Table of contents

`contourplot` – **contour and implicit plots**

!! This file has not been edited yet !!

Call(s):

- `contourplot(`<*scene-option*`, ...`>*object*`, ...)`

Parameters:

`expr_x, expr_y, expr_z` — expressions
`identifier_1, identifier_2` — identifier
`expr_1, expr_2, expr_3` — expressions

Options:

Related Functions:

Details:

- `contourplot` serves for the graphical representation of contour plots of surfaces and plots of implicit defined functions. A three-dimensional surface is described in a call of the above function by the following expression sequence:

```
[expr_x, expr_y, expr_z], var_u = [expr_1, expr_2],
                var_v = [expr_3, expr_4]
```

  Here, `expr_x`, `expr_y` and `expr_z` are MuPAD expressions depending on the variables `var_u` and `var_v`. These expressions are used to describe the `x`-, `y`- and the `z`-coordinate of the different sample points of the surface. `var_u` and `var_v` have to be identifier. The expressions `expr_1` and `expr_2` specify the lower and the upper bound of the range, in which the variable `var_u` is evaluated. The same is valid for the expressions `expr_3`, `expr_4` and the variable `var_v`.

  The contour lines are drawn on the bottom of the viewing box. The contour lines are drawn with respect to the height when using the option `Style = Attached`.

  Apart from the different objects the user can give many options in order to specify the graphical representation of the scene as well as the individual objects. Since a call of the above routine serves for the generation of a three-dimensional surface plot, all options available for such a scene and objects of the mode `Surface` can be used here.

The procedures of the library `plotlib` can be exported by `export(plotlib)`, so that the short notation `contourplot` instead of `contourplot` can be used.

Example 1.

```
>> plotlib::contourplot([[x,y,exp(x*y)],x=[-1,1],y=[-1,1]]):

>> plotlib::contourplot([[x,y,exp(x*y)],x=[-1,1],y=[-1,1],Colors=[Height]])

>>  [notest]
   plotlib::contourplot([[x,y,exp(x*y)],x=[-1,1],y=[-1,1],Style=Attached]):

>> // Contour plot of two functions
   plotlib::contourplot(
       [[x,y,sin(x*y)],x=[-PI,PI],y=[-PI,PI],Grid=[20,20]],
       [[x,y,x + 2*y] ,x=[-PI,PI],y=[-PI,PI],Colors=[Flat,RGB::Blue]]
   ):

>> // Plot of the implicit function defined by (x^2+y^2)^3-
(x^2-y^2)^2 = 0
   plotlib::contourplot([[x,y,(x^2+y^2)^3-(x^2-y^2)^2],x=[-
1,1],y=[-1,1],
                        Contours=[0], Grid=[20,20]]):
```

Background:

⊞

`dataplot` – 2- and 3-dimensional plot of list of data

!! This file has not been edited yet !!

Call(s):

⊞ `dataplot(type,list<,Colors=colors>`
`<,Titles=titles>)`

Parameters:

`colors` — a list of RGB values
`type` — the plotting type
`list` — a list or list of lists of data
`titles` — a list of strings

Options:

Related Functions:

Details:

- `dataplot`creates a 2- or 3-dimensional plot of a list of numeric values. The first argument specifies how to plot the data - the plotting type. The second argument is a list or a list of lists of numeric values, depending on the plotting type. The next two arguments, which are optional, are used to specify colors for the objects and to specify titles for the objects. You can move the titles inside VCam. Just click on the title and move it with the cursor.

  The following plotting types are available:

  - **Piechart and Piechart3d:**
    The second argument is a list `[d_1,...,d_n]`. A piechart is drawn. You can specify a color for every `d_i`. The current implementation of piechart drawing is however quite slow.
  - **Line, Curve, Column, Beam:**
    These plotting types are used to create a 2-d plotting from data. The second argument is a list of lists of numeric values. Each of the values in the list `[y_1,...,y_n]` are interpreted as the y-value corresponding to the x-values `1,...,n`, i.e. the points `(1,y_1),...,(n,y_n)` are plotted. The values of one list defines an object. You can specify a color for every object with the third argument. Curve computes the Lagrange interpolation polynomial to plot the curve.
  - **Plain and Surface:**
    These plotting types are used to create a 3-d surface plotting from data. The second argument is a list of lists of numeric values. Each of the values in the list `[z_i_1,...,z_i_n]` are interpreted as the z-value corresponding to the y-value i and x-values `1,...,n`, i.e. the points `(1,i,z_1),...,(n,i,z_n)`. Consists the list of `m` lists `i=1,...,m`. Surface computes the 2-dimensional Lagrange interpolation polynomial to plot the surface.

Example 1.

```
>> export(plotlib): export(RGB):
   dataplot(Piechart, [5,12,38,14,25]):

>> dataplot(Piechart3d, [5,12,38], Colors=[RoyalBlue,VioletRed,GreenPale]):
```

```
>> dataplot(Line,
            [[5,10,24,-3],[6,5,2,18],[19,45,12,-10]],
            Colors=[Red,Green,Blue]):

>> dataplot(Curve,
            [[5,10,24,-3],[6,5,2,18],[19,45,12,-10]],
            Colors=[Red,Green,Blue]):

>> dataplot(Column,
            [[5,10,24,-3],[6,5,2,18],[19,45,12,-10]],
            Colors=[Red,Green,Blue]):

>> dataplot(Beam,
            [[5,10,24,-3],[6,5,2,18],[19,45,12,-10]],
            Colors=[Red,Green,Blue]):

>> dataplot(Column, [[5,10,24,-3,6,5,2,18]]):

>> dataplot(Plain,
            [[5,10,24,-3],[6,5,2,18],[19,45,12,-10]],
            Colors=[Red,Red,Green]):

>> dataplot(Surface, [[5,10,24,-3],[6,5,2,18],[19,45,12,-10]]):
```

Background:

⊞

`densityplot` – two-dimensional density plots

!! This file has not been edited yet !!

Call(s):

⊞ `densityplot(`*<scene-option*`, ...`*>object*`, ...)`

Parameters:

`expr_x, expr_y, expr_z`	— expressions
`identifier_1, identifier_2`	— identifier
`expr_1, expr_2, expr_3`	— expressions

Options:

Related Functions:

Details:

- `densityplot` serves for the graphical representation of two dimensional density plots of surfaces.

  A three-dimensional surface is described in a call of the above function by the following expression sequence:

  ```
  [expr_x, expr_y, expr_z], var_u = [expr_1, expr_2],
                var_v = [expr_3, expr_4]
  ```

  Here, `expr_x`, `expr_y` and `expr_z` are MuPAD expressions depending on the variables `var_u` and `var_v`. These expressions are used to describe the `x`-, `y`- and the `z`-coordinate of the different sample points of the surface. `var_u` and `var_v` have to be identifier. The expressions `expr_1` and `expr_2` specify the lower and the upper bound of the range, in which the variable `var_u` is evaluated. The same is valid for the expressions `expr_3`, `expr_4` and the variable `var_v`.

  Apart from the different objects the user can give many options in order to specify the graphical representation of the scene as well as the individual objects. Since a call of the above routine serves for the generation of a three-dimensional surface plot, all options available for such a scene and objects of the mode `Surface` can be used here.

  The procedures of the library `plotlib` can be exported by `export(plotlib)`, so that the short notation `densityplot` instead of `densityplot` can be used.

Example 1.

```
>> plotlib::densityplot([Mode=Surface,
                [u,v,1/2*sin(u*u + v*v)],
                u=[0,PI], v=[0, PI]]):
```

Background:

-

`implicitplot` – implicit plot of smooth functions

`implicitplot` is used to get a plot of $f = 0$ for a smooth f from $\mathbb{R}^2 \to \mathbb{R}$. f must be regular almost everywhere on this curve.

Call(s):

- `implicitplot(expr,x=a..b,y=c..d<, options>)`
- `implicitplot([expr, ...],x=a..b,y=c..d,<, options>)`

Parameters:

`expr`	—	Function(s) to plot, given as expression(s) in two indeterminates
`x,y`	—	Indeterminates used in `expr`
`a..b,c..d`	—	Ranges to plot

Options:

`Grid` `= gridval`	—	Grid division to use for finding starting points
`Colors` `= [col1, ...]`	—	Colors used for plotting the components.
`MinStepsize` `= hmin`	—	The minimum step-size for tracing a contour
`MaxStepsize` `= hmax`	—	The maximum step-size for tracing a contour
`StartingStepsize` `= hstart`	—	The step-size the iteration starts with
`Precision` `= eps`	—	Precision of the Newton iteration
`Ticks` `= [tx, ty]`	—	Number of ticks on the coordinate axes
`Contours` `= [c1, ...]`	—	Contours to plot
`Splines` `= Boolean`	—	If set to `TRUE`, the contours will be plotted with cubic splines; otherwise, straight lines will be used. Default: `FALSE`.
`Factor` `= Boolean`	—	If set to `TRUE`, each function will be (attempted to be) factorized prior to iterating. This may improve the results. Default: `FALSE`.

Return Value: `implicitplot` returns the value of type `DOM_NULL`.

Related Functions: `plotlib::contourplot`

Details:

- `implicitplot` plots $f = 0$ by a curve tracking method. For this, it first generates startpoints with a Newton iteration starting ad grid points (the number of grid points can be controlled with the optional parameter) and then iteratively applies the implicit function lemma to get a local approximation to the curve. This approximation is then improved with another Newton step. On hitting a point where f is not regular, the iteration stops.

Option <*`Grid`* `= gridval`>:

- `gridval` must either be a list of two positive integers, the number of divisions in the two coordinates, or a single integer, which is equivalent to repeating this integer twice.
- Increase this number if you suspect not all components are found. The default is `[5, 5]`.

Option <*`Colors`* `= [col1, ...]`>:

- The colors used for plotting the components. A list is expected, each element of which must be a valid argument for the `Color=[Flat,...]` option of `plot2d`.
- Default: `[[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0], [1.0, 1.0, 0.0], [1.0, 0.0, 1.0], [0.0, 1.0, 1.0], [1.0, 1.0, 1.0]]`.

Option <*`MinStepsize`* `= hmin`>:

- The minimum step-size for the iteration following a contour. This is a lower limit for the adaptive iteration to avoid spending a lot of time without real progress.
- The default is 1/1.000.000 of the minimum of width or height of the area under consideration.
- Increase this number if you think the algorithm gets stuck in a place unimportant to your application.

Option <*`MaxStepsize`* `= hmax`>:

- *`MaxStepsize`* is complementary to *`MinStepsize`* in that it defines the maximum step-width for the iteration.
- If this value is set too high, the algorithm may skip over details of the curves; if it is set lower than needed, the computation takes longer.
- The default is 1/100 of the shorter of width and height.

Option <*StartingStepsize* = hstart>:

- The step-size the iteration starts with. The default is $1/1.000$ of the shorter side of the area.

Option <*Precision* = eps>:

- This floating-point value indicates the relative precision which should be achieved in the Newton iteration.
- Lower values may lead to more accurate results, but slow down the computation. Values smaller than $10.0^{-DIGITS}$ may cause the function not to return.
- The default is $10.0^{2-DIGITS}$.

Option <*Ticks* = [tx, ty]>:

- The number of ticks to be displayed on the coordinate axes. Either indicate values for both axes as a list or a single integer which is taken for both axes.
- The default is `[10, 10]`.

Option <*Contours* = [c1, ...]>:

- A list of values for which the functions implicitly defined by $f(x, y) = c_i$ should be plotted.
- Default: `[0]`.

Example 1. We plot the family $x = y^2 + c$ for c in $-5, \ldots, 5$.

```
>> plotlib::implicitplot(y^2 - x, x = -1..25, y = -5..5, Con-
tours = [$-5..5])
```

Example 2. To demonstrate how to plot multiple implicit functions, we plot $y = x^2$, $y = x$ and $x = y^2$.

```
>> plotlib::implicitplot([x^2 - y, x - y, x - y^2], x = -4..4, y = -
4..4)
```

Example 3. Let's have a look at elliptic curves.

```
>> plotlib::implicitplot((x^3 + x + 2) - y^2, x = -5..5, y = -
5..5)
```

Example 4. `implicitplot` handles quite complex expressions:

```
>> plotlib::implicitplot((1-0.99*exp(x^2+y^2))*(x^10-1-y),
                         x=-1.25..1.25,y=-1.1..2):
>> F2 := (x,y) -> x^4*y^4+sin(x)*cos(y)-(x-1)*(y-2)*exp(-x^2):
   plotlib::implicitplot(F2(x,y),x=-10..10,y=-10..10):
>> plotlib::implicitplot(F2(x,y),x=-3.5..1,y=-10..2.5):
>> plotlib::implicitplot(F2(x,y),x=-10..-6,y=-1..1):
>> plotlib::implicitplot(F2(x,y),x=-0.5..0.5,y=-25..-20):
   delete F2:
```

In some cases, `DIGITS` must be increased to get a correct result. In the following example, problems occur around the origin with the default setting of `DIGITS` when a small region is to be displayed. First, we display the whole picture:

```
>> F3 := (x,y) -> y*(3*x^2-y^2)-(x^2+y^2)^2:
   plotlib::implicitplot(F3(x, y), x = -1..1, y = -1.3..0.7):
```

Near the origin, numeric cancellation occurs. If you try to depict a small area around the origin of the above curve, you need to increase `DIGITS`.

```
>> delete DIGITS:
   plotlib::implicitplot(F3(x, y), x = -0.005..0.005, y = -
0.005..0.005):
```

```
>> DIGITS := 15:
   plotlib::implicitplot(F3(x, y), x = -0.005..0.005, y = -
0.005..0.005):
   delete DIGITS:
```

Example 5. When plotting functions with many components in their set of zeroes, you should use the option *`Grid`* to increase the number of found components. This situation cannot be detected automatically.

```
>> plotlib::implicitplot(sin(x^3 - x*y), x = -10..10, y = -
5..5)
>> plotlib::implicitplot(sin(x^3 - x*y), x = -10..10, y = -
5..5, Grid = 50);
```

Background:

- Curve Tracking algorithms are usually found in numerics to track stable and instable manifolds of dynamic systems and in homotopy methods for finding roots of highly complicated functions.

`cylindricalplot` – cylindrical coordinates

!! This file has not been edited yet !!

Call(s):

- `cylindricalplot(`<*scene-option*`, ...`>*object*`, ...)`

Parameters:

`identifier_1, identifier_2` — identifier
`expr_r, expr_phi, expr_z` — expressions
`expr_1, expr_2, expr_3` — expressions

Options:

Related Functions:

Details:

- `cylindricalplot` serves for the graphical representation of three-dimensional surfaces in cylindrical coordinates. With this any number of objects in cylindrical coordinates can be grouped to a scene and displayed.

  A three-dimensional surface in cylindrical coordinates is described in a call of the above function by the following expression sequence:

```
[expr_r, expr_phi, expr_z], var_phi = [expr_1,
expr_2],
                var_z = [expr_3, expr_4]
```

  Here, `expr_r`, `expr_phi` and `expr_z` are MuPAD expressions depending on the variables `var_phi` and `var_z`. These expressions are used to describe the radius, the angle and the `z`-coordinate of the different sample points of the surface. `var_phi` and `var_z` have to be identifier. The expressions `expr_1` and `expr_2` specify the lower and the upper bound of the range, in which the variable `var_phi` is evaluated. The same is valid for the expressions `expr_3`, `expr_4` and the variable `var_z`.

Apart from the different objects the user can give many options in order to specify the graphical representation of the scene as well as the individual objects. Since a call of the above routine serves for the generation of a three-dimensional surface plot, all options available for such a scene and objects of the mode `Surface` can be used here.

Furthermore the procedures can be exported by `export(plotlib)`, so that the short notation `cylindricalplot` instead of `cylindricalplot` can be used.

Example 1.

```
>> export(plotlib):
   cylindricalplot(Axes = Box, Ticks = 0,
                        [[1, u, z],
                         u = [-PI, PI],  z = [-1, 1],
                         Grid = [20, 20],
                         Style = [HiddenLine, Mesh]
                        ]);

>> export(plotlib):
   cylindricalplot(Axes = Corner, Ticks = 0,
                        [[u, u, z],
                         u = [-PI, PI], z = [-PI, PI],
                         Grid = [30, 30],
                         Color = [Height],
                         Style = [ColorPatches, AndMesh]
                        ],
                        [[-u, u, z],
                         u = [-PI, PI], z = [-2, 2],
                         Grid = [30, 30],
                         Color = [Height],
                         Style = [ColorPatches, AndMesh]
                        ]);
```

Background:

`fieldplot` – **Vectorfields**

!! This file has not been edited yet !!

Call(s):

- `fieldplot(`*<scene-option*`, ...>`*object*`, ...)`

Parameters:

`expr_x, expr_y` — expressions
`int_1, int_2` — positive integers greater than 1
`identifier_1, identifier_2` — identifier
`expr_1, expr_2, expr_3` — expressions

Options:

Related Functions:

Details:

- `fieldplot` serves for the graphical representation of two-dimensional vectorfields. With this any number of such objects can be grouped to a scene and displayed.

  A two-dimensional vectorfield is described by the following expression sequence:

  ```
  [expr_x, expr_y], var_x = [expr_1, expr_2],
              var_y = [expr_3, expr_4]
  ```

  Here, `expr_x` and `expr_y` are expressions depending on the variables `var_x` and `var_y`. These expressions are used to specify the vectorfield to be plotted. `var_x` and `var_y` have to be identifier. The expressions `expr_1` and `expr_2` specify the lower and the upper bound of the range, in which the variable `var_x` is evaluated. The same is valid for the expressions `expr_3`, `expr_4` and the identifier `var_y`.

  Apart from the specification of the different vectorfields the user can give many options in order to influence the graphical representation of the scene and the corresponding objects. Since a call of the above routine internally is represented as a two-dimensional scene containing only objects, which consist out of graphical primitives, all options available for such scenes and objects can be chosen here.

  Furthermore these procedures can be exported, by use of `export(plotlib)`, so that the short notation `fieldplot` instead of `fieldplot` can be used.

Example 1.

```
>> export(plotlib):
   fieldplot(Axes = Origin, Ticks = 0,
                  [[1,sin(y+cos(x))],
                   x = [-PI, PI], y = [-PI, PI],
                   Grid = [30, 30], Color = [Height]
                  ]);
```

```
>> export(plotlib):
   fieldplot(Axes = Origin, Ticks = 0,
                  [[-y^2, x^2],
                   x = [-4, 4], y = [-4, 4],
                   Grid = [30, 30], Color = [Height]
                  ]);
```

Background:

⊞

`inequalityplot` – two dimensional plot of inequalities

!! This file has not been edited yet !!

Call(s):

```
⊞ inequalityplot(inequalities ,left..right ,bot-
                 tom..top <n> <Colors = [color1,
                 color2, color3]> <Sceneoptions =
                 [option_sequence]> )
```

Parameters:

`color1, color2, color3`	—	color names (RGB values)
`option_sequence`	—	a sequence of equations representing the scene options of a `plot2d` command
`left, right, bottom, top`	—	real numerical values
`inequalities`	—	a list `[f1,f2,..]` of real valued functions of two arguments
`n`	—	optional non-negative integer

Options:

Related Functions:

Details:

- `inequalityplot` serves for displaying points (x, y) in the rectangle

  $$Q = [\texttt{left}, \texttt{right}] \times [\texttt{bottom}, \texttt{top}]$$

  satisfying the inequalities

  $$f_1(x, y) \geq 0 \quad \text{and} \quad f_2(x, y) \geq 0 \quad \text{and} \quad \ldots \qquad (1)$$

  specified by the list of functions `inequalities`$= [f_1, f_2, \ldots]$.

  The rectangle Q is divided into $2^n \times 2^n$ subrectangles. The default value for n is 6.

  A subrectangle is displayed with the color `color1`, if all its points (x, y) satisfy $f_1(x, y) > 0$ and $f_2(x, y) > 0$ etc. Consequently, all points of this color are guaranteed to satisfy (1).

  A subrectangle is displayed with `color3`, if there is a function f in `inequalities` such that all points in the subrectangle satisfy $f(x, y) < 0$. Consequently, all points of this color are guaranteed not to satisfy (1).

  The remaining subrectangles are displayed with the color `color2`. They cover the boundary of the region defined by (1).

  The default colors are `Colors = [RGB::Green, RGB::Yellow, RGB::Red]`.

  The plot is constructed via `plot2d(option_sequence, [Mode = List, [..]])`, where `[..]` is a list of polygons computed by `inequalityplot`. `option_sequence` may consist of all valid scene options for `plot2d`. The default values are

  `Sceneoptions=[Scaling=UnConstrained, Labeling=TRUE, Labels=["",""]]`.

  Interval arithmetic is used to check the inequalities. The input functions must be suitable for this kind of arithmetic, i.e., the calls

  `f1(Dom::Interval(left..right), Dom::Interval(bottom..top))`

  etc. must produce valid intervals.

Example 1.

```
>> f1:= proc(x,y) begin x^2 + y^2 - 1 end_proc:

>> plotlib::inequalityplot([f1], -1..1, -1..1, 5);

>> f2:= (x,y) -> cos(x) - y: f3:= (x,y) -> cos(x) + y:
```

```
>> plotlib::inequalityplot([f2, f3], -PI..PI, -2..2, 5);

>> scene_options:= [Scaling=Constrained, Labeling=TRUE, Axes=Box,
                    BackGround=RGB::White, ForeGround=RGB::Black]:

>> colors:=[RGB::Red, RGB::Black, RGB::White]:

>> plotlib::inequalityplot([f1, f2, f3], -2..2, -1..1, 5,
            Colors=colors, Sceneoptions=scene_options);
```

Background:

-

`polarplot` – **polar coordinates**

!! This file has not been edited yet !!

Call(s):

- `polarplot(`*<scene-option*`, ...>`*object*`, ...)`

Parameters:

`identifier_1, identifier_2` — identifier
`expr_r, expr_phi` — expressions
`expr_1, expr_2, expr_3` — expressions

Options:

Related Functions:

Details:

- `polarplot` serves for the graphical representation of two-dimensional curves in polar coordinates. With this any number of objects in polar coordinates can be grouped to a scene and displayed.

  A two-dimensional curve in polar coordinates is described in a call of the above function by the following expression sequence:

  ```
  [expr_r, expr_phi], var_phi = [expr_1, expr_2]
  ```

Here, `expr_r` and `expr_phi` are MuPAD expressions depending on the variable `var_phi`. These expressions are used to describe the radius as well as the angle of the different sample points of the curve. `var_phi` has to be an identifier. The expressions `expr_1` and `expr_2` specify the lower and the upper bound of the range, in which the variable `var_phi` is evaluated.

Apart from the different objects the user can give many options in order to specify the graphical representation of the scene as well as the individual objects. Since a call of the above routine serves for the generation of a two-dimensional curve, all options available for such a scene and objects of the mode `Curve` can be used here.

These procedures can be exported, by use of `export(plotlib)`, so that the short notation `polarplot` instead of `polarplot` can be used.

Example 1.

```
>> export(plotlib):
   polarplot(Axes = None, Ticks = 0,
                  [[Phi, Phi],
                   Phi = [0, 4*PI], Grid = [100]
                  ]);
```

Background:

`polygonplot` – plotting of 2-dimensional polygons

!! This file has not been edited yet !!

Call(s):

- `polygonplot(`<*scene-option*`, ...`>*polygons*
  <*option*><`,specification`> `)`

Parameters:

`specification`	— Options `Convex` or `Concave`
`r, g, b`	— real numbers between 0 and 1
`polygon_i`	— `DOM_POLYGON`
`identifier, expr`	— expressions

Options:

Related Functions:

Details:

- `polygonplot` serves for graphical representation of 2-dimensional polygons.

  With *option* it is optionally possible to color the border lines of the polygons, whereby `r`, `g` und `b` have to be real values between `0.0` and `1.0` determining the amount of red, green and blue in the RGB color model. The default value of *option* is the list `[0.0, 0.0, 1.0]`.

  To specify the kind of the polygons the options `Convex` and `Concave` can be chosen. If no specification is given then the objects will be considered as arbitrary (e.g. self intersecting) polygons.

  If a polygon is simple, i.e. its borders do not intersect, the option `Concave` will be used. A polygon is convex if each line between two points of the polygon lies inside the polygon.

  If a polygon is either convex or concave it is recommended to specify the corresponding option due to effeciency. However, in cases where this can not be decided a specification should not be given in order to avoid incorrect filling of the objects.

  scene-option can be used to specify the scene. See table Options for a Scene for a complete description of possible values for `identifier` and `expr`.

  This function is part of the library `plotlib` and can be exported with `export(plotlib,polygonplot)` in order to use the short notation `polygonplot`.

Example 1.

```
>> export(plotlib,polygonplot);

>> polygonplot(LineWidth=2,
      Title="Example 1: square",
      [polygon(point(0,0),point(0,1),point(1,1),point(1,0),
      Filled=TRUE,Closed=TRUE)],Convex);

>> polygonplot(LineWidth=2,
      Title="Example 2: self-intersecting 2D-polygon",
      [polygon(point(0,0),point(1,1),point(0,1),point(1,0),
      Color=[1,1,0],Filled=TRUE,Closed=TRUE)],[.7,0,1]);
```

```
>> polygonplot(LineWidth=2,
      Title="Example 3: three different 2D-polygons",
      [polygon(point(0,0),point(1,0),point(2,1),point(1,2),
      point(0,2),Color=[0,.9,.2],Filled=TRUE,Closed=TRUE),
      polygon(point(2,0),point(3,0),point(4,1),point(3,2),
      point(2,2),point(3,1),
      Color=[0,.8,.8],Filled=TRUE,Closed=TRUE),
      polygon(point(4,0),point(6,0),point(6,2),point(4,2),
      point(5,1),Color=[0,0,1],Filled=TRUE,Closed=TRUE)],
      [0,0,0],Concave);
```

It is possible that the algorithm works different by self-intersecting polygons. The following example demonstrates that the hole which comes from the intersection can either be filled or unfilled:

```
>>    plotlib::polygonplot(LineWidth=2,Title=
      "Example 4: Different kinds of self-intersected 2D-polygons",
      [polygon(point(0,3),point(4,3),point(4,1),point(2,1),
      point(2,5),point(1,5),point(1,0),point(5,0),point(5,4),
      point(0,4),Color=[1,0.9,0],Closed=TRUE,Filled=TRUE),
      polygon(point(6,0),point(10,0),point(10,5),point(9,5),
      point(9,1),point(7,1),point(7,3),point(11,3),point(11,4),
      point(6,4),Color=[1,0.9,0],Closed=TRUE,Filled=TRUE)],
      [0.9,0.4,0]);
```

Background:

-

`sphericalplot` – **spherical coordinates**

!! This file has not been edited yet !!

Call(s):

- `sphericalplot(`<*scene-option*, ...>*object*, ...`)`

Parameters:

`identifier_1, identifier_2`	— identifier
`expr_1, expr_2, expr_3`	— expressions
`expr_r, expr_phi, expr_theta`	— expressions

Options:

Related Functions:

Details:

- `sphericalplot` serves for the graphical representation of three-dimensional surfaces in spherical coordinates. With this any number of objects in spherical coordinates can be grouped to a scene and displayed.

  A three-dimensional surface in spherical coordinates is described in a call of the above function by the following expression sequence:

  ```
  [expr_r, expr_phi, expr_z], var_phi = [expr_1,
  expr_2],
                var_theta = [expr_3, expr_4]
  ```

  Here, `expr_r`, `expr_phi` and `expr_theta` are MuPAD expressions depending on the variables `var_phi` and `var_theta`. These expressions are used to describe the radius as well as the horicontal and vertical angle of the different sample points of the surface. `var_phi` and `var_theta` have to be identifier. The expressions `expr_1` and `expr_2` specify the lower and the upper bound of the range, in which the variable `var_phi` is evaluated. The same is valid for the expressions `expr_3`, `expr_4` and the variable `var_theta`.

  Apart from the different objects the user can give many options in order to specify the graphical representation of the scene as well as the individual objects. Since a call of the above routine serves for the generation of a three-dimensional surface plot, all options available for such a scene and objects of the mode `Surface` can be used here.

  The procedures implemented in this library can be exported by `export(plotlib)`, such that instead of `sphericalplot` the short notation `sphericalplot` can be used.

Example 1.

```
>> export(plotlib):
   sphericalplot(Axes = Box, Ticks = 0,
                      [[1, Phi, Theta],
                       Phi = [-PI, PI],  Theta = [0, PI],
                       Grid = [20, 20],
                       Style = [HiddenLine, Mesh]
                      ]);
```

Background:

`xrotate` – Surfaces of revolution (x-axis)

!! This file has not been edited yet !!

Call(s):

- `xrotate(<`*scene-option*`, ...>`*object*`, ...)`

Parameters:

`expr_x, expr_y`	— expressions
`identifier_1, identifier_2`	— identifier
`expr_1, expr_2, expr_3`	— expressions

Options:

Related Functions:

Details:

- `xrotate` serves for the representation of surfaces of revolution of two-dimensional curves around the `x`-axis. With this any number of surfaces of revolution can be grouped to a scene and displayed.

  A surface of revolution is described in a call of the above function by the following expression sequence:

  ```
  [expr_x, expr_y, expr_z], var = [expr_1, expr_2],
                angle = [expr_3, expr_4]
  ```

  Here, `expr_x` and `expr_y` are MuPAD expressions depending on the variable `var`. These expressions are used to describe the two-dimensional curve to be rotated around the `x`-axis. `var` has to be an identifier. The expressions `expr_1` and `expr_2` specify the lower and the upper bound, in which the curve variable `var` is evaluated. Furthermore the expressions `expr_3` and `expr_4` give the range, in which the rotation angle `angle` is evaluated.

  Apart from the different surfaces of revolution the user can give many options in order to influence the graphical representation of the scene and the individual obejcts. Since a call of the above function is internally represented by a three-dimensional scene, all options available for such scenes and objects with the mode `Surface` can be used here.

The procedures implemented in this library can be exported by use of `export(plotlib)`, such that instead of `xrotate` the short notation `xrotate` can be used.

Example 1.

```
>> export(plotlib):
   xrotate(Axes = Box, Ticks = 0,
                [[x, 2*sin(x)], x = [0, 2*PI],
                 angle = [0, 2*PI], Grid = [30, 30]
                ]);
```

Background:

`yrotate` – Surfaces of revolution (y-axis)

!! This file has not been edited yet !!

Call(s):

`yrotate(`<*scene-option*`, ...`>*object*`, ...)`

Parameters:

`expr_x, expr_y`	— expressions
`identifier_1, identifier_2`	— identifier
`expr_1, expr_2, expr_3`	— expressions

Options:

Related Functions:

Details:

- `yrotate` serves for the representation of surfaces of revolution of two-dimensional curves around the `y`-axis. With this any number of surfaces of revolution can be grouped to a scene and displayed.

  A surface of revolution is described in a call of the above function by the following expression sequence:

  ```
  [expr_x, expr_y], var = [expr_1, expr_2],
                 angle = [expr_3, expr_4]
  ```

  Here, `expr_x` and `expr_y` are MuPAD expressions depending on the variable `var`. These expressions are used to describe the two-dimensional curve to be rotated around the `y`-axis. `var` has to be an identifier. The expressions `expr_1` and `expr_2` specify the lower and the upper bound, in which the curve variable `var` is evaluated. Furthermore the expressions `expr_3` and `expr_4` give the range, in which the rotation angle `angle` is evaluated.

  Apart from the different surfaces of revolution the user can give many options in order to influence the graphical representation of the scene and the individual obejcts. Since a call of the above function is internally represented by a three-dimensional scene, all options available for such scenes and object with the mode `Surface` can be used here.

  The procedures implemented in this library can be exported by use of `export(plotlib)`, such that instead of `yrotate` the short notation `yrotate` can be used.

Example 1.

```
>> export(plotlib):
   yrotate(Axes = Box, Ticks = 0,
               [[x, 2*sin(x)], x = [0, 2*PI],
                angle = [0, 3/2*PI], Grid = [30, 30]
               ]);
```

Background: