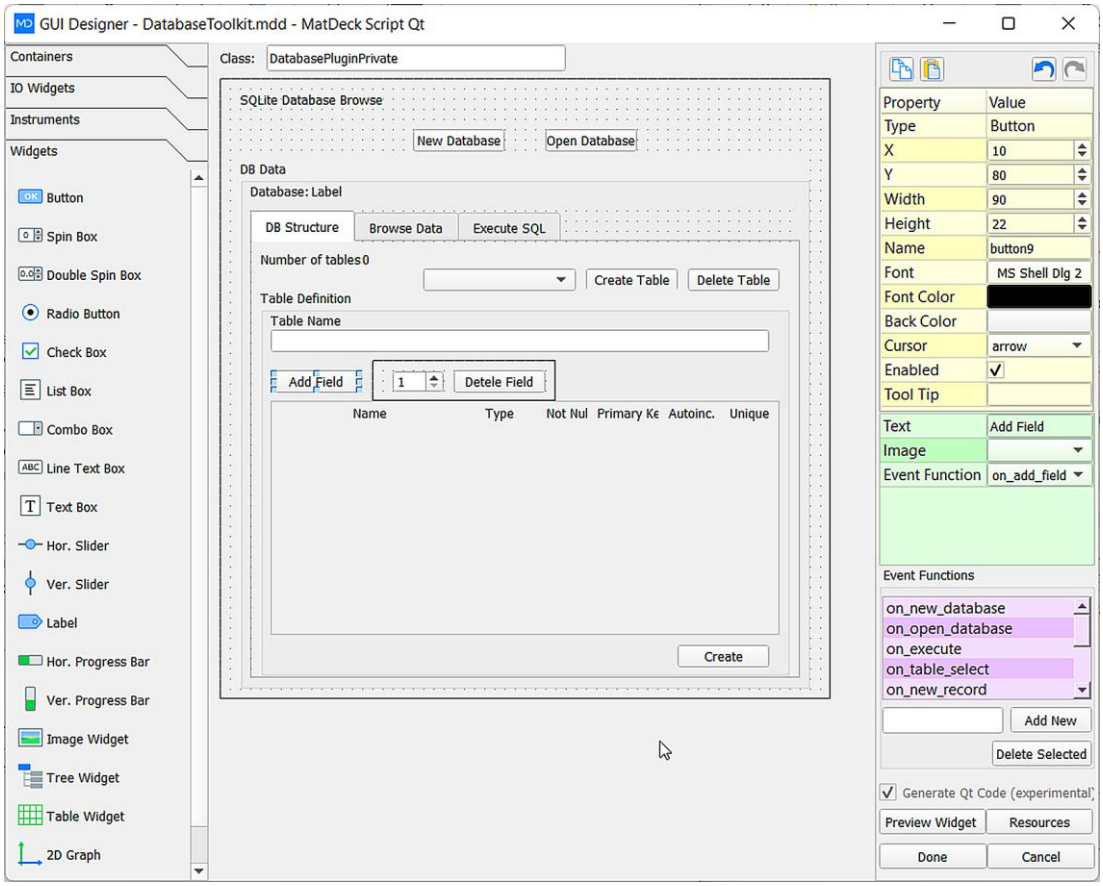
MatDeck

Julia (QML)
GUI Designer

Drag-and-drop GUI building for MatDeck · User manual



Contents

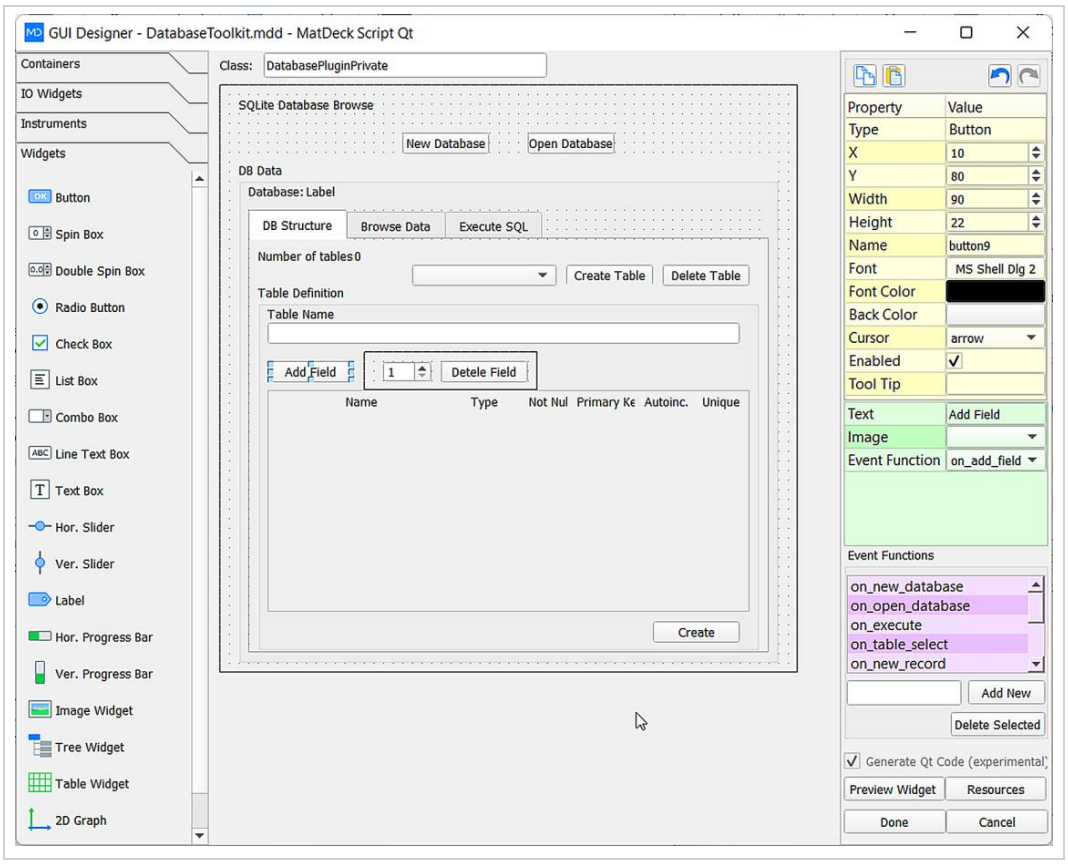
Introduction

The Unlimited Julia (QML) GUI Designer is available in MD Python Designer, Engineering Designer, Visionary Deck, MatDeck Student, MatDeck Academic, MatDeck Home and MatDeck. Its limited form is available in Lite MD Python Designer.

MatDeck products now offer nine GUI Designers — MatDeck Script (Qt runtime), MD Python, PySide2, Tkinter, Custom Tkinter, Kivy, Flet, Julia (QML) and Rust (egui). MD Python Designer and higher MD products have unlimited access to every GUI Designer, whereas Lite MD Python Designer is limited to only a few elements and MatDeck Free does not include any MD GUI Designer.

MD GUI Designers use drag-and-drop visual aids to build a fully functioning GUI with a sleek, modern interface; the widgets and components are generated automatically, leaving no boilerplate work to the user. MD Python Designer ships with every language, library, module and extension already installed, so no extra setup is needed and you can start creating straight away.

MD Python Designer also comes with a specialised IDE and GUI Designer for Julia (QML). As mentioned above, all necessary libraries and modules come preinstalled.



This is what a MD GUI Designer looks like. As you can see, the form itself contains no code and was created solely by dragging and dropping GUI elements into the work area.

> **Please note** The Julia (QML) GUI Designer generates two files. One is a normal Julia (.jl) file; the other, a .mpy file, is used by the designer to save and remember what your GUI looked like and cannot be opened as source.

Available widgets

The Widgets tab of the Julia (QML) GUI Designer always provides the core controls, and the Containers, Inputs & data and Graphs groups add the elements listed below.

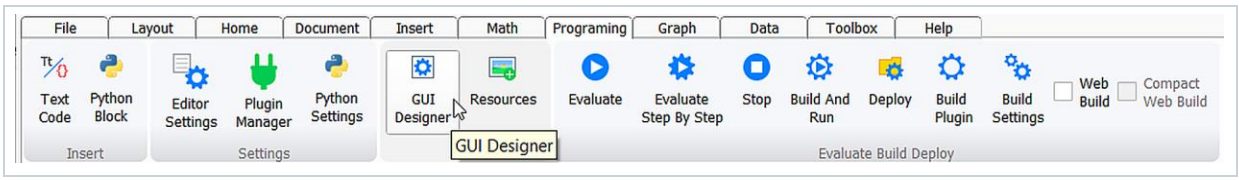
Category	Widgets available
Core widgets	Button, Label, Radio Button, Check Box, Combo Box, Line Edit, Text Box, Horizontal Slider, Vertical Slider, Horizontal Progress Bar
Containers	Root Widget, Group Box, Tab Widget, Scroll Area, Frame, Menu Bar, Tool Bar, Custom Widget
Inputs & data	Spin Box, Double Spin Box, List Box, Table, Tree Widget, Image, Vertical Progress Bar, Circular Progress, Toggle Switch, Spinner, Divider, Icon Button
Graphs	2D Graph, 3D Graph

Feature support: advanced element properties (tooltips, cursor, enabled state, fonts and colours) are supported. The designer does not generate event-function stubs for the Julia (QML) target — you connect behaviour directly in the generated code (see Connecting behaviour).

Opening a MD GUI Designer

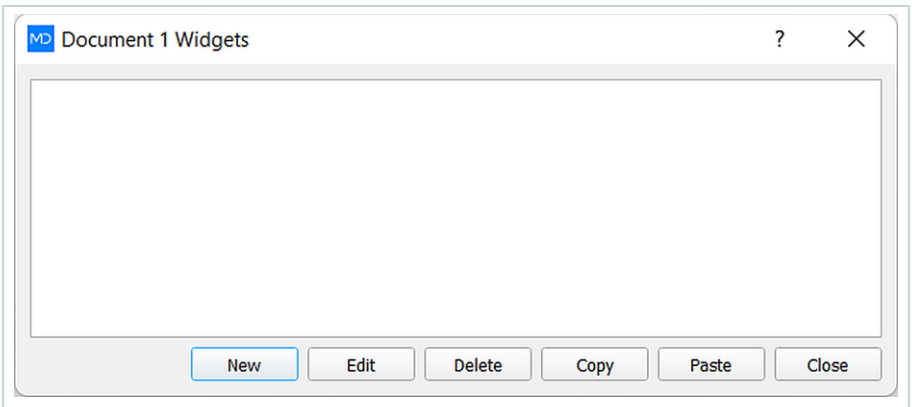
The first thing to do is select the Julia Script. This is done by hovering over the small triangle next to the New icon.

To open the Julia (QML) GUI Designer, go to the Programming tab and press the 'GUI Designer' button (Picture 1).



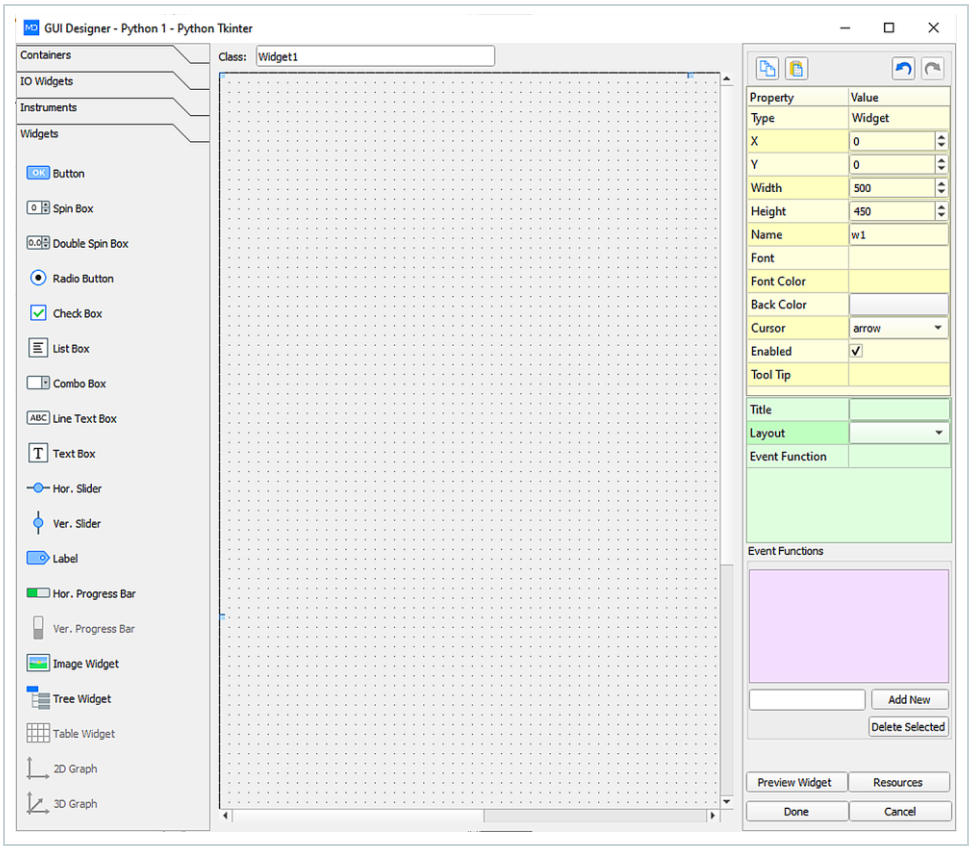
Picture 1: Widgets button

A new window opens containing a list of all widgets created in the current Julia Script (Picture 2). From this window you can create a new widget, or edit or delete an existing one. To open the designer, click New; the Edit button only opens the designer when a widget already exists.



Picture 2: Document widgets editor

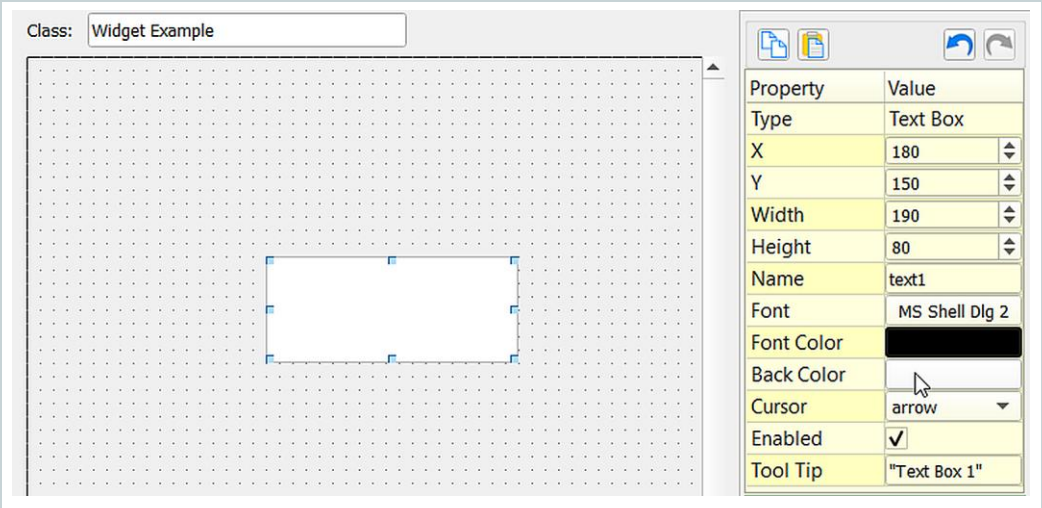
To create a GUI, press New and an empty Julia (QML) GUI Designer tab opens. When doing so, make sure no existing widget is selected in the window. To edit or delete a pre-existing widget, select it first, then press the appropriate button at the bottom of the tab. The default look of an empty designer is shown below.



Picture 3: GUI Designer

On the left side are all the GUI elements; the middle is the working area, where widgets are placed. To place an element, click the preferred GUI element and then click where on the working area you want it. For example, to add a Text Box, click the Text Box button on the left and then click anywhere on the working area.

On the right-hand side, the properties of the selected element are displayed and can be modified (Picture 4). You can also set the widget name by typing it in the field above the main working area.

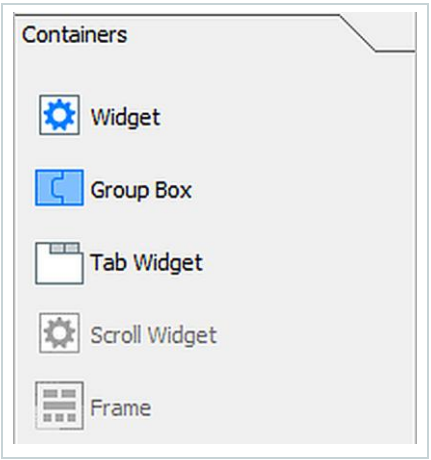


Picture 4: GUI Element Property

The designer builds the visual side of the GUI. To make widgets interact, you add the behaviour yourself in the generated Julia code (see Connecting behaviour). We focus on saving you time and effort on the layout and on simplifying the required coding.

Containers

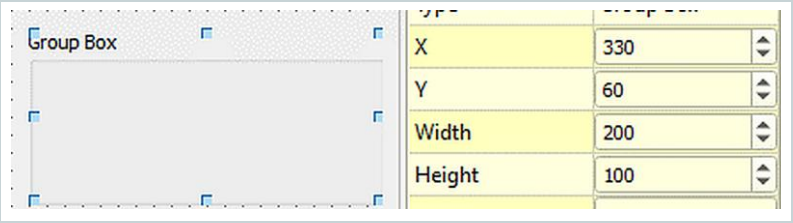
Opening the Containers menu from the top-left of the designer reveals a list of container elements. Containers are placed on the main widget area and act as carriers for other GUI elements.



Picture 5: Containers

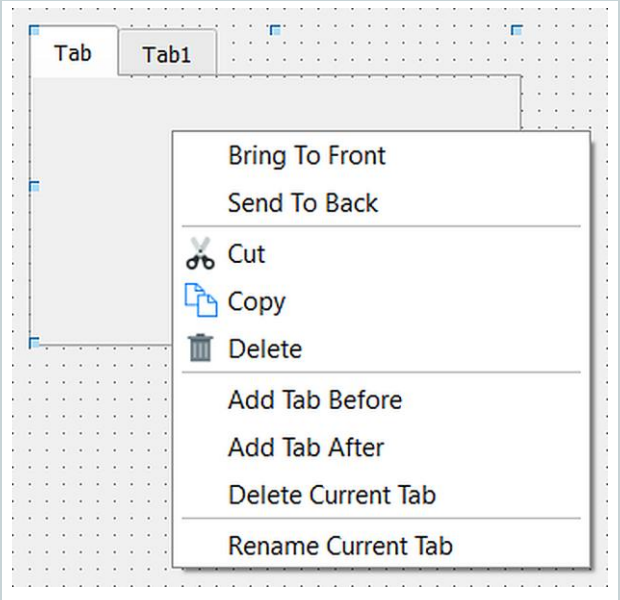
To insert a container, use the same method as any other GUI element: select the container, then click where on the working area you want to place it.

When a container or element is selected it can be resized by dragging the blue handles around it, or by changing its Width and Height values in the Property table on the right.



Picture 6: Resizing of object

When a Tab widget is used, right-click it to open the context menu, where you can rename the current tab, insert a new one, or change the stacking order of elements.



Tab widget context menu

To preview your widget and check your progress at any point, press the 'Preview Widget' button in the bottom-right corner. It opens a new interactive window where you can test your widget.

Connecting behaviour

The GUI Designer produces the layout and widget properties; it does not generate event-function stubs for the Julia (QML) target. You connect behaviour in the generated code — for example, by reacting to an Observable that backs a widget, or by handling a QML signal. In the example below, the progress bar follows the slider:

```
# The designer builds the layout; wire behaviour here.
on(hslider1) do value
    hprogressbar1[] = value
end
```

Widget settings

Selecting an element shows its settings in the Property table. Below are the settings that matter most for each element; unless noted, every widget also exposes standard layout properties (position, size, name, tool tip, colours and font).

Button, Check Box, Combo Box, Line Edit, Text Box and Label

A Button's 'Text' field sets its visible label. A Check Box toggles an option on or off — its 'Text' sets the label and its 'Value' sets whether it starts checked. A Combo Box shows a drop-down list whose rows are populated from code. A Line Edit accepts a single line of typed text and a Text Box accepts multiple lines. A Label displays static text set in its 'Text' field.

Spin Box and Double Spin Box

A Spin Box sets an integer via 'Minimum', 'Maximum', 'Step' and 'Value'. A Double Spin Box behaves the same but accepts fractional (floating-point) values.

Radio Button and List Box

A Radio Button's 'Text' sets its label and 'Value' its default state; within one container only a single radio button can be checked at a time. A List Box shows selectable rows; its 'Value' sets the default selection.

Sliders and Progress Bars

A Slider uses 'Minimum', 'Maximum', 'Step' and 'Value', with 'Orientation' switching between horizontal and vertical — both are available on the Widgets tab. A Progress Bar uses the same range and value fields, and both horizontal and vertical progress bars are available in this designer.

Image, Circular Progress, Toggle Switch and Spinner

An Image element shows a picture set with its 'Image' field. A Circular Progress indicator shows completion as a ring using 'Minimum', 'Maximum' and 'Value'. A Toggle Switch is an on/off control whose 'Value' sets its default state. A Spinner is an animated busy indicator with no value to set.

Divider, Icon Button and Tree Widget

A Divider draws a thin separating line to group parts of the layout. An Icon Button behaves like a Button but shows an icon, set with its 'Image' field. A Tree Widget shows a hierarchy of items: right-click it and choose Add Item to add a node, Add Sub Item to nest one, or Edit Item to rename it.

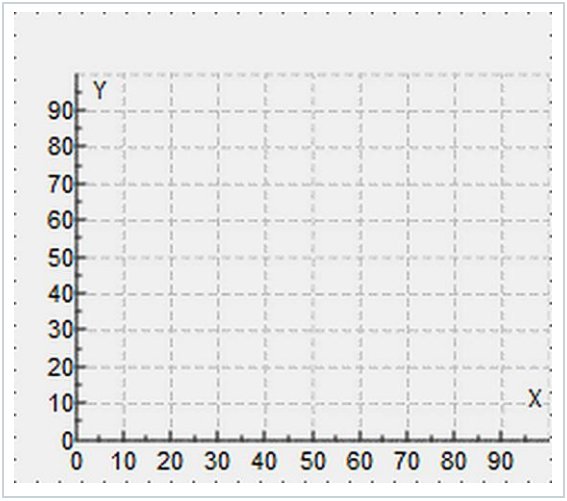
Tables and 2D/3D graphs have no property-panel settings; all of their properties are set from their context menus, and their values are assigned from code (see below).

Using tables, 2D and 3D graphs

Graphs and tables cannot be populated in the GUI Designer itself; they are customised there, but their values are assigned from the generated Julia code.

2D graphs

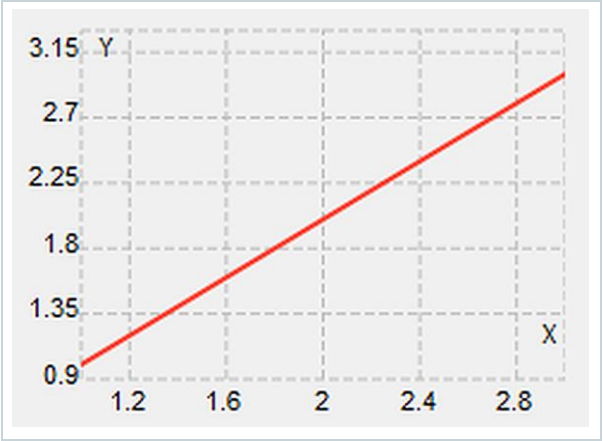
Assign a 2D graph's value with an array of x and y data:



2D graph

```
set_widget_value(graph2d1, [1 2 3; 1 2 3])
```

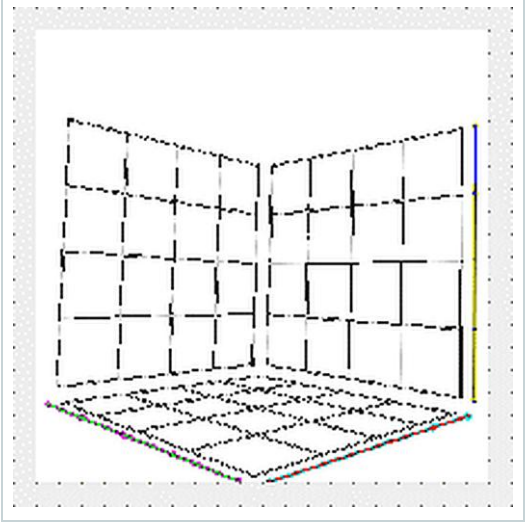
With its value assigned it looks like this:



2D graph with values

3D graphs

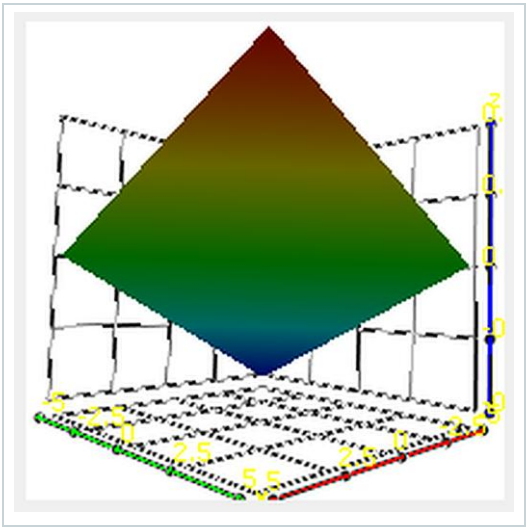
Assign a 3D graph's value using a surface definition:



3D graph

```
a = surface3d(sin(x + y), x, -5, 5, 50, y, -5, 5, 50)
set_widget_value(graph3d1, a)
```

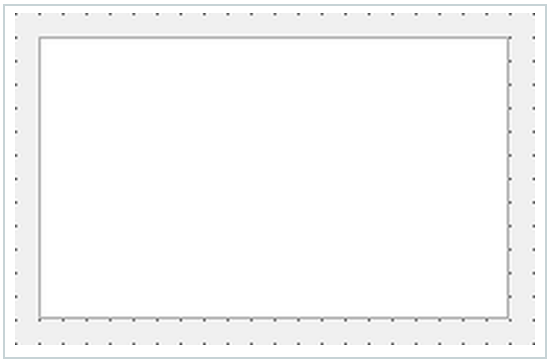
With its value assigned it looks like this:



3D graph with values

Tables

Assign a table's value with an array, just like a 2D graph. Columns are assigned automatically.



Table

```
set_widget_value(table1, [1 2 3; 1 2 3])
```

With its value assigned it looks like this:

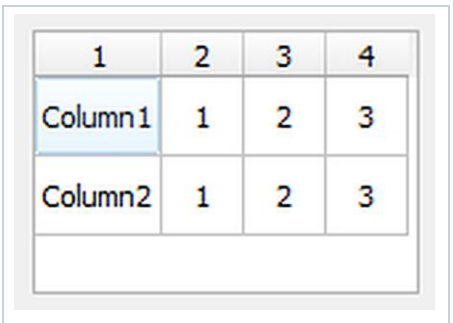
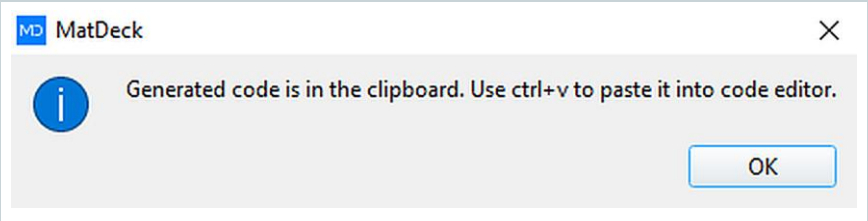


Table with values

Finishing up

When you finish your widget design, press ‘Done’. The designer generates the widget code and pastes it into the script you are working in; if you are working in a MatDeck document, the code is copied to your clipboard so you can paste it anywhere. The first time you press ‘Done’ for a new widget, the notification window (Picture 8) appears; after that, code updates silently.



Picture 8: Notification window

Using GUI Designer code in a Julia script

The generated code creates the visual side of the GUI: element positions, sizes and properties are coded exactly as designed. Values that back the widgets are exposed as Observables so you can read and update them, and the layout is loaded from the designer’s output. A representative generated file looks like this:

```
using QML, Observables

# Values backing the designer's widgets
hslider1      = Observable(0)
hprogressbar1 = Observable(0)
button1_text  = Observable("Button")

# Load the layout produced by the GUI Designer
load(joinpath(@__DIR__, "Widget1.qml"),
     hslider1 = hslider1,
     hprogressbar1 = hprogressbar1,
     button1_text = button1_text)

exec()   # show the window
```

Generated Julia code (representative)

> **Notice** The preferred workflow is to create and change widget designs in the GUI Designer, then generate or update the code from there. If you edit the code directly and afterwards open and save changes in the GUI Designer, any edits made directly in the code will be lost.

Reading and setting widget values

Each widget is backed by an Observable. Read its current value by indexing the Observable with empty brackets:

```
A = hslider1[]
```

Set a widget’s value by assigning to the Observable; any part of the GUI bound to it updates automatically:

```
hprogressbar1[] = 50
```

Combine the two inside a reaction so one widget follows another, as shown in Connecting behaviour.

Executing and outputting GUIs

Once a design is complete, the generated code is placed straight into the script. GUIs can be output in the following ways:

- Evaluation of generated code
- Build and Run

Evaluating the generated code runs the GUI application; by default it appears as a stand-alone window separate from the script it was evaluated in. Press the 'Evaluate' icon in the Programming toolbar to run the current code, or use Build and Run to package it.

The Programming toolbar