

1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0

See also:

`Plot` (page 2821), `Parametric2DLineSeries` (page 2866)

`sympy.plotting.plot.`**`plot3d`**`(*args, show=True, **kwargs)`

Plots a 3D surface plot.

Usage

Single plot

`plot3d(expr, range_x, range_y, **kwargs)`

If the ranges are not specified, then a default range of (-10, 10) is used.

Multiple plot with the same range.

`plot3d(expr1, expr2, range_x, range_y, **kwargs)`

If the ranges are not specified, then a default range of (-10, 10) is used.

Multiple plots with different ranges.

`plot3d((expr1, range_x, range_y), (expr2, range_x, range_y), ..., **kwargs)`

Ranges have to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

Arguments

expr : Expression representing the function along x.

range_x
: [(`Symbol` (page 976), float, float)] A 3-tuple denoting the range of the x variable, e.g. (x, 0, 5).

range_y
: [(`Symbol` (page 976), float, float)] A 3-tuple denoting the range of the y variable, e.g. (y, 0, 5).

Keyword Arguments

Arguments for `SurfaceOver2DRangeSeries` class:

nb_of_points_x
: [int] The x range is sampled uniformly at `nb_of_points_x` of points.

nb_of_points_y
: [int] The y range is sampled uniformly at `nb_of_points_y` of points.

Aesthetics:

surface_color
: [Function which returns a float] Specifies the color for the surface of the plot. See `Plot` (page 2821) for more details.

If there are multiple plots, then the same series arguments are applied to all the plots. If you want to set these options separately, you can index the returned `Plot` object and set it.

Arguments for `Plot` class:

title
    [str] Title of the plot.

size
    [(float, float), optional] A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to `None`, meaning the size will be set by the default backend.

Examples

```
>>> from sympy import symbols
>>> from sympy.plotting import plot3d
>>> x, y = symbols('x y')
```

Single plot

```
>>> plot3d(x*y, (x, -5, 5), (y, -5, 5))
Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```

Multiple plots with same range

```
>>> plot3d(x*y, -x*y, (x, -5, 5), (y, -5, 5))
Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
[1]: cartesian surface: -x*y for x over (-5.0, 5.0) and y over (-5.0, 5.
↪0)
```

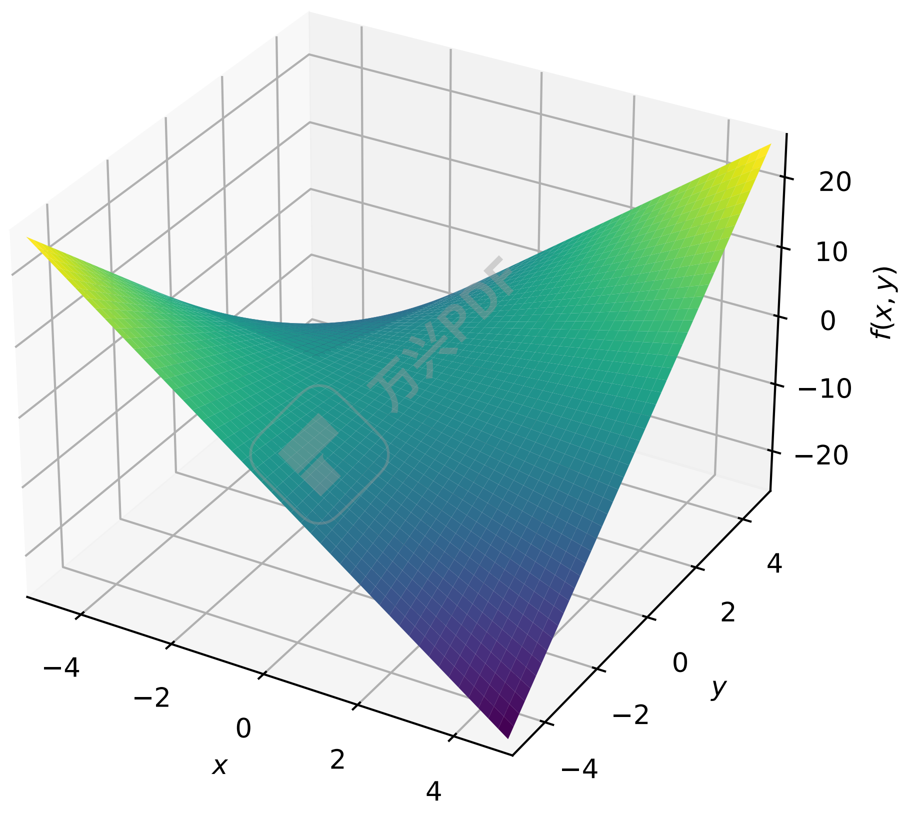
Multiple plots with different ranges.

```
>>> plot3d((x**2 + y**2, (x, -5, 5), (y, -5, 5)),
...     (x*y, (x, -3, 3), (y, -3, 3)))
Plot object containing:
[0]: cartesian surface: x**2 + y**2 for x over (-5.0, 5.0) and y over (-
↪5.0, 5.0)
[1]: cartesian surface: x*y for x over (-3.0, 3.0) and y over (-3.0, 3.0)
```

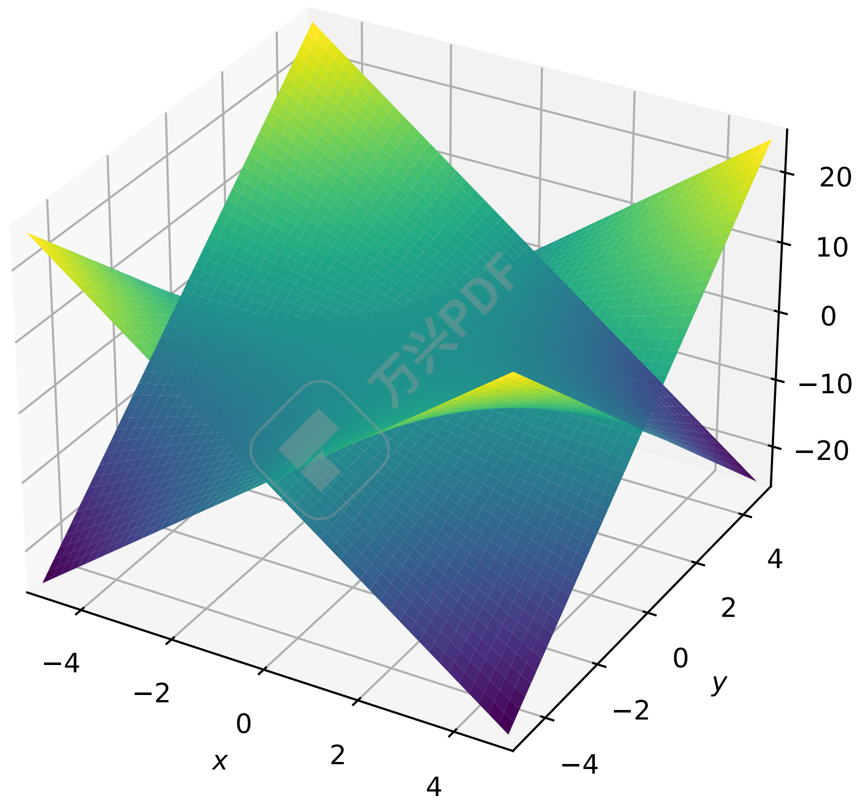
See also:

Plot (page 2821), *SurfaceOver2DRangeSeries* (page 2867)

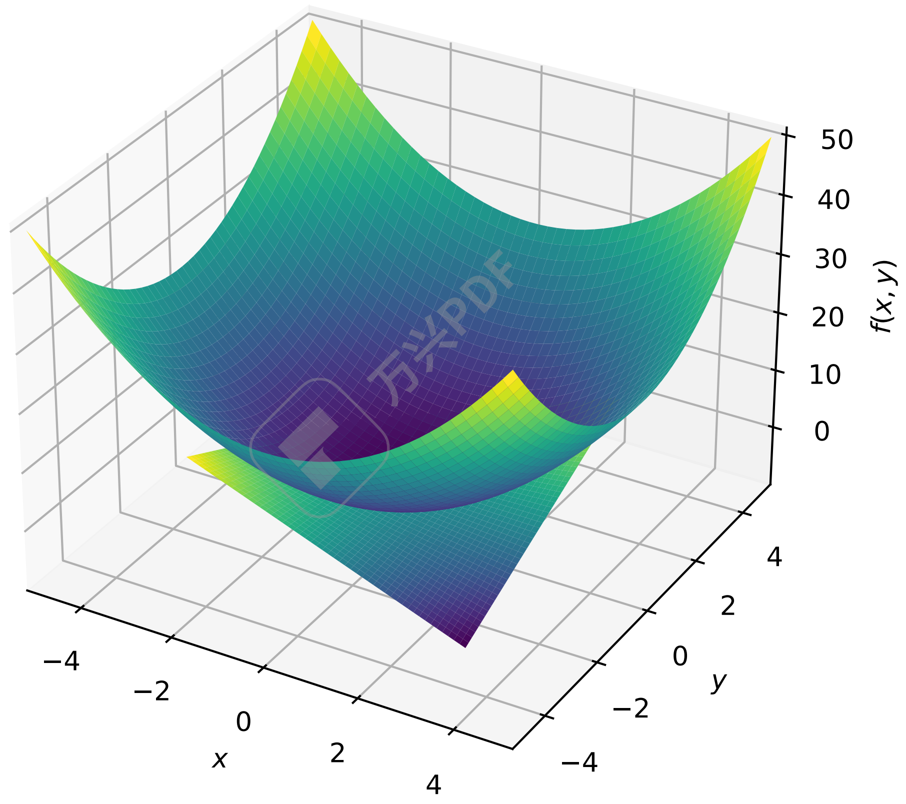
`sympy.plotting.plot.`**`plot3d_parametric_line`**(**args*, *show=True*, ***kwargs*)
    Plots a 3D parametric line plot.



−4
−2
0
2
4
x
−4
−2
0
2
4
y
−20
−10
0
10
20
f(x, y)



20
10
0
−10
−20
f(x, y)
−4
−2
0
2
4
x
4
2
0
−2
−4
y


50
40
30
20
10
0
f(x, y)
−4
−2
0
2
4
x
4
2
0
−2
−4
y

Usage

Single plot:

`plot3d_parametric_line(expr_x, expr_y, expr_z, range, **kwargs)`

If the range is not specified, then a default range of (-10, 10) is used.

Multiple plots.

`plot3d_parametric_line((expr_x, expr_y, expr_z, range), ..., **kwargs)`

Ranges have to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

Arguments

expr_x : Expression representing the function along x.

expr_y : Expression representing the function along y.

expr_z : Expression representing the function along z.

range
: [(*Symbol* (page 976), float, float)] A 3-tuple denoting the range of the parameter variable, e.g., (u, 0, 5).

Keyword Arguments

Arguments for `Parametric3DLineSeries` class.

nb_of_points : The range is uniformly sampled at `nb_of_points` number of points.

Aesthetics:

line_color
: [string, or float, or function, optional] Specifies the color for the plot. See `Plot` to see how to set color for the plots. Note that by setting `line_color`, it would be applied simultaneously to all the series.

label
: [str] The label to the plot. It will be used when called with `legend=True` to denote the function with the given label in the plot.

If there are multiple plots, then the same series arguments are applied to all the plots. If you want to set these options separately, you can index the returned `Plot` object and set it.

Arguments for `Plot` class.

title
: [str] Title of the plot.

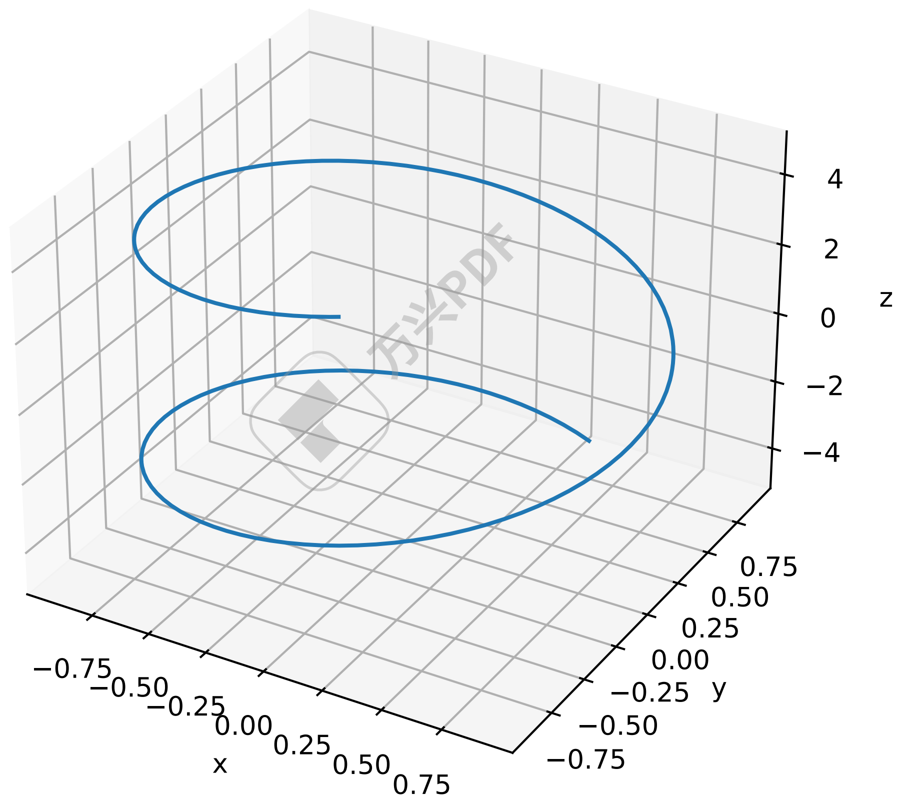
size
: [(float, float), optional] A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to `None`, meaning the size will be set by the default backend.

Examples

```
>>> from sympy import symbols, cos, sin
>>> from sympy.plotting import plot3d_parametric_line
>>> u = symbols('u')
```

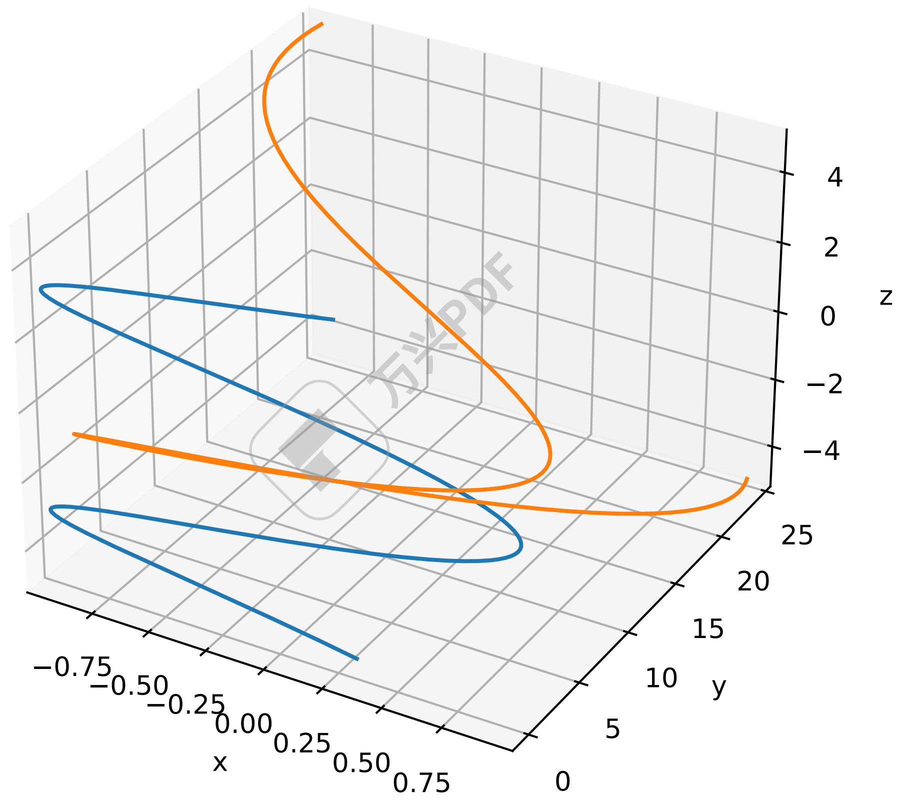
Single plot.

```
>>> plot3d_parametric_line(cos(u), sin(u), u, (u, -5, 5))
Plot object containing:
[0]: 3D parametric cartesian line: (cos(u), sin(u), u) for u over (-5.0,␣
↪5.0)
```



Multiple plots.

```
>>> plot3d_parametric_line((cos(u), sin(u), u, (u, -5, 5)),
...     (sin(u), u**2, u, (u, -5, 5)))
Plot object containing:
[0]: 3D parametric cartesian line: (cos(u), sin(u), u) for u over (-5.0,␣
↪5.0)
[1]: 3D parametric cartesian line: (sin(u), u**2, u) for u over (-5.0, 5.
↪0)
```


4
2
0
−2
−4
z
25
20
15
10
5
0
y
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
x

See also:

`Plot` (page 2821), `Parametric3DLineSeries` (page 2867)

`sympy.plotting.plot.`**`plot3d_parametric_surface`**`(`**args, show=True, **kwargs*`)`

Plots a 3D parametric surface plot.

Explanation

Single plot.

`plot3d_parametric_surface(expr_x, expr_y, expr_z, range_u, range_v, **kwargs)`

If the ranges is not specified, then a default range of (-10, 10) is used.

Multiple plots.

`plot3d_parametric_surface((expr_x, expr_y, expr_z, range_u, range_v), ..., **kwargs)`

Ranges have to be specified for every expression.

Default range may change in the future if a more advanced default range detection algorithm is implemented.

Arguments

expr_x : Expression representing the function along x.

expr_y : Expression representing the function along y.

expr_z : Expression representing the function along z.

range_u
[(`Symbol` (page 976), float, float)] A 3-tuple denoting the range of the u variable, e.g. (u, 0, 5).

range_v
[(`Symbol` (page 976), float, float)] A 3-tuple denoting the range of the v variable, e.g. (v, 0, 5).

Keyword Arguments

Arguments for `ParametricSurfaceSeries` class:

nb_of_points_u
[int] The `u` range is sampled uniformly at `nb_of_points_v` of points

nb_of_points_y
[int] The `v` range is sampled uniformly at `nb_of_points_y` of points

Aesthetics:

surface_color
[Function which returns a float] Specifies the color for the surface of the plot. See `Plot` (page 2821) for more details.

If there are multiple plots, then the same series arguments are applied for all the plots. If you want to set these options separately, you can index the returned `Plot` object and set it.

Arguments for `Plot` class:

title
[str] Title of the plot.

size
[(float, float), optional] A tuple in the form (width, height) in inches to specify the size of the overall figure. The default value is set to `None`, meaning the size will be set by the default backend.

Examples

```
>>> from sympy import symbols, cos, sin
>>> from sympy.plotting import plot3d_parametric_surface
>>> u, v = symbols('u v')
```

Single plot.

```
>>> plot3d_parametric_surface(cos(u + v), sin(u - v), u - v,
...     (u, -5, 5), (v, -5, 5))
Plot object containing:
[0]: parametric cartesian surface: (cos(u + v), sin(u - v), u - v) for u␣
↪over (-5.0, 5.0) and v over (-5.0, 5.0)
```

See also:

Plot (page 2821), *ParametricSurfaceSeries* (page 2867)

`sympy.plotting.plot_implicit.`**`plot_implicit`**(*expr*, *x_var=None*, *y_var=None*, *adaptive=True*, *depth=0*, *points=300*, *line_color='blue'*, *show=True*, ***kwargs*)

A plot function to plot implicit equations / inequalities.

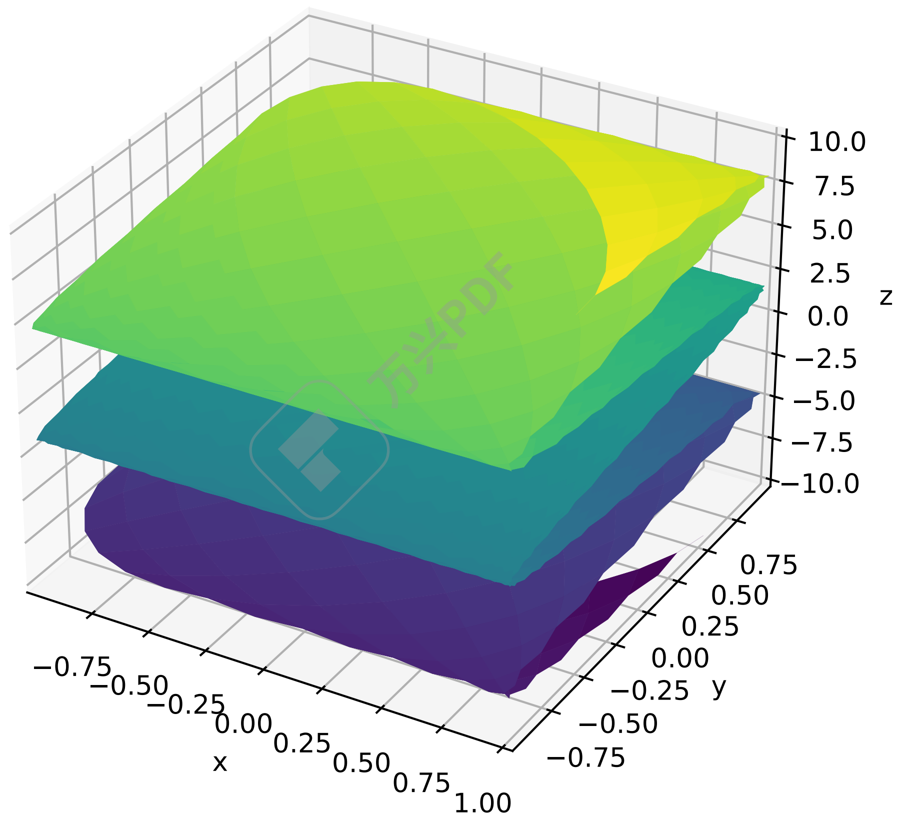
Arguments

- `expr` : The equation / inequality that is to be plotted.
- `x_var` (optional) : symbol to plot on x-axis or tuple giving symbol and range as `(symbol, xmin, xmax)`
- `y_var` (optional) : symbol to plot on y-axis or tuple giving symbol and range as `(symbol, ymin, ymax)`

If neither `x_var` nor `y_var` are given then the free symbols in the expression will be assigned in the order they are sorted.

The following keyword arguments can also be used:

- **adaptive Boolean. The default value is set to True. It has to be**
  set to False if you want to use a mesh grid.
- **depth integer. The depth of recursion for adaptive mesh grid.**
  Default value is 0. Takes value in the range (0, 4).


10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
z
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
y
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00
x

- **points integer. The number of points if adaptive mesh grid is not**
  used. Default value is 300.
- **show Boolean. Default value is True. If set to False, the plot will**
  not be shown. See Plot for further information.
- title string. The title for the plot.
- xlabel string. The label for the x-axis
- ylabel string. The label for the y-axis

Aesthetics options:

- **line_color: float or string. Specifies the color for the plot.**
  See Plot to see how to set color for the plots. Default value is "Blue"

plot_implicit, by default, uses interval arithmetic to plot functions. If the expression cannot be plotted using interval arithmetic, it defaults to a generating a contour using a mesh grid of fixed number of points. By setting adaptive to False, you can force plot_implicit to use the mesh grid. The mesh grid method can be effective when adaptive plotting using interval arithmetic, fails to plot with small line width.

Examples

Plot expressions:

```
>>> from sympy import plot_implicit, symbols, Eq, And
>>> x, y = symbols('x y')
```

Without any ranges for the symbols in the expression:

```
>>> p1 = plot_implicit(Eq(x**2 + y**2, 5))
```

With the range for the symbols:

```
>>> p2 = plot_implicit(
...     Eq(x**2 + y**2, 3), (x, -3, 3), (y, -3, 3))
```

With depth of recursion as argument:

```
>>> p3 = plot_implicit(
...     Eq(x**2 + y**2, 5), (x, -4, 4), (y, -4, 4), depth = 2)
```

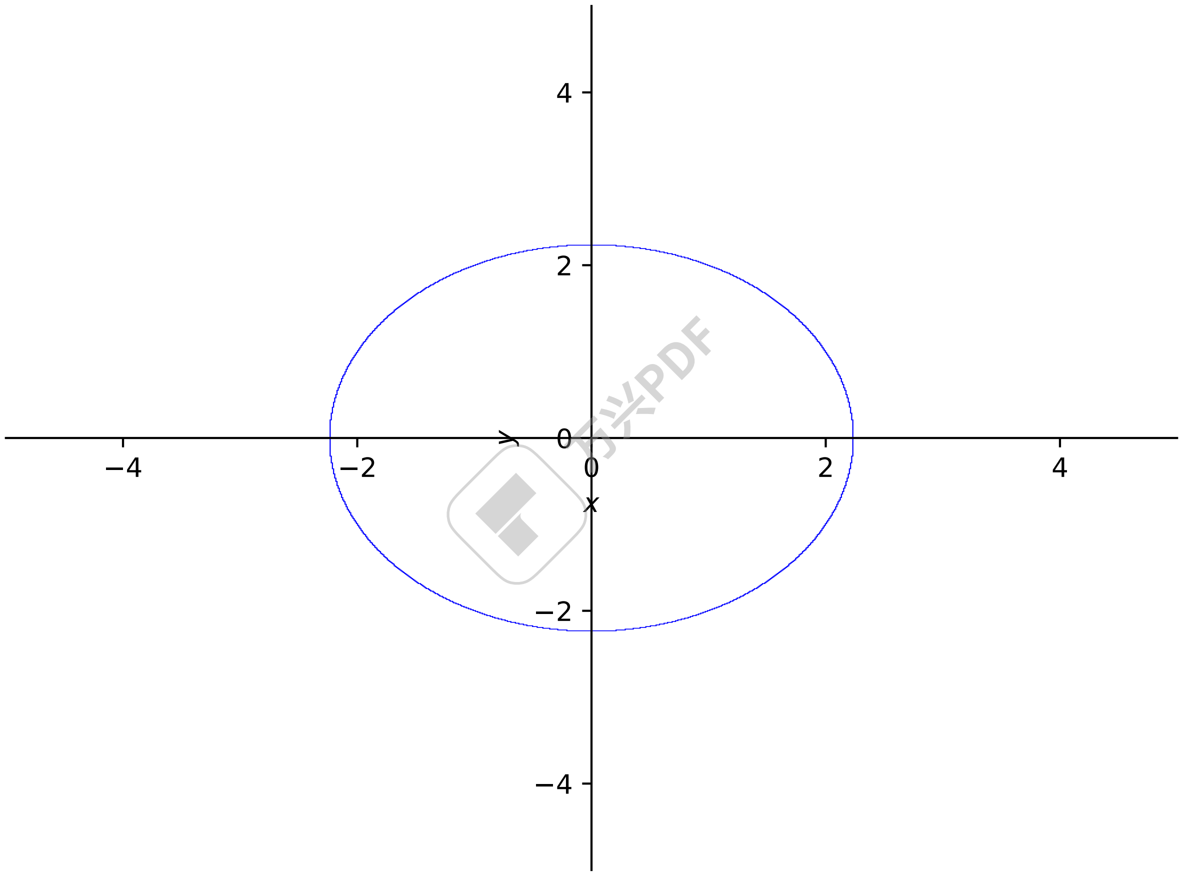
Using mesh grid and not using adaptive meshing:

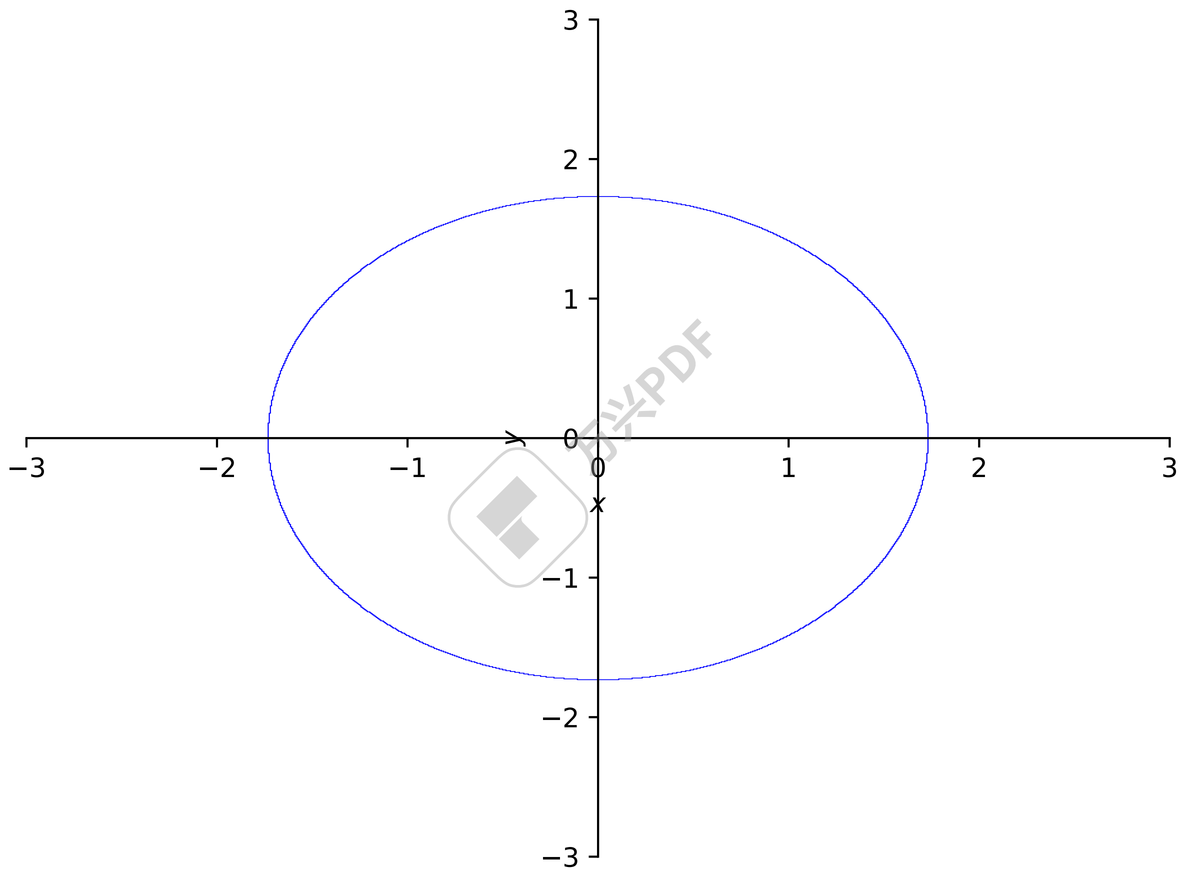
```
>>> p4 = plot_implicit(
...     Eq(x**2 + y**2, 5), (x, -5, 5), (y, -2, 2),
...     adaptive=False)
```

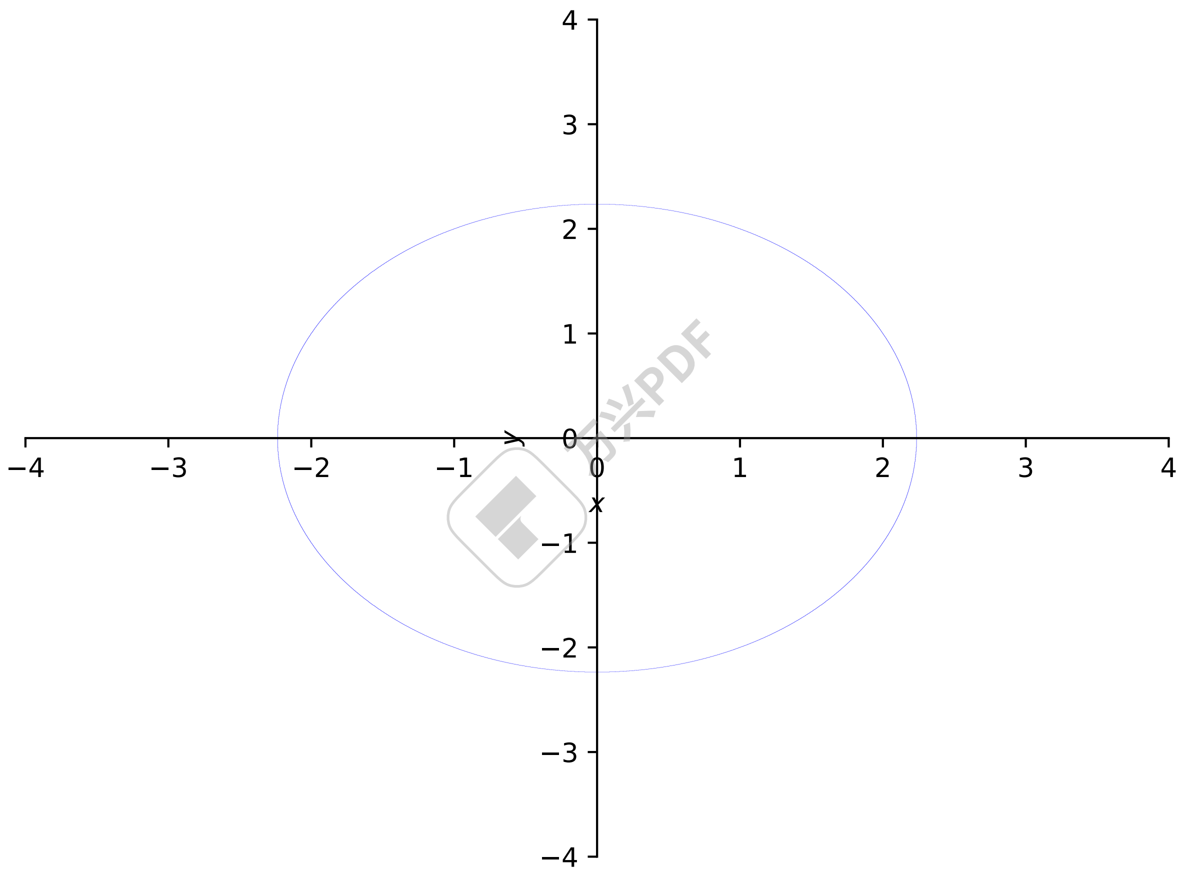
Using mesh grid without using adaptive meshing with number of points specified:

```
>>> p5 = plot_implicit(
...     Eq(x**2 + y**2, 5), (x, -5, 5), (y, -2, 2),
...     adaptive=False, points=400)
```

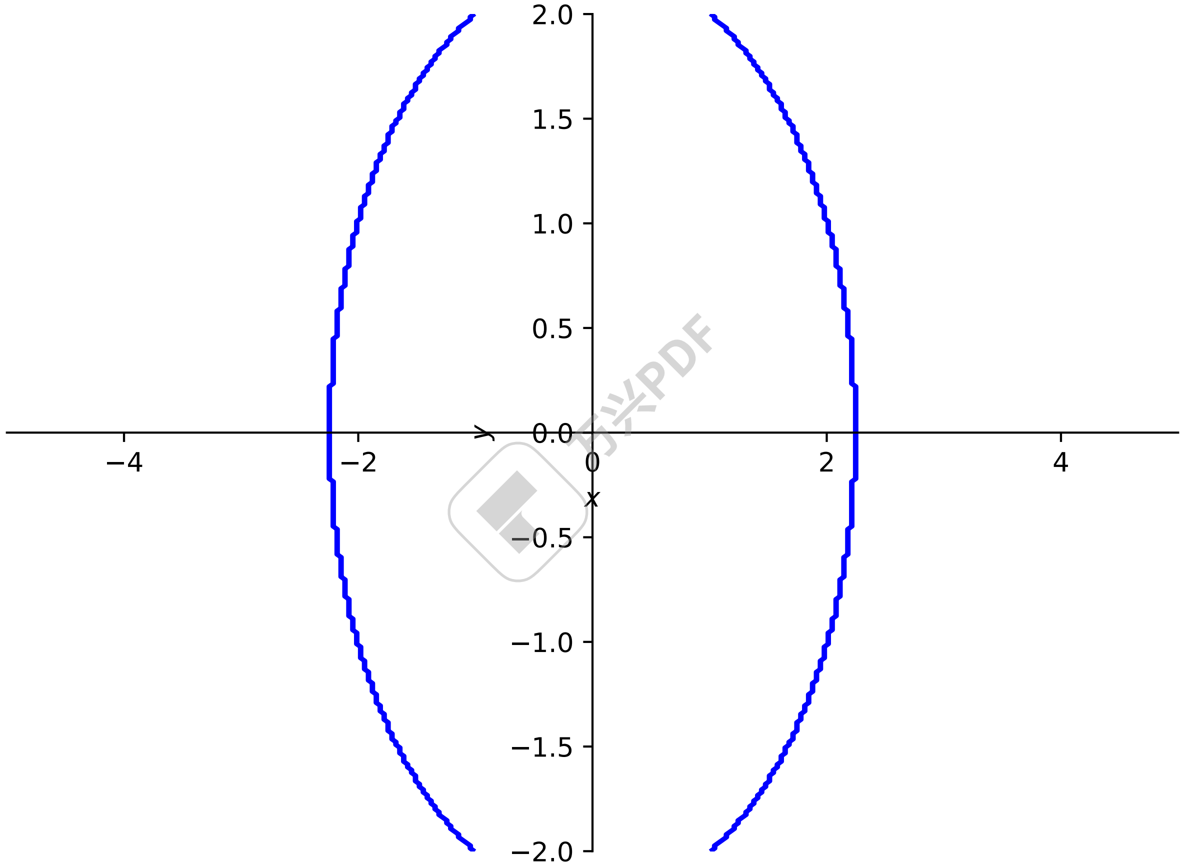
Plotting regions:

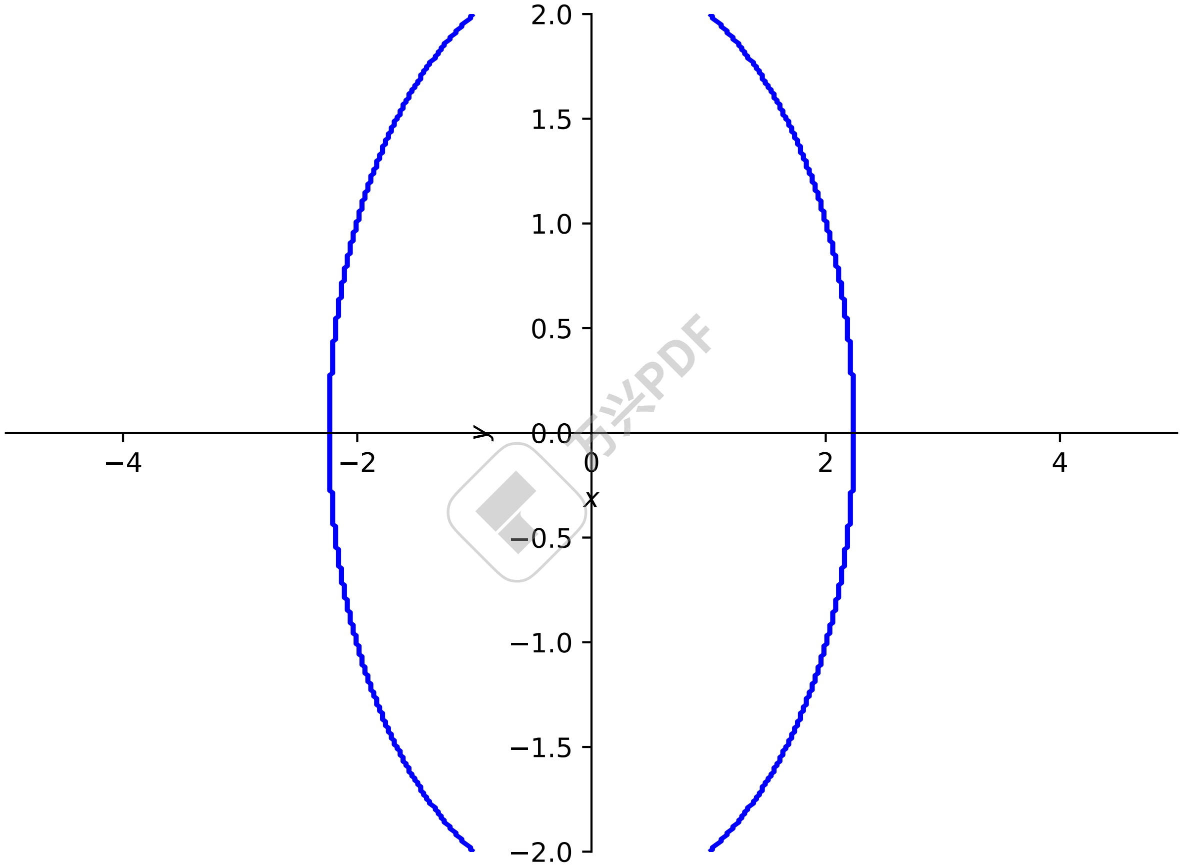

4
2
0
−2
−4
−4
−2
0
2
4
x
y


3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
x
y

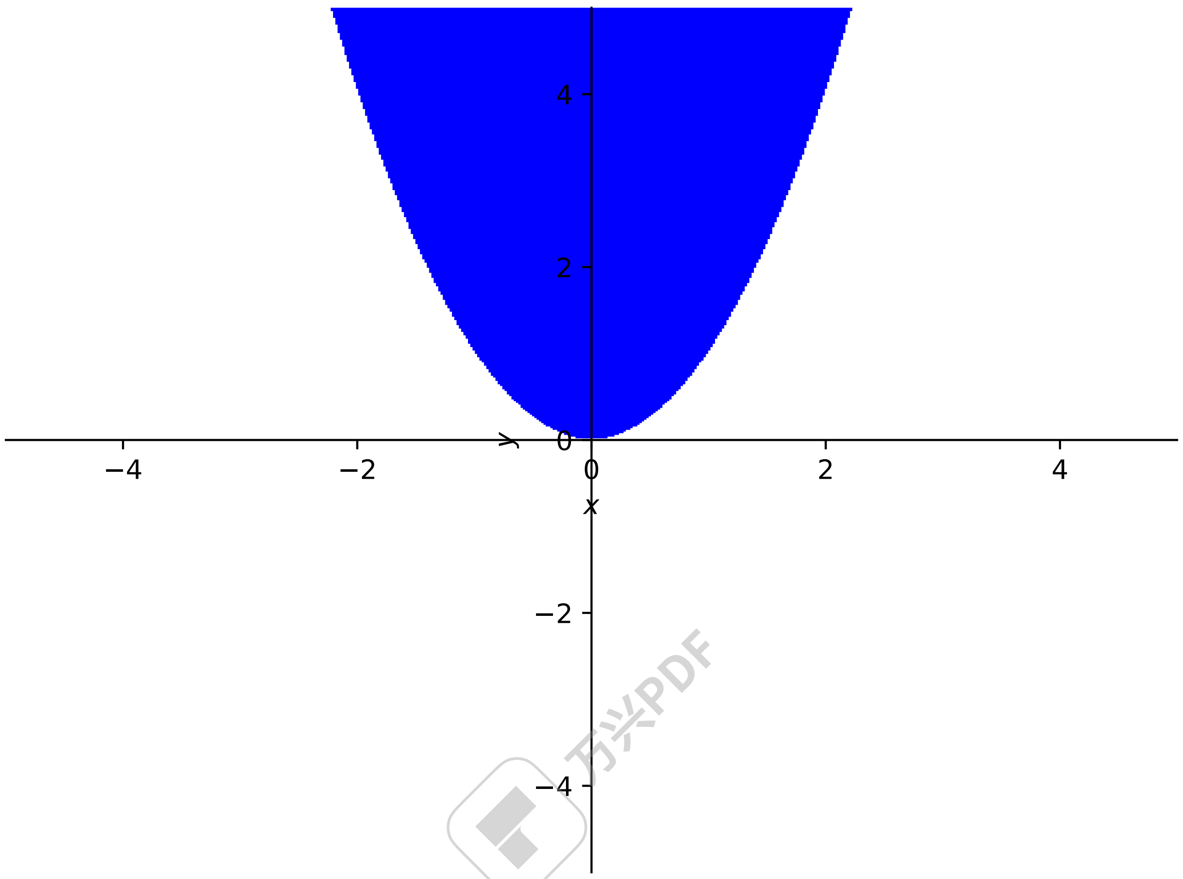


4
3
2
1
0
−1
−2
−3
−4
y
x
−4
−3
−2
−1
0
1
2
3
4


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−4
−2
0
2
4
y
x


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−4
−2
0
2
4
x
y

```
>>> p6 = plot_implicit(y > x**2)
```



Plotting Using boolean conjunctions:

```
>>> p7 = plot_implicit(And(y > x, y > -x))
```

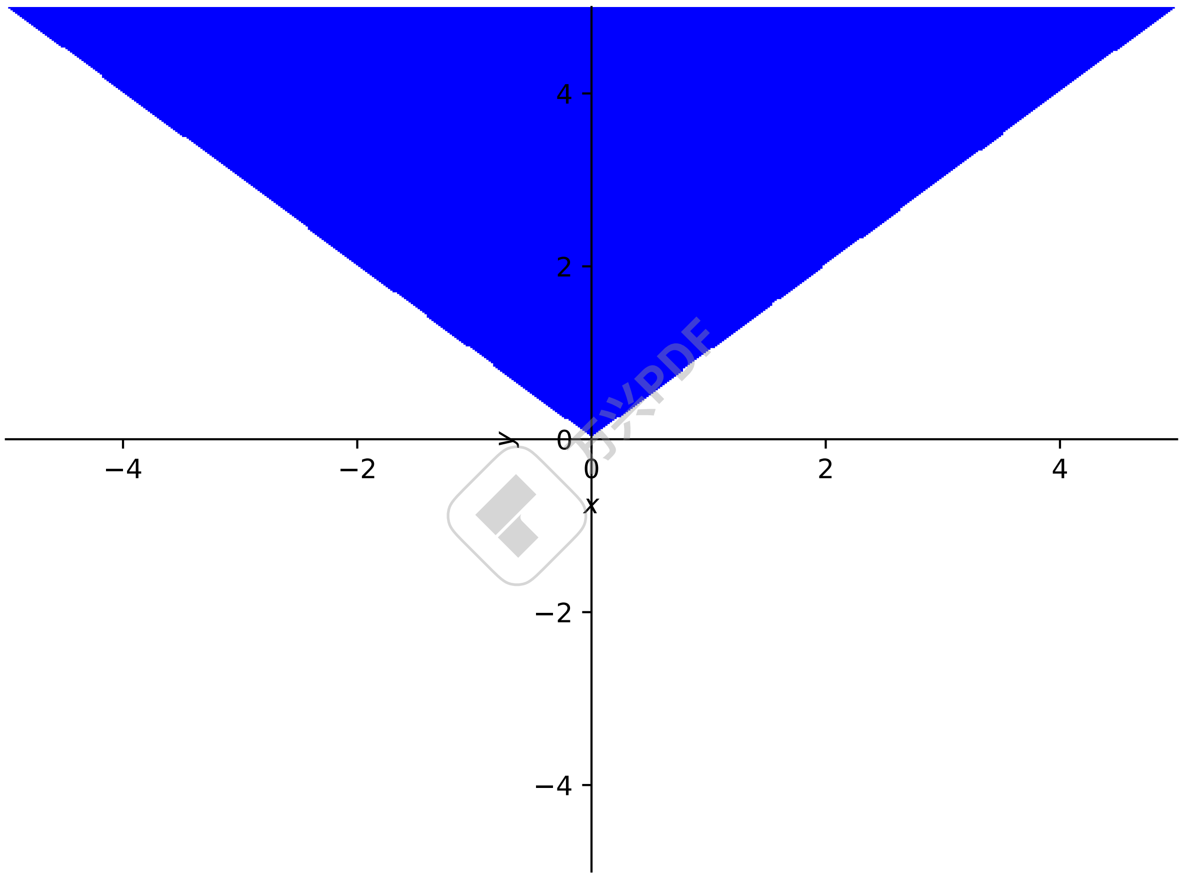
When plotting an expression with a single variable (y - 1, for example), specify the x or the y variable explicitly:

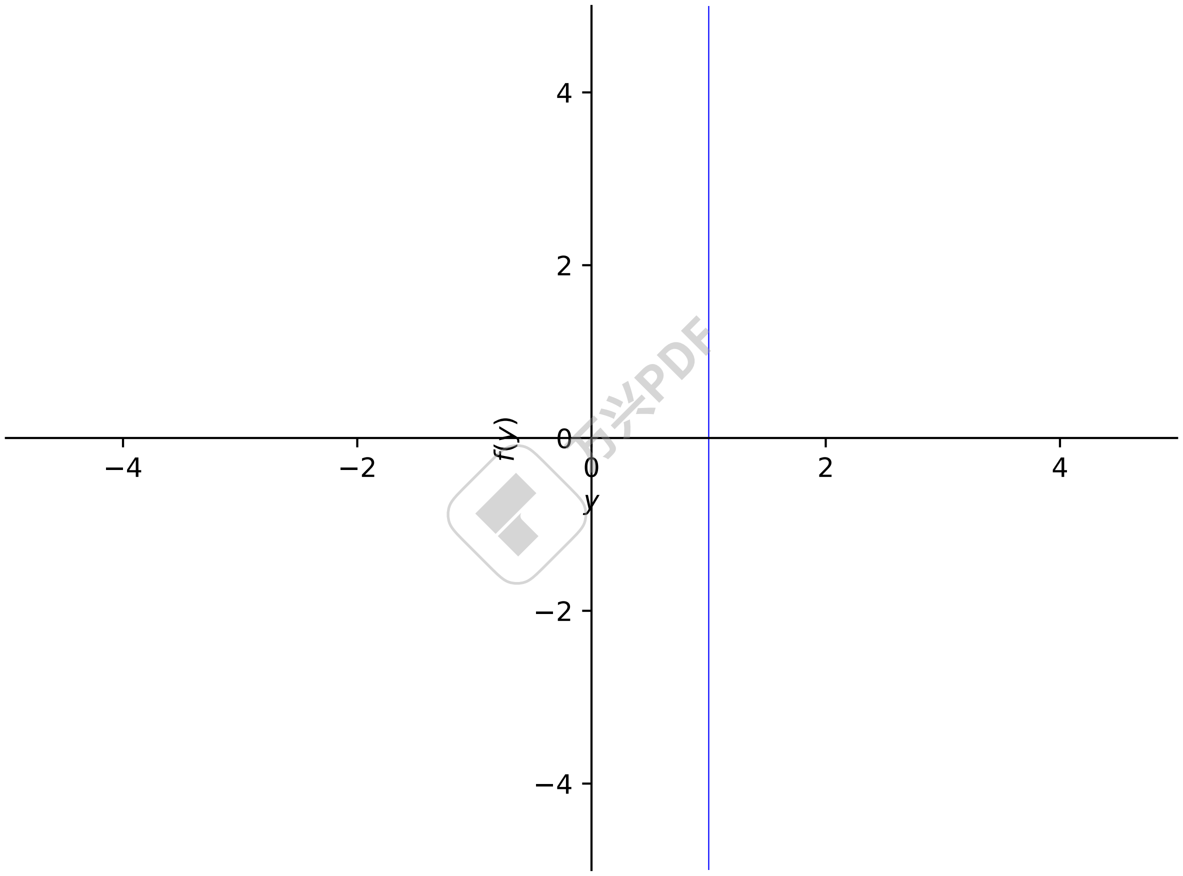
```
>>> p8 = plot_implicit(y - 1, y_var=y)
>>> p9 = plot_implicit(x - 1, x_var=x)
```

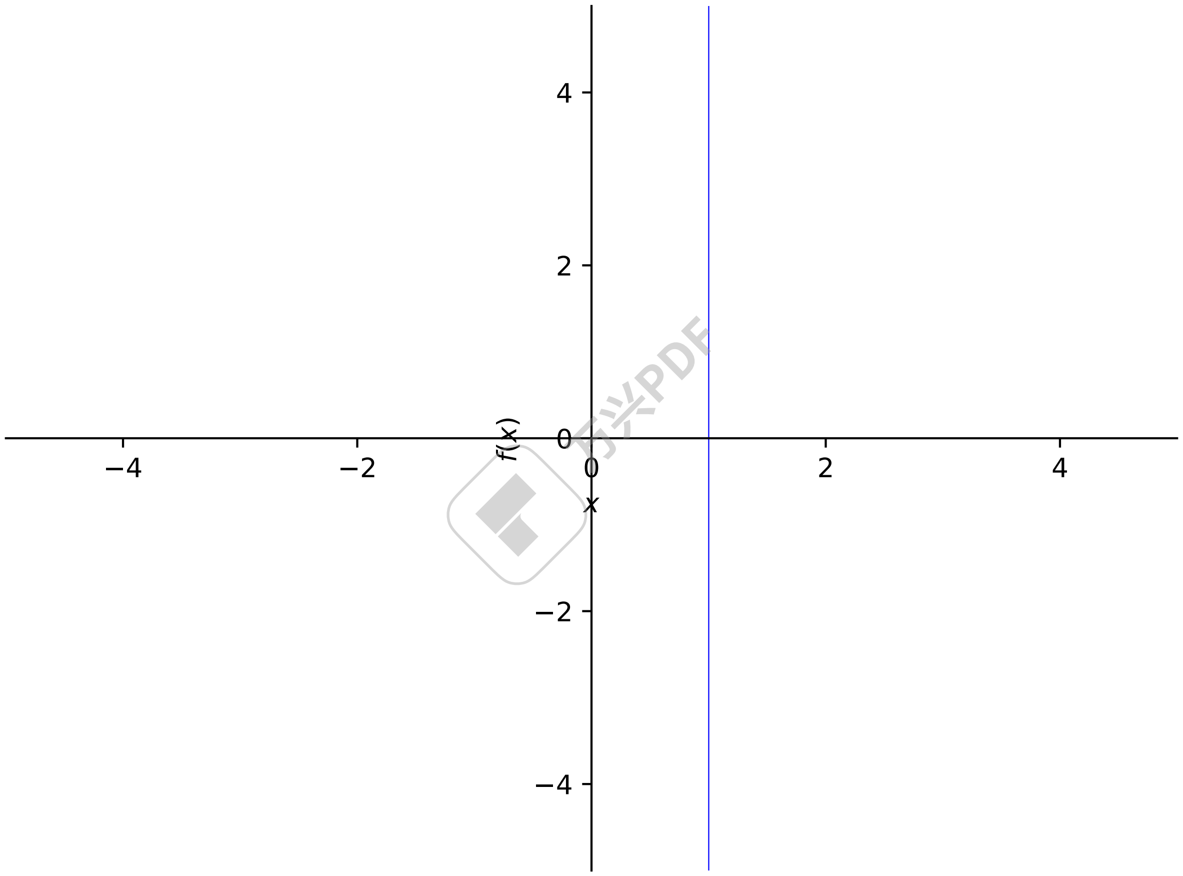
PlotGrid Class

class sympy.plotting.plot.**PlotGrid**(*nrows*, *ncolumns*, **args*, *show=True*, *size=None*, ***kwargs*)

This class helps to plot subplots from already created SymPy plots in a single figure.


4
2
0
−2
−4
−4
−2
0
2
4
y
x

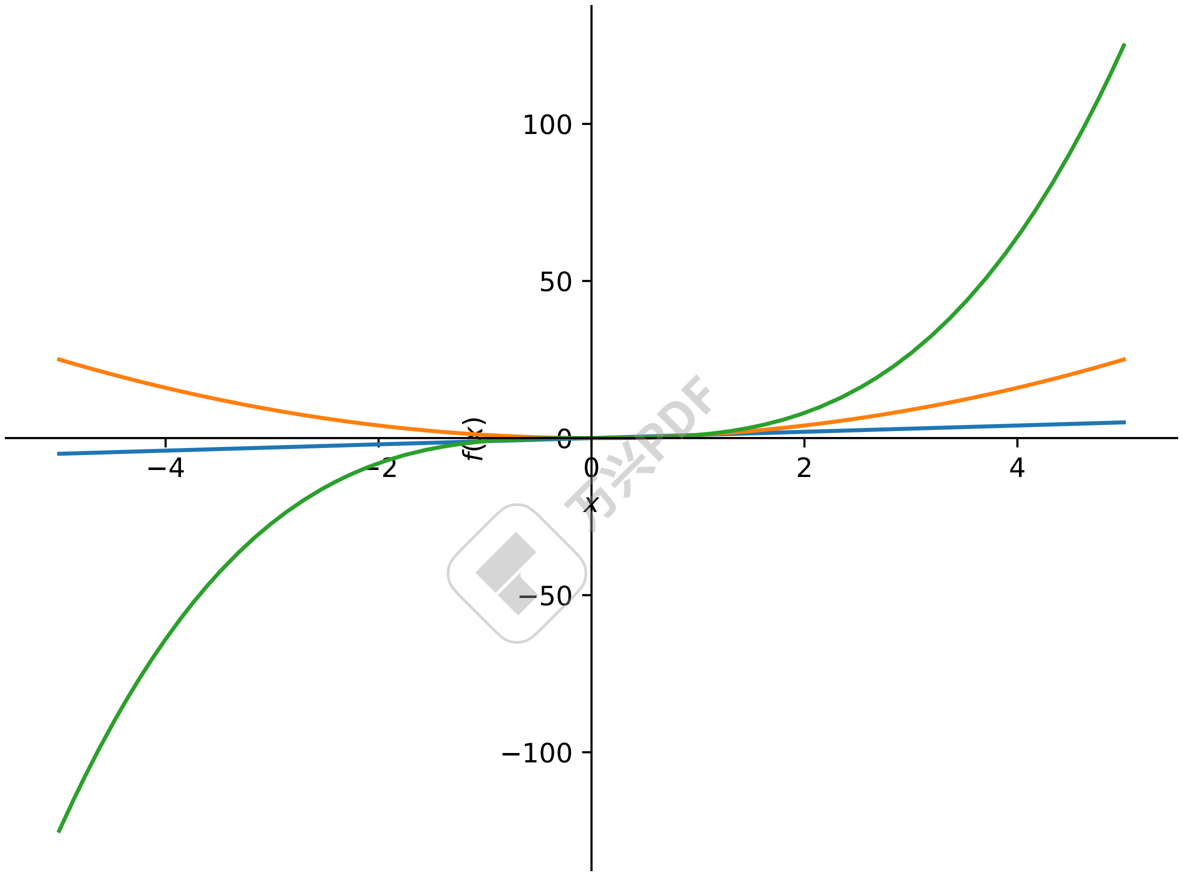

4
2
0
−2
−4
f(y)
−4
−2
0
2
4
y



4
2
0
−2
−4
−4
−2
0
2
4
f(x)
x

Examples

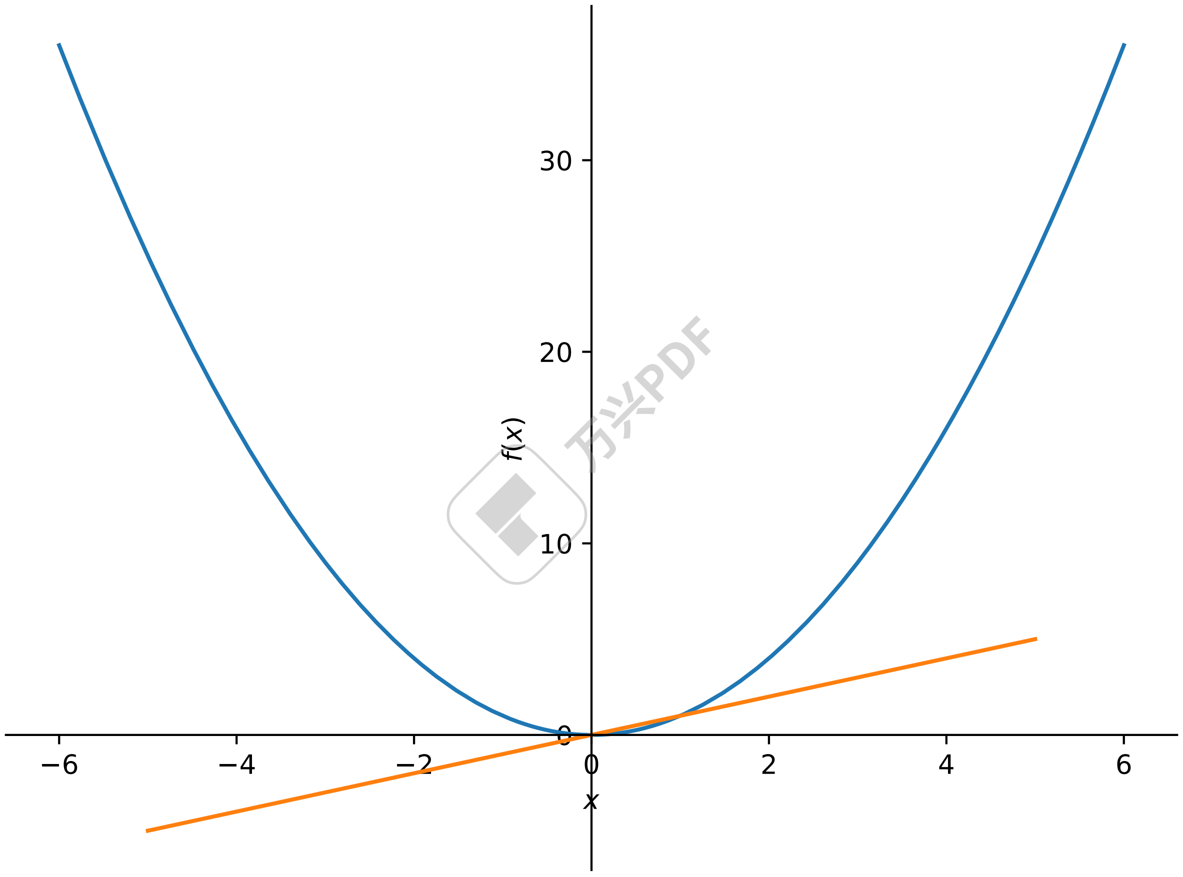
```
>>> from sympy import symbols
>>> from sympy.plotting import plot, plot3d, PlotGrid
>>> x, y = symbols('x, y')
>>> p1 = plot(x, x**2, x**3, (x, -5, 5))
>>> p2 = plot((x**2, (x, -6, 6)), (x, (x, -5, 5)))
>>> p3 = plot(x**3, (x, -5, 5))
>>> p4 = plot3d(x*y, (x, -5, 5), (y, -5, 5))
```

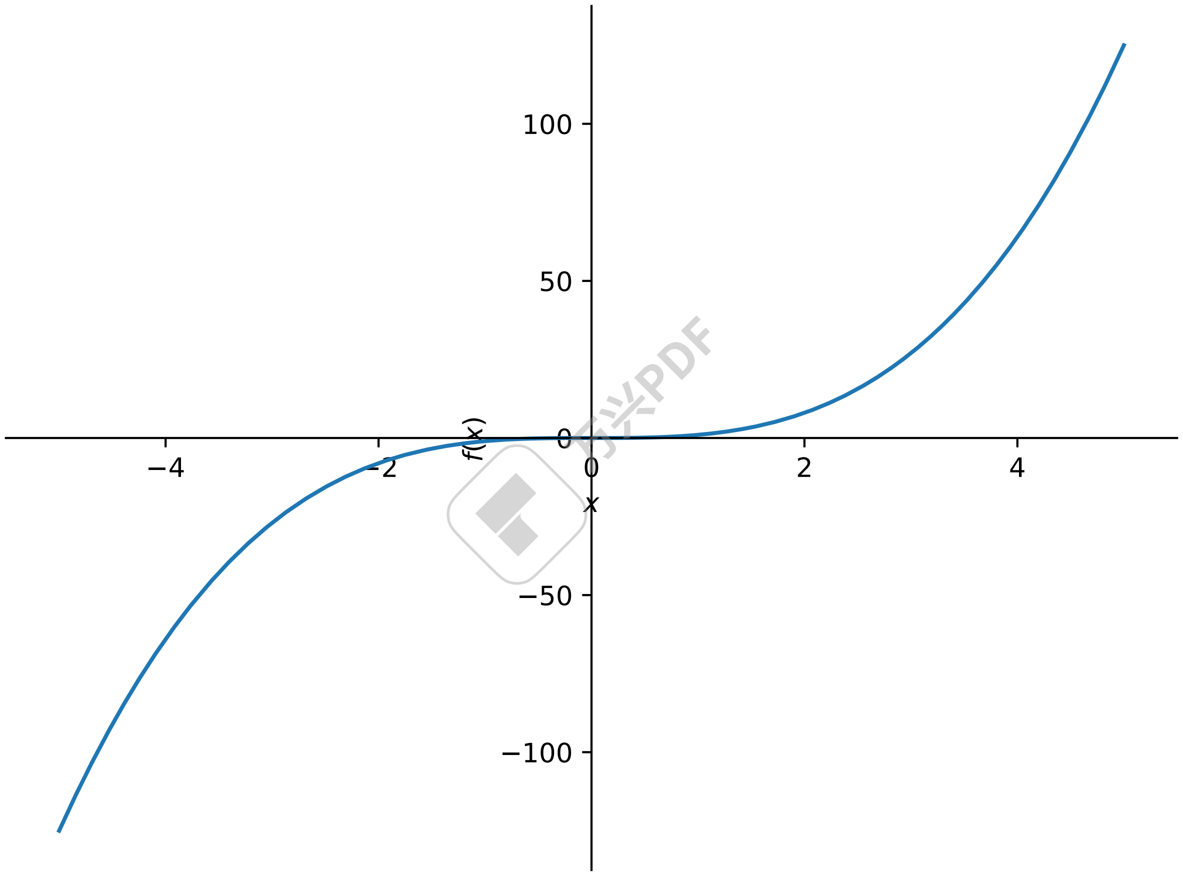


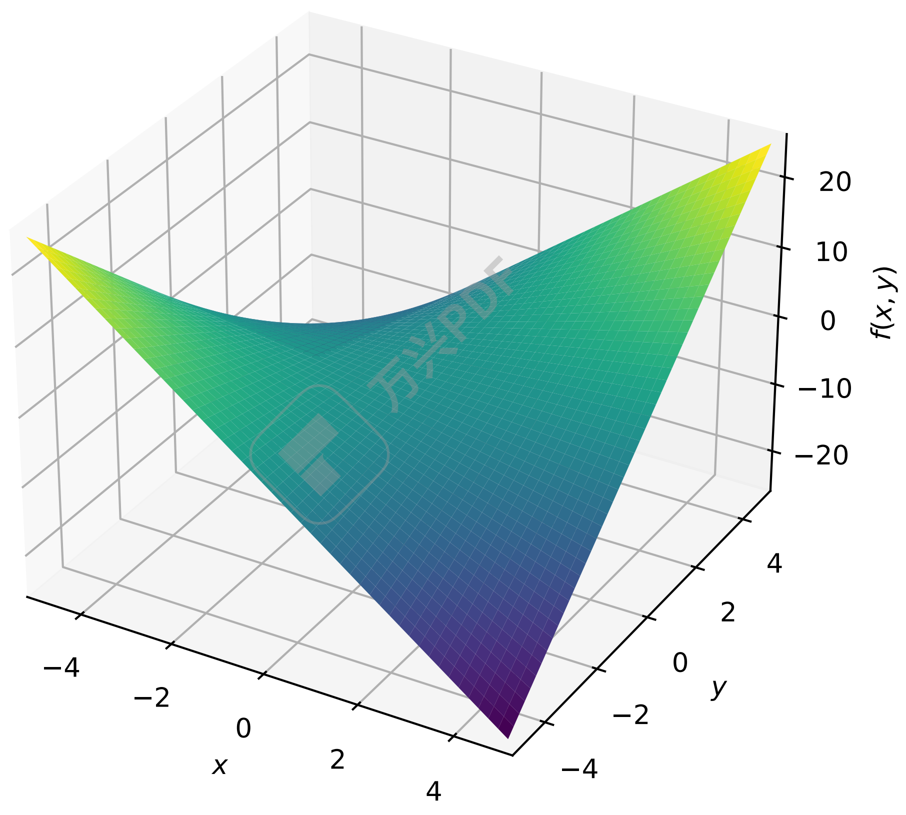
Plotting vertically in a single line:

```
>>> PlotGrid(2, 1, p1, p2)
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: x for x over (-5.0, 5.0)
[1]: cartesian line: x**2 for x over (-5.0, 5.0)
[2]: cartesian line: x**3 for x over (-5.0, 5.0)
Plot[1]:Plot object containing:
[0]: cartesian line: x**2 for x over (-6.0, 6.0)
[1]: cartesian line: x for x over (-5.0, 5.0)
```

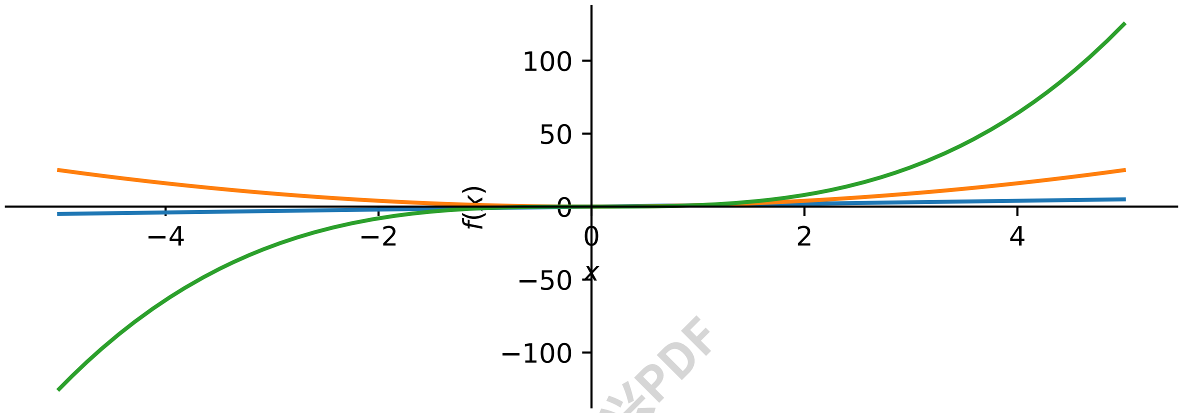
Plotting horizontally in a single line:


30
20
10
0
f(x)
−6
−4
−2
0
2
4
6
x

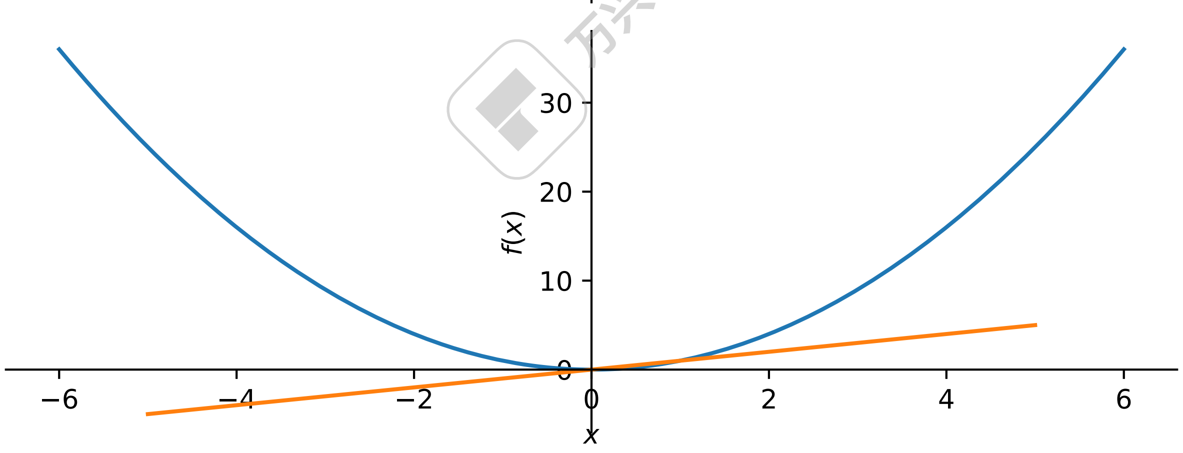

100
50
0
−50
−100
−4
−2
0
2
4
x
f(x)



20
10
0
−10
−20
f(x, y)
4
2
0
y
−2
−4
−4
−2
0
x
2
4

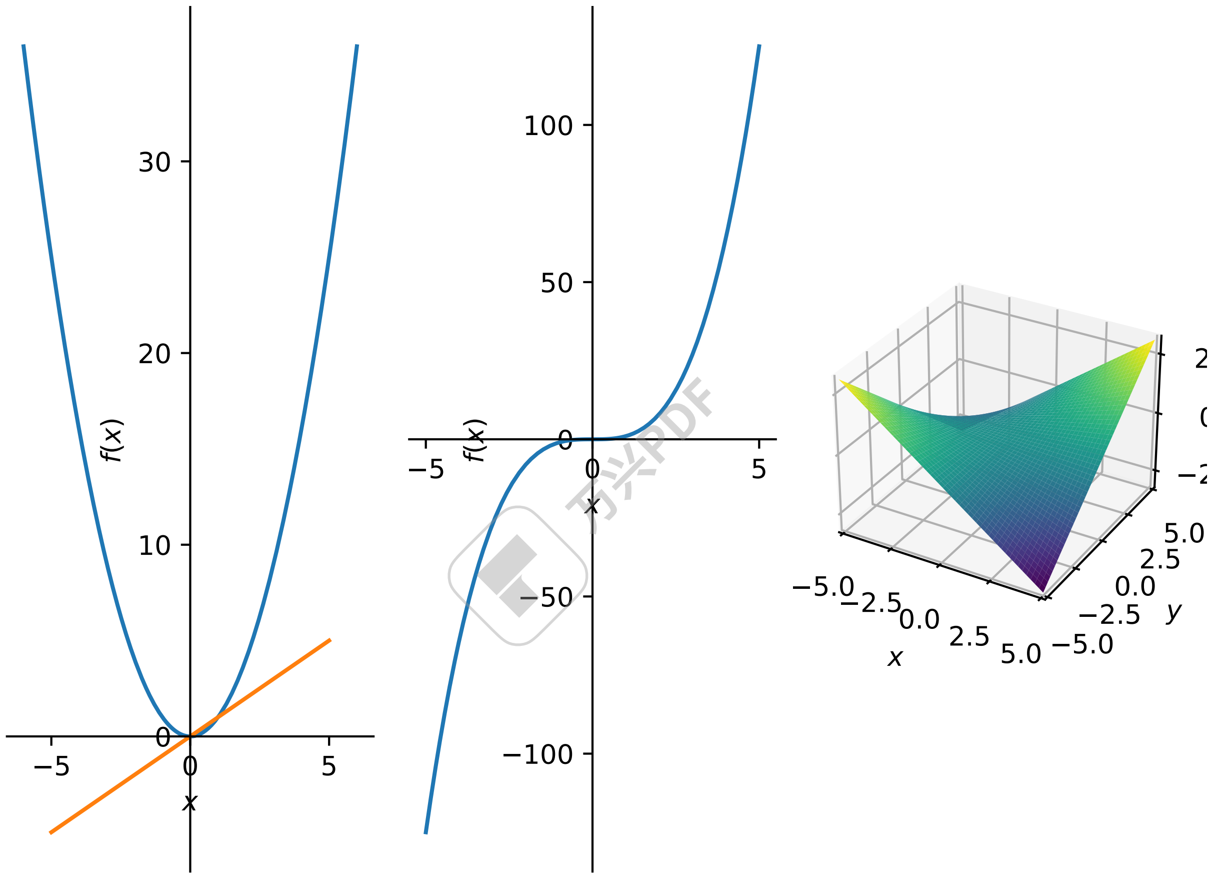


100
50
0
−50
−100
−4
−2
0
2
4
f(x)
x



30
20
10
0
−6
−4
−2
0
2
4
6
f(x)
x

```
>>> PlotGrid(1, 3, p2, p3, p4)
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: x**2 for x over (-6.0, 6.0)
[1]: cartesian line: x for x over (-5.0, 5.0)
Plot[1]:Plot object containing:
[0]: cartesian line: x**3 for x over (-5.0, 5.0)
Plot[2]:Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```



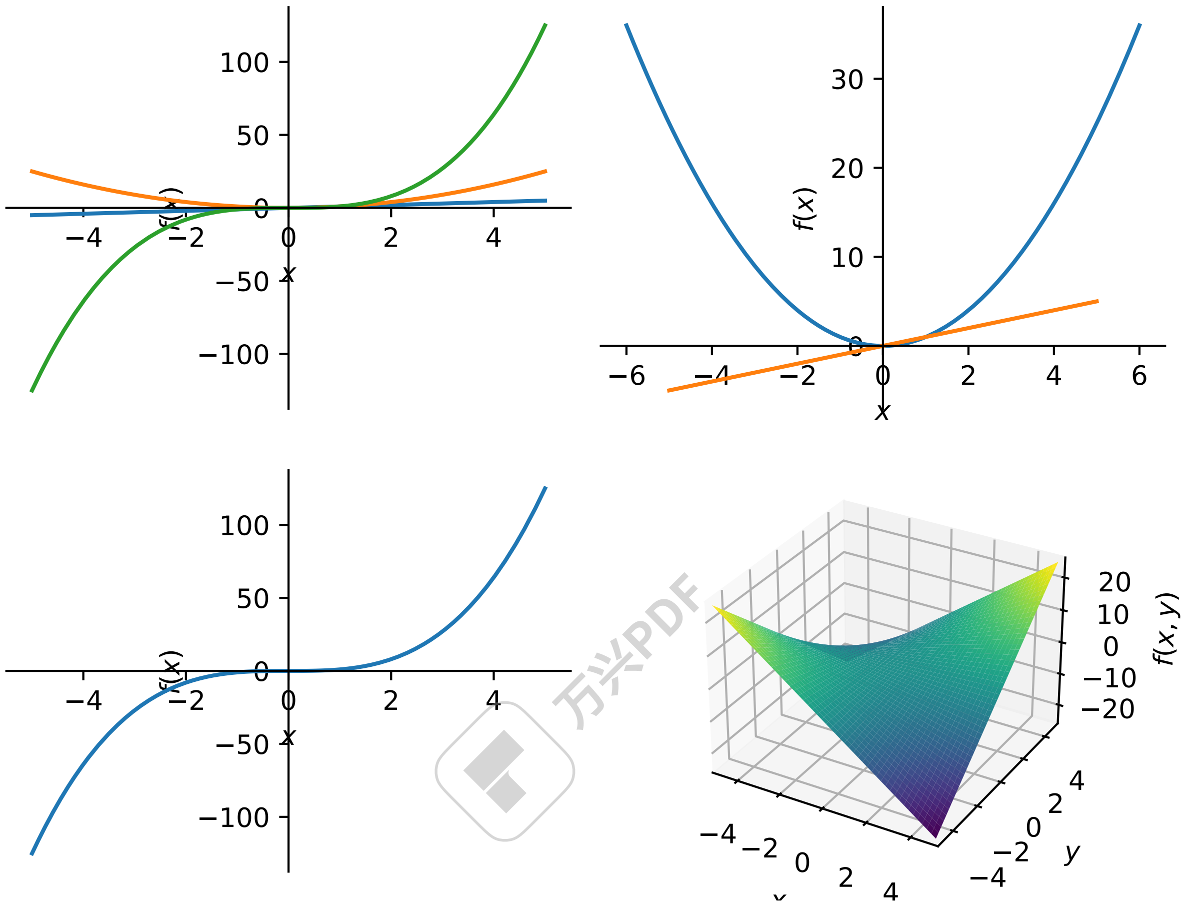
Plotting in a grid form:

```
>>> PlotGrid(2, 2, p1, p2, p3, p4)
PlotGrid object containing:
Plot[0]:Plot object containing:
[0]: cartesian line: x for x over (-5.0, 5.0)
[1]: cartesian line: x**2 for x over (-5.0, 5.0)
[2]: cartesian line: x**3 for x over (-5.0, 5.0)
Plot[1]:Plot object containing:
[0]: cartesian line: x**2 for x over (-6.0, 6.0)
[1]: cartesian line: x for x over (-5.0, 5.0)
Plot[2]:Plot object containing:
[0]: cartesian line: x**3 for x over (-5.0, 5.0)
```

(continues on next page)

(continued from previous page)

```
Plot[3]:Plot object containing:
[0]: cartesian surface: x*y for x over (-5.0, 5.0) and y over (-5.0, 5.0)
```



Series Classes

class `sympy.plotting.plot.`**`BaseSeries`**

Base class for the data objects containing stuff to be plotted.

Explanation

The backend should check if it supports the data series that is given. (e.g. TextBackend supports only LineOver1DRangeSeries). It is the backend responsibility to know how to use the class of data series that is given.

Some data series classes are grouped (using a class attribute like is_2Dline) according to the api they present (based only on convention). The backend is not obliged to use that api (e.g. LineOver1DRangeSeries belongs to the is_2Dline group and presents the get_points method, but the TextBackend does not use the get_points method).

class sympy.plotting.plot.**Line2DBaseSeries**

A base class for 2D lines.

- adding the label, steps and only_integers options
- making is_2Dline true
- defining get_segments and get_color_array

get_data()

Return lists of coordinates for plotting the line.

Returns

x : list

List of x-coordinates

y

[list] List of y-coordinates

z

[list] List of z-coordinates in case of Parametric3DLineSeries

class sympy.plotting.plot.**LineOver1DRangeSeries**(*expr*, *var_start_end*, ***kwargs*)

Representation for a line consisting of a SymPy expression over a range.

get_points()

Return lists of coordinates for plotting. Depending on the `adaptive` option, this function will either use an adaptive algorithm or it will uniformly sample the expression over the provided range.

Returns

x : list

List of x-coordinates

y

[list] List of y-coordinates

Explanation

The adaptive sampling is done by recursively checking if three points are almost collinear. If they are not collinear, then more points are added between those points.

References

[R692]

class sympy.plotting.plot.**Parametric2DLineSeries**(*expr_x*, *expr_y*, *var_start_end*, ***kwargs*)

Representation for a line consisting of two parametric SymPy expressions over a range.

get_points()

Return lists of coordinates for plotting. Depending on the `adaptive` option, this function will either use an adaptive algorithm or it will uniformly sample the expression over the provided range.

Returns

x : list

List of x-coordinates

y

[list] List of y-coordinates

Explanation

The adaptive sampling is done by recursively checking if three points are almost collinear. If they are not collinear, then more points are added between those points.

References

[R693]

class sympy.plotting.plot.**Line3DBaseSeries**

A base class for 3D lines.

Most of the stuff is derived from Line2DBaseSeries.

class sympy.plotting.plot.**Parametric3DLineSeries**(*expr_x*, *expr_y*, *expr_z*, *var_start_end*, ***kwargs*)

Representation for a 3D line consisting of three parametric SymPy expressions and a range.

class sympy.plotting.plot.**SurfaceBaseSeries**

A base class for 3D surfaces.

class sympy.plotting.plot.**SurfaceOver2DRangeSeries**(*expr*, *var_start_end_x*, *var_start_end_y*, ***kwargs*)

Representation for a 3D surface consisting of a SymPy expression and 2D range.

class sympy.plotting.plot.**ParametricSurfaceSeries**(*expr_x*, *expr_y*, *expr_z*, *var_start_end_u*, *var_start_end_v*, ***kwargs*)

Representation for a 3D surface consisting of three parametric SymPy expressions and a range.

class sympy.plotting.plot_implicit.**ImplicitSeries**(*expr*, *var_start_end_x*, *var_start_end_y*, *has_equality*, *use_interval_math*, *depth*, *nb_of_points*, *line_color*)

Representation for Implicit plot

Backends

class sympy.plotting.plot.**BaseBackend**(*parent*)

Base class for all backends. A backend represents the plotting library, which implements the necessary functionalities in order to use SymPy plotting functions.

How the plotting module works:

1. **Whenever a plotting function is called, the provided expressions are** processed and a list of instances of the *BaseSeries* (page 2865) class is created, containing the necessary information to plot the expressions (e.g. the expression, ranges, series name, …). Eventually, these objects will generate the numerical data to be plotted.
2. **A *Plot* (page 2821) object is instantiated, which stores the list of** series and the main attributes of the plot (e.g. axis labels, title, …).
3. **When the `show` command is executed, a new backend is instantiated,** which loops through each series object to generate and plot the numerical data. The backend is also going to set the axis labels, title, …, according to the values stored in the Plot instance.

The backend should check if it supports the data series that it is given (e.g. *TextBackend* (page 2869) supports only *LineOver1DRangeSeries* (page 2866)).

It is the backend responsibility to know how to use the class of data series that it's given. Note that the current implementation of the `*Series` classes is "matplotlib-centric": the numerical data returned by the `get_points` and `get_meshes` methods is meant to be used directly by Matplotlib. Therefore, the new backend will have to pre-process the numerical data to make it compatible with the chosen plotting library. Keep in mind that future SymPy versions may improve the `*Series` classes in order to return numerical data "non-matplotlib-centric", hence if you code a new backend you have the responsibility to check if its working on each SymPy release.

Please explore the *MatplotlibBackend* (page 2868) source code to understand how a backend should be coded.

See also:

MatplotlibBackend (page 2868)

class sympy.plotting.plot.**MatplotlibBackend**(*parent*)

This class implements the functionalities to use Matplotlib with SymPy plotting functions.

static get_segments(*x*, *y*, *z=None*)

Convert two list of coordinates to a list of segments to be used with Matplotlib's `LineCollection`.

Parameters

x : list

List of x-coordinates

y

[list] List of y-coordinates

z

[list] List of z-coordinates for a 3D line.

process_series()

Iterates over every Plot object and further calls _process_series()

class sympy.plotting.plot.**TextBackend**(*parent*)

Pyglet Plotting

This is the documentation for the old plotting module that uses pyglet. This module has some limitations and is not actively developed anymore. For an alternative you can look at the new plotting module.

The pyglet plotting module can do nice 2D and 3D plots that can be controlled by console commands as well as keyboard and mouse, with the only dependency being pyglet.

Here is the simplest usage:

```
>>> from sympy import var
>>> from sympy.plotting.pygletplot import PygletPlot as Plot
>>> var('x y z')
>>> Plot(x*y**3-y*x**3)
```

To see lots of plotting examples, see `examples/pyglet_plotting.py` and try running it in interactive mode (`python -i plotting.py`):

```
$ python -i examples/pyglet_plotting.py
```

And type for instance `example(7)` or `example(11)`.

See also the Plotting Module wiki page for screenshots.

Plot Window Controls

Camera	Keys
Sensitivity Modifier	SHIFT
Zoom	R and F, Page Up and Down, Numpad + and -
Rotate View X,Y axis	Arrow Keys, A,S,D,W, Numpad 4,6,8,2
Rotate View Z axis	Q and E, Numpad 7 and 9
Rotate Ordinate Z axis	Z and C, Numpad 1 and 3
View XY	F1
View XZ	F2
View YZ	F3
View Perspective	F4
Reset	X, Numpad 5

Axes	Keys
Toggle Visible	F5
Toggle Colors	F6

Window	Keys
Close	ESCAPE
Screenshot	F8

The mouse can be used to rotate, zoom, and translate by dragging the left, middle, and right mouse buttons respectively.

Coordinate Modes

`Plot` supports several curvilinear coordinate modes, and they are independent for each plotted function. You can specify a coordinate mode explicitly with the 'mode' named argument, but it can be automatically determined for cartesian or parametric plots, and therefore must only be specified for polar, cylindrical, and spherical modes.

Specifically, `Plot(function arguments)` and `Plot.__setitem__(i, function arguments)` (accessed using array-index syntax on the `Plot` instance) will interpret your arguments as a cartesian plot if you provide one function and a parametric plot if you provide two or three functions. Similarly, the arguments will be interpreted as a curve is one variable is used, and a surface if two are used.

Supported mode names by number of variables:

- 1 (curves): parametric, cartesian, polar
- 2 (surfaces): parametric, cartesian, cylindrical, spherical

```
>>> Plot(1, 'mode=spherical; color=zfade4')
```

Note that function parameters are given as option strings of the form "`key1=value1; key2 = value2`" (spaces are truncated). Keyword arguments given directly to plot apply to the plot itself.

Specifying Intervals for Variables

The basic format for variable intervals is [var, min, max, steps]. However, the syntax is quite flexible, and arguments not specified are taken from the defaults for the current coordinate mode:

```
>>> Plot(x**2) # implies [x,-5,5,100]
>>> Plot(x**2, [], []) # [x,-1,1,40], [y,-1,1,40]
>>> Plot(x**2-y**2, [100], [100]) # [x,-1,1,100], [y,-1,1,100]
>>> Plot(x**2, [x,-13,13,100])
>>> Plot(x**2, [-13,13]) # [x,-13,13,100]
>>> Plot(x**2, [x,-13,13]) # [x,-13,13,100]
>>> Plot(1*x, [], [x], 'mode=cylindrical') # [unbound_theta,0,2*Pi,40], [x,-1,
↪1,20]
```

Using the Interactive Interface

```
>>> p = Plot(visible=False)
>>> f = x**2
>>> p[1] = f
>>> p[2] = f.diff(x)
>>> p[3] = f.diff(x).diff(x)
>>> p
[1]: x**2, 'mode=cartesian'
[2]: 2*x, 'mode=cartesian'
[3]: 2, 'mode=cartesian'
>>> p.show()
>>> p.clear()
>>> p
<blank plot>
>>> p[1] =  x**2+y**2
>>> p[1].style = 'solid'
>>> p[2] = -x**2-y**2
>>> p[2].style = 'wireframe'
>>> p[1].color = z, (0.4,0.4,0.9), (0.9,0.4,0.4)
>>> p[1].style = 'both'
>>> p[2].style = 'both'
>>> p.close()
```

Using Custom Color Functions

The following code plots a saddle and color it by the magnitude of its gradient:

```
>>> fz = x**2-y**2
>>> Fx, Fy, Fz = fz.diff(x), fz.diff(y), 0
>>> p[1] = fz, 'style=solid'
>>> p[1].color = (Fx**2 + Fy**2 + Fz**2)**(0.5)
```

The coloring algorithm works like this:

1. Evaluate the color function(s) across the curve or surface.
2. Find the minimum and maximum value of each component.
3. Scale each component to the color gradient.

When not specified explicitly, the default color gradient is $f(0.0) = (0.4, 0.4, 0.4) \rightarrow f(1.0) = (0.9, 0.9, 0.9)$. In our case, everything is gray-scale because we have applied the default color gradient uniformly for each color component. When defining a color scheme in this way, you might want to supply a color gradient as well:

```
>>> p[1].color = (Fx**2 + Fy**2 + Fz**2)**(0.5), (0.1,0.1,0.9), (0.9,0.1,0.1)
```

Here's a color gradient with four steps:

```
>>> gradient = [ 0.0, (0.1,0.1,0.9), 0.3, (0.1,0.9,0.1),
...              0.7, (0.9,0.9,0.1), 1.0, (1.0,0.0,0.0) ]
>>> p[1].color = (Fx**2 + Fy**2 + Fz**2)**(0.5), gradient
```

The other way to specify a color scheme is to give a separate function for each component r, g, b. With this syntax, the default color scheme is defined:

```
>>> p[1].color = z,y,x, (0.4,0.4,0.4), (0.9,0.9,0.9)
```

This maps z->red, y->green, and x->blue. In some cases, you might prefer to use the following alternative syntax:

```
>>> p[1].color = z,(0.4,0.9), y,(0.4,0.9), x,(0.4,0.9)
```

You can still use multi-step gradients with three-function color schemes.

Plotting Geometric Entities

The plotting module is capable of plotting some 2D geometric entities like line, circle and ellipse. The following example plots a circle centred at origin and of radius 2 units.

```
>>> from sympy import *
>>> x,y = symbols('x y')
>>> plot_implicit(Eq(x**2+y**2, 4))
```

Similarly, *plot_implicit()* (page 2847) may be used to plot any 2-D geometric structure from its implicit equation.

Plotting polygons (Polygon, RegularPolygon, Triangle) are not supported directly.

Plotting with ASCII art

sympy.plotting.textplot.**textplot**(*expr*, *a*, *b*, *W=55*, *H=21*)

Print a crude ASCII art plot of the SymPy expression 'expr' (which should contain a single symbol, e.g. x or something else) over the interval [a, b].

Examples

```
>>> from sympy import Symbol, sin
>>> from sympy.plotting import textplot
>>> t = Symbol('t')
>>> textplot(sin(t)*t, 0, 15)
 14 |                                                  ...
    |                                                     .
    |                                                 .
    |                                                      .
    |                                                .
    |                            ...
    |                           /   .               .
    |                          /
    |                         /      .
    |                        .        .            .
1.5 |----.......--------------------------------------------
    |....       \           .          .
```

(continues on next page)

(continued from previous page)

```
    |            \         /                      .
    |             ..      /             .
    |               \    /                       .
    |                ....
    |                                    .
    |                                     .     .
    |
    |                                      .   .
-11 |________________________________________________________
     0                          7.5                        15
```

Stats

SymPy statistics module

Introduces a random variable type into the SymPy language.

Random variables may be declared using prebuilt functions such as Normal, Exponential, Coin, Die, etc… or built with functions like FiniteRV.

Queries on random expressions can be made using the functions

Expression	Meaning
`P(condition)`	Probability
`E(expression)`	Expected value
`H(expression)`	Entropy
`variance(expression)`	Variance
`density(expression)`	Probability Density Function
`sample(expression)`	Produce a realization
`where(condition)`	Where the condition is true

Examples

```
>>> from sympy.stats import P, E, variance, Die, Normal
>>> from sympy import simplify
>>> X, Y = Die('X', 6), Die('Y', 6) # Define two six sided dice
>>> Z = Normal('Z', 0, 1) # Declare a Normal random variable with mean 0, std 1
>>> P(X>3) # Probability X is greater than 3
1/2
>>> E(X+Y) # Expectation of the sum of two dice
7
>>> variance(X+Y) # Variance of the sum of two dice
35/6
>>> simplify(P(Z>1)) # Probability of Z being greater than 1
1/2 - erf(sqrt(2)/2)/2
```

One could also create custom distribution and define custom random variables as follows:

1. If you want to create a Continuous Random Variable:

```
>>> from sympy.stats import ContinuousRV, P, E
>>> from sympy import exp, Symbol, Interval, oo
>>> x = Symbol('x')
>>> pdf = exp(-x) # pdf of the Continuous Distribution
>>> Z = ContinuousRV(x, pdf, set=Interval(0, oo))
>>> E(Z)
1
>>> P(Z > 5)
exp(-5)
```

1.1 To create an instance of Continuous Distribution:

```
>>> from sympy.stats import ContinuousDistributionHandmade
>>> from sympy import Lambda
>>> dist = ContinuousDistributionHandmade(Lambda(x, pdf), set=Interval(0, oo))
>>> dist.pdf(x)
exp(-x)
```

2. If you want to create a Discrete Random Variable:

```
>>> from sympy.stats import DiscreteRV, P, E
>>> from sympy import Symbol, S
>>> p = S(1)/2
>>> x = Symbol('x', integer=True, positive=True)
>>> pdf = p*(1 - p)**(x - 1)
>>> D = DiscreteRV(x, pdf, set=S.Naturals)
>>> E(D)
2
>>> P(D > 3)
1/8
```

2.1 To create an instance of Discrete Distribution:

```
>>> from sympy.stats import DiscreteDistributionHandmade
>>> from sympy import Lambda
>>> dist = DiscreteDistributionHandmade(Lambda(x, pdf), set=S.Naturals)
>>> dist.pdf(x)
2**(1 - x)/2
```

3. If you want to create a Finite Random Variable:

```
>>> from sympy.stats import FiniteRV, P, E
>>> from sympy import Rational, Eq
>>> pmf = {1: Rational(1, 3), 2: Rational(1, 6), 3: Rational(1, 4), 4:␣
↪Rational(1, 4)}
>>> X = FiniteRV('X', pmf)
>>> E(X)
29/12
>>> P(X > 3)
1/4
```

3.1 To create an instance of Finite Distribution:

```
>>> from sympy.stats import FiniteDistributionHandmade
>>> dist = FiniteDistributionHandmade(pmf)
>>> dist.pmf(x)
Lambda(x, Piecewise((1/3, Eq(x, 1)), (1/6, Eq(x, 2)), (1/4, Eq(x, 3) | Eq(x,␣
↪4)), (0, True)))
```

Random Variable Types

Finite Types

sympy.stats.**DiscreteUniform**(*name, items*)

Create a Finite Random Variable representing a uniform distribution over the input set.

Parameters

items : list/tuple

Items over which Uniform distribution is to be made

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import DiscreteUniform, density
>>> from sympy import symbols
```

```
>>> X = DiscreteUniform('X', symbols('a b c')) # equally likely over a,␣
↪b, c
>>> density(X).dict
{a: 1/3, b: 1/3, c: 1/3}
```

```
>>> Y = DiscreteUniform('Y', list(range(5))) # distribution over a range
>>> density(Y).dict
{0: 1/5, 1: 1/5, 2: 1/5, 3: 1/5, 4: 1/5}
```

References

[R803], [R804]

sympy.stats.**Die**(*name, sides=6*)

Create a Finite Random Variable representing a fair die.

Parameters

sides : Integer

Represents the number of sides of the Die, by default is 6

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import Die, density
>>> from sympy import Symbol
```

```
>>> D6 = Die('D6', 6) # Six sided Die
>>> density(D6).dict
{1: 1/6, 2: 1/6, 3: 1/6, 4: 1/6, 5: 1/6, 6: 1/6}
```

```
>>> D4 = Die('D4', 4) # Four sided Die
>>> density(D4).dict
{1: 1/4, 2: 1/4, 3: 1/4, 4: 1/4}
```

```
>>> n = Symbol('n', positive=True, integer=True)
>>> Dn = Die('Dn', n) # n sided Die
>>> density(Dn).dict
Density(DieDistribution(n))
>>> density(Dn).dict.subs(n, 4).doit()
{1: 1/4, 2: 1/4, 3: 1/4, 4: 1/4}
```

sympy.stats.**Bernoulli**(*name, p, succ=1, fail=0*)

Create a Finite Random Variable representing a Bernoulli process.

Parameters

p : Rational number between 0 and 1

Represents probability of success

succ : Integer/symbol/string

Represents event of success

fail : Integer/symbol/string

Represents event of failure

Returns

RandomSymbol

Examples

```
>>> from sympy.stats import Bernoulli, density
>>> from sympy import S
```

```
>>> X = Bernoulli('X', S(3)/4) # 1-0 Bernoulli variable, probability = 3/
↪4
>>> density(X).dict
{0: 1/4, 1: 3/4}
```

```
>>> X = Bernoulli('X', S.Half, 'Heads', 'Tails') # A fair coin toss
>>> density(X).dict
{Heads: 1/2, Tails: 1/2}
```