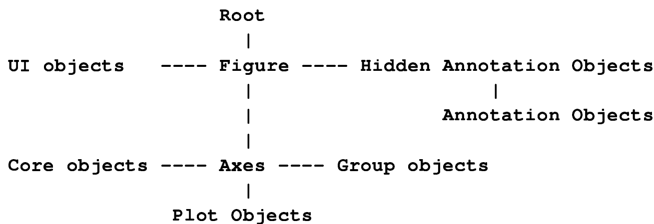
Handle graphics

Plots, windows, polygons etc. are stored in a tree structure. The windows (figure) are child-nodes to the Root (which can think of as the screen: it is created by Matlab and contains data).
The axes is a child of a figure-window and the plot data is a child of the axes etc. Each node has a set of attributes, properties that can take different values.
A "figure" has a Color-property which is the colour around the drawing area in the window. The standard value of this colour is the RGB-vector `[0.8 0.8 0.8]`. Here is the tree, there are hundreds of properties in total.



This layout is new for Matlab v7, in previous versions there were less talk about objects. A figure is a window in which the graphics is displayed. Figures contain menus, toolbars, user-interface objects (e.g. buttons and sliders), context menus (a menu bound to a curve for example), axes.
Annotation objects are things like arrows, rectangles and are usually created using the builtin plot editor.
Core objects are axes, image, light, line, patch, rectangle, surface, text.
Groups objects can be used to collectively refer to several axes, for example.
Plot Objects group together core objects. We will not look at all the objects in detail, so what follows is a simplified presentation. The manual contains more than 120 pages on the subject.

Let us look at an example. We have just started Matlab (say) and have typed the following commands:

```
>> format compact
>> x = 1:10;
>> plot(x, x.^2, 'r')
>> grid on
>> xlabel('x')
>> ylabel('y')
>> title('y = x^2')
```

The tree is linked together by handles (pointers). They are of type double and usually have many decimals. The handles of the figure windows are positive integers and the Root has handle zero. Using the function `get` we can access the value of a property for an object pointed to by the handle:
`get(handle, 'PropertyName')`.
`set(handle, 'PropertyName', value)` sets the value.

Some properties are read only. `get(handle)` prints the values of all the properties and `set(handle)` displays all property names and their possible values for the object.

Let us start to inspect the Root. I have (usually) edited the output to make it shorter. My comments after `%`.
This is about a third of what is printed.

```
>> get(0)
  CurrentFigure = [1]            % figure 1
  Diary = off
  DiaryFile = diary
  FixedWidthFontName = Courier New
  Format = short                 % set with format
  FormatSpacing = compact        % set with format
  ScreenDepth = [24]
  ScreenSize = [1 1 1280 1024]
  Units = pixels
  Children = [1]                 % The figure window
```


To see the possible alternatives for `Format`, we can do

```
>> set(0, 'Format')
[ short | long | shortE | longE | shortG | longG | hex |
  bank | + | rational | debug | shortEng | longEng ]
```

and to change format to `longE` we can

```
>> set(0, 'Format', 'longE')
>> pi
ans =
     3.141592653589793e+00
```

Usually we would instead type the shorter:

```
>> format long e
```

Property names are not case sensitive and we can shorten the name as long as it becomes unique.

```
>> set(0, 'uNiTs', 'centimeters')
>> set(0, 'units', 'centimeters')
>> set(0, 'uni', 'centimeters')
>> set(0, 'u', 'centimeters')
??? Error using ==> set
Ambiguous root property: 'u'.
```

Let us now look at the Children of the Root. We have only one child, the figure window. Since the handle is an integer we need not fetch it, but I have done so just to show how `get` works.

```
>> hf = get(0, 'Children')   % hf for handle to figure
hf =
     1
```

Here are a few of the properties:

```
>> get(hf)
  Color = [0.8 0.8 0.8]      % border colour
  Colormap = [ (64 by 3) double array]
  CurrentAxes = [153.009]    % handle to the axes
  DoubleBuffer = on          % for animation
  IntegerHandle = on         % figure 1 has handle 1
```



```
  NumberTitle = on            % Figure 1, 2 ...
  Renderer = painters         % Hidden lines removal
  Resize = on                 % Can freeze the size
  WindowButtonDownFcn =       % A Callback
  WindowButtonMotionFcn =     % Another
  WindowButtonUpFcn =         % Another
  ButtonDownFcn =             % Click over an object
  Children = [153.009]        % Same as axes
  CreateFcn =                 % More callbacks
  DeleteFcn =
  Parent = [0]                % Root
  Tag =                       % For us
  UserData = []               % For us
  Visible = on                % Can hide the window
>> set(1, 'Color', [1 0 0]) % change to red
```

Let us now look at the axes. Sometime it is inconvenient to go down in the tree this way so there are functions that gives the handles directly.

```
 gca  Get handle to current axis.
 gcf  Get handle to current figure.
 gcbo Get handle to current callback object.
 gco  Get handle to current object.
 gcbf Get handle to current callback figure.
```

So

```
>> gcf
ans = 1

>> get(get(0, 'Child'), 'Child')
ans = 1.530087890625000e+02

>> gca
ans = 1.530087890625000e+02

>> ha = get(1, 'Children')
ha = 1.530087890625000e+02  % Don't write the decimals
```

```
>> get(ha)
  AmbientLightColor = [1 1 1]
  Box = on
  CameraPosition = [5.5 50 17.3205]
  CameraUpVector = [0 1 0]
  CLim = [0 1]
  FontAngle = normal
  FontName = Helvetica
  FontSize = [10]
  FontUnits = points
  FontWeight = normal
  GridLineStyle = :
  LineWidth = [0.5]
  NextPlot = replace
  Projection = orthographic
  Position = [0.13 0.11 0.775 0.815] % llx, lly, w, h
  Title = [160.016]          % made with title command
  XLabel = [155.018]         % made with xlabel
  XTick = [ (1 by 10) double array]
  XTickLabel = 1 2 3 4 5 6 7 8 9 10
  Children = [154.088]       % the plot data
   ...
```

Let us make the grid- and axle lines wider (not the curve), use a larger font for the ticks

```
>> set(ha, 'LineWidth', 2, 'FontSize', 16, ...
         'FontWeight', 'Bold')
```

The title is hardly readable so lets make that larger as well:

```
>> set(get(ha, 'Title'))
  FontAngle: [ {normal} | italic | oblique ]
  FontName
  FontSize
  FontWeight: [ light | {normal} | demi | bold ]
  HorizontalAlignment: [ {left} | center | right ]
```



```
>> get(get(ha, 'Title'), 'String')
ans =
y = x^2

>> set(get(ha, 'Title'), 'Fontsize', 16)
```

This is not so convenient, so many commands can set the properties directly.

```
>> title('y = x^2', 'Fontsize',16, 'Fontweight','Bold')
```

We have one level left in the tree. Let us look at a leaf (terminal node), the child to the axes.

```
>> hp = get(ha, 'Children')
hp =
    1.540881347656250e+02
>> get(hp)
           Color: [1 0 0]
       LineStyle: '-'
       LineWidth: 5.000000000000000e-01
          Marker: 'none'
      MarkerSize: 6                   % useful
           XData: [1 2 3 4 5 6 7 8 9 10]
           YData: [1 4 9 16 25 36 49 64 81 100]
           ZData: [1x0 double]        % empty
   ButtonDownFcn: []
        Children: [0x1 double]        % no child
            Type: 'line'
   UIContextMenu: []
        UserData: []
         Visible: 'on'
          Parent: 1.530087890625000e+02

>> set(hp)
ans =
             LineStyle: {5x1 cell}
                Marker: {14x1 cell}
...
```



```
>> set(hp, 'LineStyle')
[ {-} | -- | : | -. | none ]  % {-} the current
>> set(hp, 'Marker')
[ + | o | * | . | x | square | diamond | v | ^ | > | < |
 pentagram | hexagram | {none} ]
```

I can change one point on the curve by typing:

```
>> y = get(hp, 'Ydata')
y =
  1  4  9 16 25 36 49 64 81 100
>> y(3) = 100;
>> set(hp, 'Ydata', y)
```

I can change the line width, but I would usually do it using the plot-command. The curve replaces the old one. The plot-function returns the handle.

```
>> hp = plot(x, x.^2, 'r', 'LineWidth', 2)
hp =
     1.540887451171875e+02

>> get(gca, 'Child')     % A new child
ans =
     1.540887451171875e+02

>> delete(hp)            % deletes the curve
>> get(gca, 'Child')
ans =
   Empty matrix: 0-by-1 % no child
```


It can be convenient to use structures:

```
>> prop.LineWidth = 3;
>> prop.Color = [1 0 0]
prop =
    LineWidth: 3
        Color: [1 0 0]

>> set(h1, prop)
>> set(h2, prop)
```

`str = get(handle);` returns a structure in `str`.
The field names are the property names and the field values are the corresponding values of the properties.

Properties have default, factory, values. We can see the 589 of them by typing `get(0, 'factory')` (only for the root). Matlab searches for a value beginning with the current object, going up in the tree until a user-defined or factory-defined value is found. We can define our own default values, which will affect objects after the change. Say that we would like to increase the font size for axes and text, say `xlabel`, in general.

```
>> diary factory             % diary filename
>> get(0, 'factory')
>> diary off
>> !grep -i font factory  % unix; edited
              factoryAxesFontAngle: 'normal'
               factoryAxesFontName: 'Helvetica'
               factoryAxesFontSize: 10
              factoryAxesFontUnits: 'points'
             factoryAxesFontWeight: 'normal'
              factoryTextFontAngle: 'normal'
               factoryTextFontName: 'Helvetica'
               factoryTextFontSize: 10
              factoryTextFontUnits: 'points'
             factoryTextFontWeight: 'normal'
```

```
>> figure(1)
% Change Factory (in the name) to Default, to set the
% default value. This works for plots in figure 1 only.
>> set(1, 'DefaultAxesFontSize', 16,  ...
          'DefaultTextFontSize', 16)

% This works for all windows,
>> set(0, 'DefaultAxesFontSize', 16, ...
          'DefaultTextFontSize', 16)
```

To keep the defaults one can save them in `~/matlab/startup.m`, which is executed when Matlab starts.
`reset(handle)` resets the values, of the object, to the factory defaults. (so `DefaultAxesFontSize` is set to 10).
To reset (remove) a specific default property, type
`set(0, 'DefaultAxesFontSize', 'remove')` for example.

Sometimes we get arrays with handles:

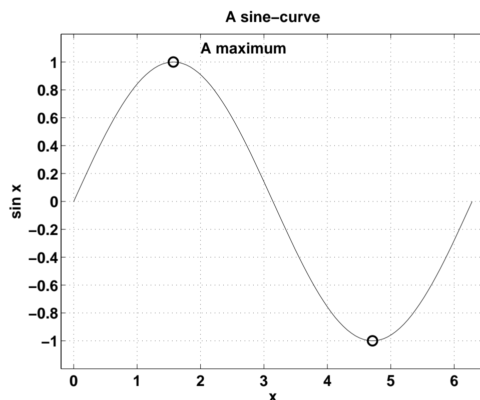
```
>> hp = plot(x, x.^2, 'k-', x, x.^2, 'ro')
hp =
   1.5400e+02
   1.5500e+02

>> get(hp, 'Type')
ans =
    'line'
    'line'

>> get(hp, 'Marker')
ans =
    'none'
    'o'

>> set(hp, 'Color', [0 1 0]) % for all the objects
```

The `fill`-command will produce patches (polygons) etc.

The above stuff is good to know when you make presentations, reports etc. Graphics does not help much if the audience cannot see it.
Here is an example. I do not claim that I have chosen the best fonts etc. An alternative is to use the builtin plot editor (see the menus and buttons in the top of the window).

```
% Say we are going to make a transparency for a lecture
figure(1)
set(1, 'DefaultAxesFontSize',      16,   ...
       'DefaultTextFontSize',      16,   ...
       'DefaultAxesFontWeight', 'Bold',  ...
       'DefaultTextFontWeight', 'Bold')

x = linspace(0, 2 * pi);
plot(x, sin(x))
hold on
grid on

% Suppose we would like to mark min and max
h = plot(0.5 * pi * [1 3], [1 -1], 'o')    % 2 handles
set(h, 'LineWidth', 2, 'MarkerSize', 10)
axis([-0.2 2*pi+0.2 -1.2 1.2])

xlabel('x')
ylabel('sin x')
title('A sine-curve')

text(2, 1.1, 'A maximum')
```

We can give the properties in the commands as well, e.g.
`text(5, -1, 'A minimum', 'Fontsize', 10)` which overrides the default.



An alternative to the above is to use the builtin plot editor (see the upper part of the window). This is convenient when you are doing an image once. I usually generate roughly the same image many times (new data, new course etc.) in which case it is more convenient to have an automatic generation in a program.

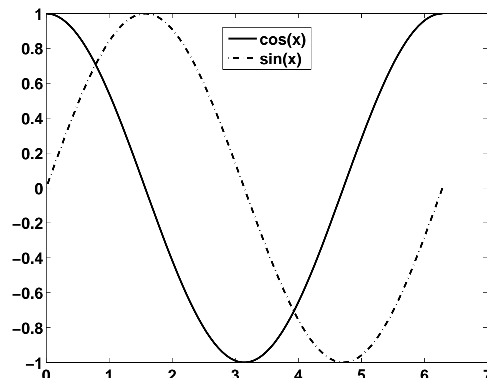
Something different: I have a command that deletes all the plot-windows (store in `~/matlab/del.m` for example)

`delete(get(0, 'Children'))`
One can use `close all` instead.

If we have several curves we should add a legend.

```
>> plot(x, cos(x), '-', x, sin(x), '-.', 'Linewidth', 2)
% Default Location is NorthEast.
% Can move the legend using the mouse as well.
>> hl = legend('cos(x)', 'sin(x)', 'Location', 'Best');

>> set(hl,  'Fontsize', 16, 'Fontweight', 'Bold')
>> set(gca, 'Fontsize', 16, 'Fontweight', 'Bold')
```

Looks like this:



Callbacks

It is common in Matlab-, OpenGL-, X11-programming to use callback routines. Such a routine is bound to a special event (e.g. the click of a mouse button) and the routine is called if the event occurs.
In this example a ButtonDownFcn-property of a curve, is used to change the colour of a curve. When we click close (5 pixels) to the curve it will change colour from blue (standard) to red. The value of the property (the callback) is, in this example, a Matlab-command. It will be executed if we click on the curve.

```
>> x = 0:0.1:2*pi;
>> h1 = plot(x, cos(x));
>> hold on
>> h2 = plot(x, sin(x));

>> get(h1)
....
        ButtonDownFcn =
        CreateFcn =
        DeleteFcn =

>> set(h1, 'ButtonDownFcn', ...
       'set(h1, ''Color'', [1 0 0])')

>> get(h1)
        ButtonDownFcn = set(h1, 'Color', [1 0 0])
```

This could be used to do the picking for the complex cosine function (in the introduction).

Note, a common misconception: the callback is not executed when we define it. It is executed if/when the action is performed.

Note also that the example shows unsafe programming, the variable `h1` may not exist when we click on the curve. Here is a better way, using the `gco`-function (get current object):

```
>> set(h1, 'ButtonDownFcn', ...
       'set(gco, ''Color'', [1 0 0])')
```

All graphics objects have three properties for which you can define callback routines:

- `ButtonDownFcn` as above.
- `CreateFcn` executes during object creation after all properties are set
- `DeleteFcn` executes just before deleting the object

User interface objects have a Callback property; more later on. Figures have the three callbacks above and (from the manual):

- `CloseRequestFcn` executes when a request is made to close the figure (by a close command, by the window manager menu, or by quitting MATLAB). Default is `closereq`.
- `KeyPressFcn` executes when users press a key while the cursor is within the figure window.
- `ResizeFcn` executes when users resize the figure window.
- `WindowButtonDownFcn` executes when users click a mouse button while the cursor is over the figure background, a disabled uicontrol, or the axes background.
- `WindowButtonMotionFcn` executes when users move the mouse within the figure window (but not over menus or title bar).
- `WindowButtonUpFcn` executes when users release the mouse button, after having pressed the mouse button within the figure.

The callback can be a Matlab command, as in the example, but also:

- a string with the name of an M-file (script or function).
- a cell array of strings (see the manual, a bit special).
- a function handle or a cell array containing a function handle and additional arguments (see the manual for the last case).

When using a function handle the callback-function must define at least two input arguments. The handle of the object generating the callback, and the event data structure (can be empty for some callbacks). Matlab passes these two arguments implicitly whenever the callback executes (it is possible to add input arguments, see the manual). Here is a simple example:

```
>> h1 = plot(x, cos(x));
>> set(h1, 'ButtonDownFcn', @my_callback)
>> type my_callback        % list a file

function my_callback(handle, event_str)
% list input arguments (only in this example)
handle
event_str

% Can skip all. Must be true for all elements.
if all(get(handle, 'Color') == [1 0 0])
  set(handle, 'Color', [0 0 1])
else
  set(handle, 'Color', [1 0 0])
end

>> handle =              % clicked on the curve
     1.540119628906250e+02
event_str =
     []
```


When we use the first alternative (a string) there are no required variables (we decide). An advantage with using function handles is that we, when making GUIs, can collect all the callbacks in one file, as in the following example. This is convenient since one tends to get many callbacks.
Here is the complex cosine-example again. We make a rectangular grid, in the complex plane, in a left subplot. Lines with constant real-parts are black, and lines with constant imaginary parts are red.
In the right subplot we plot the cosine of the points on the lines (using the same colours).
When we click on a red or black curve in either plot, the curve and the corresponding one in the other window, should become blue and twice as wide.
When we click on a blue curve in either plot, the curve and the corresponding one in the other window, should return to its original colour and get its original width.
When we click on a curve, a callback is called. In this callback we can find out the handle of the curve. The callback needs to find out the handle of the corresponding curve in the other plot. This can be solved in a number of ways.

- We can store the handles in a matrix, one row per pair of handles.
- subplot creates an axes object, so the figure has two axes-children. Each child has an array of handles to line-objects. The two handle arrays are probably ordered in the same way.
- A more general approach: use the `UserData`-property of a line to store the handle of the corresponding curve (one could store more data, e.g. a cell-array). Since the callback needs to know the original colours (can be done in several ways), I have used the `Tag`-property to store the colour as a string, `'r'` for red and `'k'` for black.

Here comes the program. The user should give intervals (real and imag) the number of lines.

```
function cos_ex(real_int, imag_int, n)
% To save space I have not included any help

figure  % New window
subplot(121); hold on % to avoid hold in the loop
subplot(122); hold on

iu = sqrt(-1);             % 50 is a bit arbitrary
im = iu * linspace(imag_int(1), imag_int(2), 50);

for re = linspace(real_int(1), real_int(2), n)
  subplot(121)
  h1 = plot([re re], imag_int, 'k');

  subplot(122)
  c = cos(re + im);
  h2 = plot(real(c), imag(c), 'k');

  set(h1,'UserData',h2, 'Tag','k', 'ButtonDownFcn',@cb)
  set(h2,'UserData',h1, 'Tag','k', 'ButtonDownFcn',@cb)
end

re = linspace(real_int(1), real_int(2), 50);
for im = linspace(imag_int(1), imag_int(2), n)
  subplot(121)
  h1 = plot(real_int, [im im], 'r');

  subplot(122)
  c = cos(re + iu * im);
  h2 = plot(real(c), imag(c), 'r');

  set(h1,'UserData',h2, 'Tag','r', 'ButtonDownFcn',@cb)
  set(h2,'UserData',h1, 'Tag','r', 'ButtonDownFcn',@cb)
end

subplot(121); axis tight
subplot(122); axis tight
```


There is a reason for:

```
c = cos(re + im);
h2 = plot(real(c), imag(c), 'k');
```

If we write like this, it may not work:

```
h2 = plot(cos(re + im), 'k');
```

Why? Consider the following:

```
>> iu = sqrt(-1);

% draws a line from (0, 0) to (0, 1) in R^2
>> plot([0; iu])
>> hold on

% a line from (1, 0) to (2, 1). Not what we want!
>> plot([0; 1])   % imag = 0

% equivalent to
>> plot([1; 0], [2; 1])

% essentially a line from (0, 0) to (1, 0)
>> plot([0; 1] + eps * iu)
```

Here comes the callback:


```
function cb(handle, event)  % note, in the same file
blue = [0 0 1];

c = get(handle, 'Color');
if all(c == blue)  % new colours, reset
% get(handle, 'Tag') is original colour 'k' or 'r'

  set(handle, 'Color', get(handle, 'Tag'), ...
              'LineWidth', 1)
  h = get(handle, 'UserData');    % other subplot
  set(h, 'Color', get(h, 'Tag'), 'LineWidth', 1)
else
% original colours, change
  set(handle, 'Color', blue, 'LineWidth', 2)
  set(get(handle, 'UserData'), 'Color', blue, ...
                                 'LineWidth', 2)
end
```

This works well in many situations. One problem is that the inverse of cos does not always exist. So there may be $z_1 \neq z_2$ with $\cos z_1 = \cos z_2$. This gives a problem with colour, clicking on z_1 may not give the same blue colour on $\cos z_1$. If $\cos z_1$ is on top of $\cos z_2$ we get a blue line, otherwise we get a mix of black and blue (or no change if we have a different line width).
A more severe problem is if we click on $\cos z_1 = \cos z_2$, only one line (not two) will become blue in the first plot. Which line reacts? Here is a short test;

```
>> v = [0 1];
>> plot(v, v, 'ButtonDownFcn', '1')  % echo 1
>> hold on
>> plot(v, v, 'ButtonDownFcn', '2')  % echo 2
>> ans =  % clicking on the line
     2
```

So the latest drawn line triggers the callback.

The following "works"; we can click on the line or on the markers.

```
>> v = linspace(0, 1, 30);
>> plot(v, v, 'ButtonDownFcn', '1')
>> hold
>> plot(v, v, 'ro', 'ButtonDownFcn', '2')
>> ans =  % clicking on a marker
     2
>> ans =  % clicking on the line
     1
```

This may be another solution in some cases:

```
>> h1 = plot(v, v, 'ButtonDownFcn', '1');
>> hold on
>> h2 = plot(v, v, 'ButtonDownFcn', '2');

>> set(h2, 'HitTest', 'Off') % cannot trigger
>> ans =                     % clicked
     1
>> set(h1, 'HitTest', 'Off') % switch this of as well
```

When both lines are "switched off" we do not get any print out (unless we have set the `ButtonDownFcn` of the current axes).

Finally an example where the event structure is not empty. Let us use the `KeyPressFcn` of a figure.

```
>> figure(1)
>> set(1, 'KeyPressFcn', @key_cb)
>> type key_cb

function key_cb(handle, event)

handle
event

>> handle = 1  % pressed the a-key with the
event =        % mouse in the window
    Character: 'a'
     Modifier: {1x0 cell}
          Key: 'a'

handle = 1     % pressed shift (part of writing A)
event =
    Character: ''
     Modifier: {1x0 cell}
          Key: 'shift'

handle = 1     % two events are generated for A
event =
    Character: 'A'
     Modifier: {'shift'}
          Key: 'a'

handle = 1     % pressed left arrow
event =
    Character: ''          % some garbage
     Modifier: {1x0 cell}  % the key sends ^[[A
          Key: 'leftarrow' % ^[ = escape
```


GUIs

It is now time to construct a more general GUI. Many things to think about when constructing a GUI, here are a few. For more references see the Diary. Some guidelines:

- No surprises! A good GUI behaves as the user expects. One should not have to hesitate when pushing a button. Nice with Undo and Cancel-alternatives.
- Consistency. Similar tasks should be done in similar ways. The user can learn principles.
- Use metaphors. A button with a magnifying glass for zooming, for example.
- Try to make the GUI self-explanatory. A user will not read manuals, perhaps not even a few lines.
- Give feedback. Did I push the button or not? Is the program running or has it crashed?
- Do not overuse strong colours, sound or movement. Keep messages readable (font, fontsize, fontweight) and clear.
- No builtin order. Modelessness. Should be able to press all buttons etc. without the program crashing. Turn off (gray out), or hide, alternatives that cannot be chosen, for example.
- Think of portability. Does the program work on another system? How does the monitor's resolution and size change the GUI? Are the sizes of buttons in pixels or cm?
- For Matlab GUIs. The users may have done other work before running your program, so be careful with using variables and windows. When your GUI quits, just clean up after your program, do not close all the windows, for example.

Matlab provides GUIDE (GUI Design Environment). You must run the GUI-mode of Matlab to use it (so do not start with `matlab -nojvm`). Then type `guide`. I will not use `guide` in this lecture.

Let us make a Quit-button. When we press the button, the window, which the button resides in, should be deleted. We make the button gray with the black text, Quit, on it. `uicontrol` is the basic tool.

```
>> figure
>> h = uicontrol;
>> set(h)
  BackgroundColor
  Callback: string -or- function handle -or- cell array
  Enable: [ {on} | off | inactive ]
  FontName
  FontSize
  ForegroundColor
  HorizontalAlignment: [ left | {center} | right ]
  KeyPressFcn: string -or- function handle -or- cell arra
  Max
  Min
  Position
  String
  Style: [ {pushbutton} | togglebutton | radiobutton |
           checkbox | edit | text | slider | frame |
           listbox | popupmenu ]
  TooltipString
  Units: [ inches | centimeters | normalized | points |
         {pixels} | characters ]
  Value
...
  Visible: [ {on} | off ]
```


We can choose between the following types:

- pushbutton, button with no memory
- togglebutton, on-off-button
- radiobutton, to choose the station on a radio (mutually exclusive)
- checkbox, tick choices
- edit, text that can be edited
- text, above a button. for example
- slider
- frame, rectangles that provide a visual enclosure for regions of a figure window (obsolete)
- listbox, scrollable list with alternatives
- popupmenu (does not work with `-nojvm`)

Some of the buttons only differ in appearance; we have to fix the functionality. A suitable button in our example is a pushbutton, which is the default. In this example we could use a string instead of a function.

```
>> type Quit_ex

function Quit_ex

hf = figure;
set(hf, 'Name',         'My GUI',        ...
        'NumberTitle',  'Off',           ...
        'MenuBar',      'None',          ...
        'Units',        'centimeters',   ...
        'Position',     [10, 10, 5, 3])
```

```
hb = uicontrol( ...
     'Style',           'pushbutton',   ...  % default
     'Units',           'centimeters',  ...
     'Position',        [0.5 0.5 2 1],  ...
     'String',          'Quit',         ...
     'TooltipString',   'Close this window', ...
     'BackgroundColor', [0.7 0.7 0.7],  ...
     'ForegroundColor', [0 0 0],        ...
     'Callback',        @Quit_cb );

function Quit_cb(handle, event)

% gcbf: Get handle to current callback figure.
%  fig = gcbf returns the handle of the figure
%        that contains the object whose callback
%        is currently executing.

delete(gcbf)
```

Position is lower left x, lower left y, width, height.

Here comes a toggle button. The string, on the button, should alternate between On and Off. The button has a Value-property. Matlab will automatically alternate the value of Value between 0 and 1.

```
>> type Toggle_ex
function Toggle_ex

hf = figure;
set(hf, 'Name',          'My GUI',       ...
        'NumberTitle',   'Off',          ...
        'MenuBar',       'None',         ...
        'Units',         'centimeters', ...
        'Position',      [10, 10, 5, 3])

% Toggle buttons set Value to Max (default 1) when
% they are down (selected) and Min (default 0)
% when up (not selected).
hb = uicontrol( ...
     'Style',           'togglebutton', ...
     'Units',           'centimeters',  ...
     'Position',        [0.5 0.5 2 1],  ...
     'String',          'Off',          ...
     'BackgroundColor', [0.7 0.7 0.7],  ...
     'ForegroundColor', [0 0 0],        ...
     'Value',           0,              ... % Initially
     'Callback',        @Toggle_cb );       % Off

function Toggle_cb(handle, event)

% If Value = 1 when we clicked, then Value = 0
% in this callback.
if get(handle, 'Value')
  set(handle, 'String', 'On')  % used to be Off
else
  set(handle, 'String', 'Off') % used to be On
end
```


A shorter version:

```
function Toggle_cb(handle, event)
str = {'Off', 'On'};
set(handle, 'String', str{1 + get(handle, 'Value')})
```

Note that `str = ['Off', 'On'];` gives one string, `'OffOn'`.

Here comes a slider, where we can set values continuously. We should put a text close to each slider. In the example we use the same callback. This is not necessary, nor is he use of the Tag.

```
>> type Slider_ex

function Slider_ex

hf = figure;
set(hf, 'Name', 'My GUI', 'NumberTitle', 'Off',    ...
'MenuBar', 'None', 'Units', 'centimeters',         ...
'Position', [10, 10, 4, 4],                        ...
'DefaultUicontrolUnits',           'centimeters', ...
'DefaultUicontrolBackgroundColor', [0.7 0.7 0.7], ...
'DefaultUicontrolForegroundColor',       [0 0 0])

uicontrol('Style', 'slider',         ...
     'Position',    [0.5 0.5 3 0.7], ...
     'Min', -1,    ...  % min value of slider
     'Max', 2,     ...  % max value
     'Value', 1,  ...  % initial value
     'Tag', 'slider_1',    ...
     'Callback', @Slider_cb );

uicontrol('Style', 'slider',            ...
     'Position', [0.5 1.5 3 0.9],       ...
     'Min', -1, 'Max', 2, 'Value', 1,   ...
     'Tag', 'slider_2',                 ...
     'Callback', @Slider_cb );
```

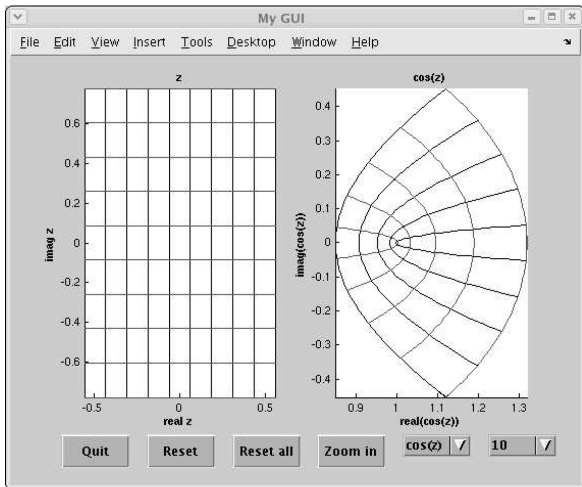


```
% Help text. May want a different BG-colour
uicontrol('Style', 'text', 'String', 'Two sliders', ...
          'FontWeight', 'Bold',...
          'Position', [0.5 2.4 3 0.5])

function Slider_cb(handle, event)
% Can have different callbacks for different
% sliders of course. Does not do anything
% useful.

val = get(handle, 'Value')
if get(handle, 'Tag') == 'slider_1'
  disp('slider_1')
else
  disp('slider_2')
end
```

This is using Matlab with Java. Turning off Java, `-nojvm`, gives a different appearance.

Notice also the area for the text (slightly darker).

Here comes a more sophisticated example. We take the old cosine-example (where we can click on the curves) and add some buttons and menus. We start the program by typing **cos_ex_gui** and get the following window:



Quit should delete the window. Reset should reset all the lines to their original colours and width. Using the left popupmenu we can choose between four functions; the plot is updated.

The next menu sets the number of grid lines; the plot is updated. Zoom in allows us the click twice in the left window to mark a smaller rectangle; the plot is updated. Reset all, resets everything (like starting over).

There should be texts above the menus.

Here is the code (> 240 lines). I have had to compress it (compared to my original). All routines in one file.

```
function cos_ex_gui
% Should have better names for the global variables
% or not use global. Can use UserData of the figure.
global ha1 ha2 hm_fun hm_n fun funcs ...
       real_int imag_int n

% default values
real_int = [-1 1]; % real interval
imag_int = [-1 1]; % imag interval
n        = 10;     % # of grid lines
fun      = 1;      % choice of function

funcs = {@(z)cos(z), @(z)sin(z), @(z)exp(z), @(z)z.^2};

make_gui    % create buttons etc.
make_plots  % draws the grid and function(grid)

% ------------------- make_gui -------------------
function make_gui
global ha1 ha2 hm_fun hm_n funcs

hf = figure;

set(hf, 'Name', 'My GUI', 'NumberTitle', 'Off', ...
         'Units', 'centimeters',                    ...
        'DefaultAxesUnits',      'centimeters', ...
        'DefaultUicontrolUnits', 'centimeters', ...
        'DefaultUicontrolFontWeight', 'Bold',   ...
        'DefaultUicontrolBackgroundColor', ...
          [0.7 0.7 0.7],                 ...
        'DefaultUicontrolForegroundColor', 'k')

ha1 = subplot(121); hold on
ha2 = subplot(122); hold on
```



```
% shrink subplots
dp = 1.2 * [0 1 0 -1];
set(ha1, 'Position', get(ha1, 'Position') + dp)
set(ha2, 'Position', get(ha2, 'Position') + dp)

% create buttons and menus
pos = [1.5 0.5 2 1]; dx = 0.5;

% Quit-button
uicontrol('Position', pos,            ...
'String',          'Quit',            ...
'TooltipString',   'close window',    ...
'Callback',        'delete(gcbf)' );  % string

% Reset-button
pos(1) = pos(1) + pos(3) + dx;
uicontrol('Position', pos,            ...
'String',          'Reset',           ...
'TooltipString',   'reset lines',     ...
'Callback',        @reset_cb);

% Reset all-button
pos(1) = pos(1) + pos(3) + dx;
uicontrol('Position', pos,              ...
'String',          'Reset all',         ...
'TooltipString',   'reset everything', ...
'Callback',        @reset_all_cb);

% Zoom-button
pos(1) = pos(1) + pos(3) + dx;
uicontrol('Position', pos,               ...
'String',          'Zoom in',            ...
'TooltipString',   'zoom in left plot', ...
'Callback',        @zoom_cb);
```



```
% Build menu-items
for fun = 1:length(funcs)
  t = char(funcs{fun});     % @(z)expression
  t = t(t ~= '.');          % rm elementwise
  items{fun} = t(5:end);
end

% Function menu
pos(1) = pos(1) + pos(3) + dx;
hm_fun = uicontrol('Style', 'popupmenu',   ...
'Position',        pos,          ...
'TooltipString',   'function',   ...
'String',          items,        ...
'Value',           1,            ...  % default
'Callback',        @menu_fun_cb);

% An alternative to cell arrays
% n-menu (number of grid lines)
pos(1) = pos(1) + pos(3) + dx;
hm_n = uicontrol('Style', 'popupmenu', ...
'Position',        pos,               ...
'String',                             ...
    '5|6|7|8|9|10|11|12|13|14|15|16|17|18|19|20', ...
'TooltipString',   '# of lines',     ...
'Value',           6,                ...
'Callback',        @menu_n_cb);

% ------------------- equal -------------------
function eq = equal(s1, s2)
% Compare two strings. used by reset_cb.
% May be of unequal length (strcmp) and different
% case (i in strcmpi). Blanks are significant
% for strcmpi, so they are removed.

eq = strcmpi(s1(s1 ~= ' '), s2(s2 ~= ' '));
```

```
% ------------------- reset_cb -------------------
function reset_cb(handle, event)
% Could call make_plots instead
% but this shows a different technique

h  = get(handle, ’Parent’);  % i.e. the figure
hc = get(h, ’Children’);     % axes and uicontrol

for h = hc(:)’          % for all axes and uicontrols
  if equal(get(h, ’Type’), ’axes’)
    hl = get(h, ’Children’);  % lines

    for hline = hl(:)’  % for all lines
      set(hline, ’Linewidth’, 1, ...
                 ’Color’, get(hline, ’Tag’))
    end
  end
end

% ------------------- reset_all_cb -------------------
function reset_all_cb(handle, event)
global hm_fun hm_n fun real_int imag_int n

real_int = [-1 1]; % default values
imag_int = [-1 1];
fun      = 1;
n        = 10;

set(hm_fun, ’Value’, fun) % reset menus
set(hm_n,   ’Value’, 6)

make_plots                % redraw
```



```
% ------------------- zoom_cb -------------------
function zoom_cb(handle, event)
%
% Can zoom in (but not out)
% There is builtin support for zoom (help zoom).
%
global real_int imag_int

[re, im] = ginput(2); % no conflict with
real_int = sort(re);  % clicking on lines
imag_int = sort(im);  % should check the values

make_plots            % redraw

% ------------------- menu_fun_cb -------------------
function menu_fun_cb(handle, event)
global fun

fun = get(handle, ’Value’);
make_plots          % redraw

% ------------------- menu_n_cb -------------------
function menu_n_cb(handle, event)
global n

n = 4 + get(handle, ’Value’);
make_plots          % redraw

% ------------------- make_plots -------------------
function make_plots
% almost like the old version
global ha1 ha2 fun funcs real_int imag_int n
iu = sqrt(-1);
```



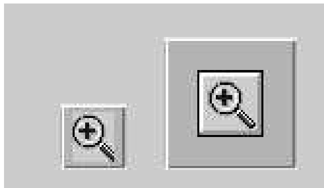
```
% Remove curves. OK if empty. This is new.
delete(get(ha1, ’Children’))
delete(get(ha2, ’Children’))

im = iu * linspace(imag_int(1), imag_int(2), 50);
for re = linspace(real_int(1), real_int(2), n)
  subplot(ha1)   % subplot(121) changes position. New.
  h1 = plot([re re], imag_int, ’k’);

  subplot(ha2)
  c = funcs{fun}(re + im);   % This is ew
  h2 = plot(real(c), imag(c), ’k’);

  set(h1, ’UserData’, h2, ’Tag’, ’k’, ...
          ’ButtonDownFcn’, @plot_cb)
  set(h2, ’UserData’, h1, ’Tag’, ’k’, ...
          ’ButtonDownFcn’, @plot_cb)
end

re = linspace(real_int(1), real_int(2), 50);
for im = linspace(imag_int(1), imag_int(2), n)
  subplot(ha1)
  h1 = plot(real_int, [im im], ’r’);

  subplot(ha2)
  c = funcs{fun}(re + iu * im);
  h2 = plot(real(c), imag(c), ’r’);

  set(h1, ’UserData’, h2, ’Tag’, ’r’, ...
          ’ButtonDownFcn’, @plot_cb)
  set(h2, ’UserData’, h1, ’Tag’, ’r’, ...
          ’ButtonDownFcn’, @plot_cb)
end
```



```
% This is new
subplot(ha1); axis tight
h(1) = xlabel(’real z’);
h(2) = ylabel(’imag z’);
h(3) = title(’z’);

subplot(ha2); axis tight
t = char(funcs{fun});   % something like @(z)expression
t = t(5:end);           % rm @(z)
t = t(t ~= ’.’);        % rm dots

% xlabel should be real(cos(z)) etc.
h(4) = xlabel([’real(’, t, ’)’]);
h(5) = ylabel([’imag(’, t, ’)’]);
h(6) = title(t);
set(h, ’FontWeight’, ’Bold’)

% ------------------- plot_cb  -------------------
function plot_cb(handle, event)
% ... same as function cb in the previous example
blue = [0 0 1];

c = get(handle, ’Color’);
if all(c == blue)  % new colours, reset
% get(handle, ’Tag’) is original colour ’k’ or ’r’

  set(handle, ’Color’, get(handle, ’Tag’), ’LineWidth’, 1
  h = get(handle, ’UserData’);    % other subplot
  set(h, ’Color’, get(h, ’Tag’), ’LineWidth’, 1)
else
% original colours, change
  set(handle, ’Color’, blue, ’LineWidth’, 2)
  set(get(handle, ’UserData’), ’Color’, blue, ...
      ’LineWidth’, 2)
end
```

It is possible to have textures on buttons. I fetched a gif-image of a magnifying glass. Matlab requires true colour (24-bit colour) and I used the **xv**-command to convert the image and saved it as a jpeg-image (highest quality). The original image has a black border.

```
>> C = imread('mag.jpg', 'jpg');  % read the file
>> image(C)                       % look at it
>> axis image                     % correct scaling
>> size(C)
ans =
    32    32     3                % a 3D-matrix

>> figure
>> uicontrol('Style',    'Pushbutton',      ...
             'Units',    'pixels',          ...  % Note
             'Position', [100 100 32 32], ...
             'CData',    C,                 ...  % Note
             'Callback', @Zoom_cb );

>> uicontrol('Style',    'Pushbutton',      ...
             'Units',    'pixels',          ...
             'Position', [150 100 64 64], ...
             'CData',    C,                 ...
             'Callback', @Zoom_cb );
```



We can add menus at the top of the window as well.

```
h  = figure;
hm = uimenu(h,  'Label', 'My menu');
% set(h, 'MenuBar', 'None') removes the standard menu

% Accelerator: type CTRL-K with the mouse in the window
alt(1) = uimenu(hm, 'Label',       'Beef',              ...
                    'Callback',    'disp(''Beef'')', ...
                    'Accelerator', 'K');

% Can have callback here as well
alt(2) = uimenu(hm, 'Label', 'Chicken');

alt(3) = uimenu(hm, 'Label',    'Fish',           ...
                    'Callback', 'disp(''Fish'')' );

% We can do hierarchical menus. Don't overuse!

uimenu(alt(2), 'Label', 'with Cashew nuts', ...
               'Callback', 'disp(''Cashew'')');

uimenu(alt(2), 'Label', 'in Curry', ...
               'Callback', 'disp(''Curry'')');

uimenu(alt(2), 'Label', 'with Peppers', ...
               'Callback', 'disp(''Peppers'')');
```

A few more words about clicking on curves.

If you choose "Data Cursor"-tool (to the right of the rotate button) you can click on an object (also in 3D) to get the coordinates.

You can change the cursor to one of several predefined:

```
>> set(gcf, 'Pointer', 'arrow')
% or 'watch' etc. See the manual.
```

The watch-cursor is an animation under Gnome.

You can make your own cursor, as well. Create a 16×16-matrix containing 1 (black), 2 (white) and NaN (transparent).
Let us make a large X.

```
>> C = eye(16); C = C + C(:, end:-1:1);
>> C = C ./ C;
Warning: Divide by zero.

>> C(6:11, 6:11)
ans =
     1   NaN   NaN   NaN   NaN     1
   NaN     1   NaN   NaN     1   NaN
   NaN   NaN     1     1   NaN   NaN
   NaN   NaN     1     1   NaN   NaN
   NaN     1   NaN   NaN     1   NaN
     1   NaN   NaN   NaN   NaN     1

>> figure(1)
>> set(1, 'Pointer',               'Custom', ...
          'PointerShapeCData',      C,       ...
          'PointerShapeHotSpot',   [8.5 8.5])
```

PointerShapeHotSpot is the pointer location.

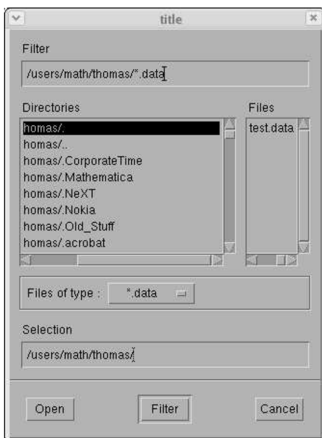
We can bind a menu (context menu) to a graphical object, e.g. a curve.

```
figure(1)

% Create a context menu
cmenu = uicontextmenu;

x = 0:0.1:1;

% and bind it to the curve
hp = plot(x, sin(x), 'UIContextMenu', cmenu);

% Define callbacks...
cb1 = 'set(hp, ''LineStyle'', ''--'')';
cb2 = 'set(hp, ''LineStyle'', '':'')';
cb3 = 'set(hp, ''LineStyle'', ''-'')';

% Define the menu alternatives
uimenu(cmenu, 'Label', 'dashed', 'Callback', cb1)
uimenu(cmenu, 'Label', 'dotted', 'Callback', cb2)
uimenu(cmenu, 'Label', 'solid',  'Callback', cb3)
```

If one RIGHT-clicks on the curve, a menu appears where we can choose between dashed, dotted and solid.

Loading files

```
> type load_file.m
function load_file
uicontrol('Style','PushButton', 'Units','centimeters',
          'Position', [1 3 2 1.5], 'String', 'Load',
          'Callback',    @load_cb )

function load_cb(handle, event)
pos    = [100 100];  % [from_left, from_top], in pixels
filter = '*.data';

[file_name, path_to_file] = ...
  uigetfile(filter, 'title', 'Location', pos);

file_name, path_to_file   % We usually don't print

>> load_file
file_name = test.data
path_to_file = /users/math/thomas/
```



Error messages

```
function error_msg(msg)

% Inactivate all other windows using a modal
% dialogue
figure( 'Units',       'centimeters',  ...
        'Position',     [15 15 4 2],   ...
        'Color',        [1.0 0.5 0.5], ...
        'MenuBar',     'None',         ...
        'NumberTitle', 'Off',          ...
        'WindowStyle', 'Modal',        ...  % Note
        'Name',        'Error' );
axis('off')

% the error message
text(0, 0.7, msg, 'FontWeight', 'Bold')

% Remove the window when we have pressed OK

uicontrol( 'Style',       'PushButton',    ...
           'Units',       'centimeters',   ...
           'Position',     [0.3, 0.3, 1, 0.7], ...
           'String',      'OK',            ...
           'Callback',    'delete(gcbf)')
```

`error_msg('Nothing to plot')` gives:

There is a function for this in Matlab:
`errordlg('message', 'title', 'modal').`
This is one of several such functions. See help or the manual.

Predefined Dialog Boxes

- dialog Create and display dialog box
- errordlg Create and display error dialog box
- helpdlg Create and display help dialog box
- inputdlg Create and display input dialog box
- listdlg Create and display list selection dialog box
- msgbox Create and display message dialog box
- pagesetupdlg Display page setup dialog box
- printdlg Display print dialog box
- questdlg Display question dialog box
- uigetdir Display standard dialog box for retrieving a directory
- uigetfile Display standard dialog box for retrieving files
- uigetpref Display dialog box for retrieving preferences
- uiputfile Display standard dialog box for saving files
- uisave Display standard dialog box for saving workspace variables
- uisetcolor Display standard dialog box for setting an object's ColorSpec
- uisetfont Display standard dialog box for setting an object's font characteristics
- waitbar Display waitbar
- warndlg Display warning dialog box

Animation in Matlab

Example: we would like to animate a square that bounces inside a rectangle. We assume that the square always hits a wall at a 45 degree angle and that no energy is lost in the contact.
Here is a simple solution:

```
function test5
global min_x max_x min_y max_y v cont

% initial position for square
x = 3 * [0 1 1 0]' + 15;
y = 3 * [0 0 1 1]' + 15;

hf = figure;
set(hf, 'DeleteFcn', @clean_up)

% plot square
h = fill(x, y, 'r');
axis equal
min_x = 0; max_x = 90; min_y = 0; max_y = 31;

% boundingbox
axis([min_x max_x min_y max_y])
set(gca, 'xtick', [], 'ytick', [])

v    = [1 1];    % initial direction
cont = 1;

while cont
% drawnow        % update screen
  pause(0.01)    % pause and update screen
  update_pos(h)
end
```

`drawnow` or `pause` is needed to flush the queue for graphics events, otherwise all the events will accumulate and nothing is plotted.

```
function update_pos(h)
global min_x max_x min_y max_y v cont

% necessary since this routine can be called when
% we have deleted the window
if ~cont, return, end

% fetch position
x = get(h, 'xdata');
y = get(h, 'ydata');

% check if squrea has hit a wall or a corner
off_y = y(3) >= max_y || y(1) <= min_y;
if x(2) >= max_x || x(1) <= min_x
  if off_y
    v = -v;
    set(h, 'Facecolor', 'g')  % change colour as well
  else
    v = [-v(1), v(2)];
    set(h, 'Facecolor', 'b')
  end
elseif off_y
  v = [v(1), -v(2)];
    set(h, 'Facecolor', 'y')
end

% update position
x = x + 0.2 * v(1);
y = y + 0.2 * v(2);

% update graphics data
set(h, 'xdata', x, 'ydata', y)
```



```
function clean_up(obj, event)
% called when we delete the window
global cont

cont = 0;
```

Another way to update the image is to use a so-called timer object. A timer object is similar to a clock that runs in parallel with ones program (a separate thread).
The clock can be set up so that it calls a callback routine at times $t_0, t_0 + \delta_t, t_0 + 2\delta_t, t_0 + 3\delta_t, \ldots$. t_0 is called start delay and δ_t period. Java must be enabled for this to work.
First some simple examples.

```
> t = timer('TimerFcn', 'disp(''tic'')',      ...
            'ExecutionMode', 'fixedSpacing', ...
            'Period', 1, 'TasksToExecute', 5)

   Timer Object: timer-1

   Timer Settings
      ExecutionMode: fixedSpacing
             Period: 1
           BusyMode: drop
            Running: off

   Callbacks
           TimerFcn: 'disp('tic')'
           ErrorFcn: ''
           StartFcn: ''
            StopFcn: ''

>> start(t)
tic
tic
tic
tic
tic
```



```
>> get(t, 'Running')
ans = off
>> delete(t)

% make a new timer
>> t = timer('TimerFcn', 'disp(''tic'')',      ...
             'ExecutionMode', 'fixedSpacing', ...
             'Period', 10, 'TasksToExecute', 5);
>> start(t)
tic
>> get(t, 'Running')
ans = on
>> stop(t)
>> delete(t)

% make a new timer
>> t = timer('TimerFcn', 'disp(''tic'')',      ...
             'ExecutionMode', 'fixedSpacing', ...
             'Period', 10, 'TasksToExecute', 5);
>> start(t)
tic
tic
>> delete(t)
Warning: One or more timer objects were stopped
         before deletion.

% make two new timers
>> t1 = timer('TimerFcn', 'disp(''tic_1'')',    ...
              'ExecutionMode', 'fixedSpacing', ...
              'Period', 1, 'TasksToExecute', 5);

>> t2 = timer('TimerFcn', 'disp(''tic_2'')',    ...
              'ExecutionMode', 'fixedSpacing', ...
              'Period', 1, 'TasksToExecute', 5);
```



```
>> start(t1)
tic_1
tic_1
>> start(t2)
tic_2
tic_1
tic_2
tic_1
tic_2
tic_1
tic_2
tic_2
>> timerfind

Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:
   1       fixedSpacing    1        'disp('tic_1')'
   2       fixedSpacing    1        'disp('tic_2')'

>> delete(timerfind)
>> timerfind
ans =
     []


% make a new timer
>> t = timer('TimerFcn', 'disp(''tic'')',      ...
             'ExecutionMode', 'fixedSpacing', ...
             'Period', 1, 'TasksToExecute', 5);
>> start(t); wait(t)    % block the command line
```

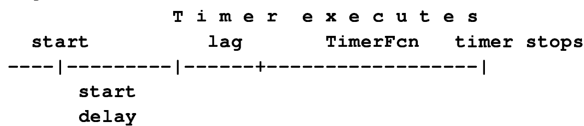
Possible to have `'TasksToExecute', Inf`

Here are some of the most important properties of a timer object.

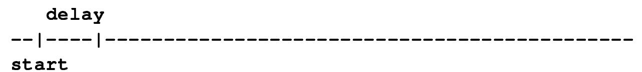
- `BusyMode` Action taken when a timer has to execute `TimerFcn` before the completion of previous execution of `TimerFcn`.
  `'drop'`, do not execute the function. (default).
  `'error'`, generate an error.
  `'queue'`, execute function at next opportunity.
- `ExecutionMode` Determines how the timer object schedules timer events. `'singleShot'` (default), `'fixedDelay'`, `'fixedRate'`, `'fixedSpacing'`.
- `Period` Specifies the delay, in seconds, between executions of `TimerFcn`.
- `Running` Indicates whether the timer is currently executing.
- `StartDelay` Specifies the delay, in seconds, between the start of the timer and the first execution of the function specified in `TimerFcn`.
- `StartFcn` Function the timer calls when it starts.
- `StopFcn` Function the timer calls when it stops.
- `TasksToExecute` Specifies the number of times the timer should execute the function specified in the `TimerFcn` property.
- `TimerFcn` Timer callback function.
- `UserData` User-supplied data.

More deails about `ExecutionMode`. The duration of the lag depends on what other processing Matlab happens to be doing at the time.

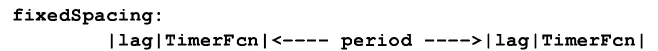
`singleShot`



Here are the other three cases:

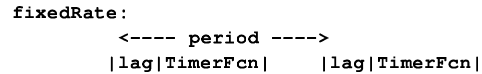


`fixedSpacing:`



`fixedDelay:`

`fixedRate:`



Here is a code for the bouncing squre using timer objects.

```
function test55
global min_x max_x min_y max_y

hf = figure;
% Can stop the square by clicking in the window
% outside the plot area (stop_go).
% When we close the window clean_up is executed.

set(hf, 'ButtonDownFcn', @stop_go, ...
        'DeleteFcn', @clean_up)
hold off

x = 3 * [0 1 1 0]' + 15;  % same as before
y = 3 * [0 0 1 1]' + 15;
h = fill(x, y, 'r');
axis equal
min_x = 0; max_x = 30; min_y = 0; max_y = 51;
axis([min_x max_x min_y max_y])
set(gca, 'xtick', [], 'ytick', [])
```



```
% Create timer and define properties
t = timer;
set(t, 'TimerFcn', @my_update, 'StartDelay', 0,  ...
       'TasksToExecute', Inf,  'Period', 0.015,  ...
       'ExecutionMode', 'fixedSpacing',          ...
       'BusyMode', 'drop');

v = [1 1];
set(t, 'UserData', {h, v})  % store h and v in Userdata

set(hf, 'UserData', t) % store handle to timer in figure
start(t)
% -------------------------------------------------------
function my_update(obj, event)
global min_x max_x min_y max_y

ud = get(obj, 'UserData');  % obj = timer
h = ud{1};
if ~ishandle(h)  % just to be sure...
  disp('no handle')
  stop(t)
  delete(t)
  return
end

v = ud{2};
x = get(h, 'xdata');
y = get(h, 'ydata');

% same as before
off_y = y(3) >= max_y || y(1) <= min_y;
if x(2) >= max_x || x(1) <= min_x
  if off_y
    v = -v;
    set(h, 'Facecolor', 'g')
  else
    v = [-v(1), v(2)];
```



```
    set(h, 'Facecolor', 'b')
  end
elseif off_y
  v = [v(1), -v(2)];
    set(h, 'Facecolor', 'y')
end

x = x + 0.2 * v(1);
y = y + 0.2 * v(2);

set(h, 'xdata', x, 'ydata', y)

set(obj, 'UserData', {h, v})

% ---------------------------------------------

function clean_up(obj, event)
disp('clean_up')
t = get(obj, 'UserData');  % obj = figure
run = get(t, 'Running');
if run(1:2) == 'on' % other is off
  stop(t)
end
delete(t)

% ---------------------------------------------

function stop_go(obj, event)
t = get(obj, 'UserData');  % obj = figure
run = get(t, 'Running');
if run(1:2) == 'on' % other is off
  stop(t)
else
  start(t)
end
```

It does happen that the timer continues to run even though we have removed the window (I do not know why). Typing `^C` in the Matlab command window seems to solve the problem.

In some versions of Matlab it may be useful to switch on double buffering (on our system it is switched on). This makes for a more steady, flicker free, animation.

In this method, two graphics pages in the video memory are used. While one page is displayed by the monitor, the other is drawn. When drawing is complete, the roles of the two pages are switched, so that the previously shown page is modified, and the previously drawn page is shown.

```
>> figure(1)
>> set(1, 'DoubleBuffer')
[ {on} | off ]
```

Another property that is important is `Renderer`. It can take one of four values, only the first are of interest to us:

```
>> set(1, 'Renderer')
[ {painters} | zbuffer | OpenGL | None ]
```

The meaning of the different values will be explained later in the course. `painters` is a fast method for drawing simple graphics having no light sources. `zbuffer` and `OpenGL` are used for more complicated scenes and `OpenGL` is also the choice when we would like to use the system's graphics hardware. Matlab switches automatically (provided `RenderMode` is set to `auto`), for example:

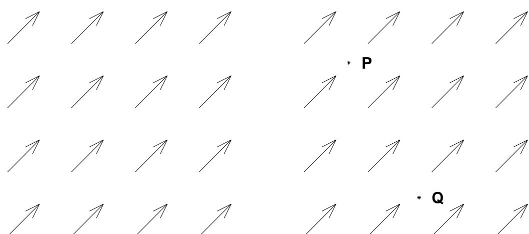
```
>> figure(1)
>> get(1, 'Renderer')
ans = None

>> plot(rand(10, 1))
>> get(1, 'Renderer')
ans = painters
```



```
>> surf(rand(10))
>> get(1, 'Renderer')
ans = painters

>> shading interp
>> get(1, 'Renderer')
ans = OpenGL

>> opengl info

Version          = 2.0.2 NVIDIA 87.62
Vendor           = NVIDIA Corporation
Renderer         = GeForce 6600/PCI/SSE2/3DNOW!
MaxTextureSize   = 4096
Visual           = 0x26 (TrueColor, depth 24, RGB mask 0x
Software         = false
# of Extensions = 120

Driver Bug Workarounds:
OpenGLBitmapZbufferBug     = 0
OpenGLWobbleTesselatorBug = 0
OpenGLLineSmoothingBug     = 0
OpenGLClippedImageBug      = 1
OpenGLEraseModeBug         = 0

>> opengl software
>> opengl info

Version          = 1.5 Mesa 6.0.1
Vendor           = Brian Paul
Renderer         = Mesa X11
MaxTextureSize   = 2048
Visual           = 0x21 (TrueColor, depth 24, RGB mask 0x
Software         = true
# of Extensions = 96

Driver Bug Workarounds:  etc.
```

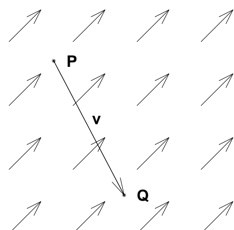

Vectors and points

Important to distinguish between point and vectors in computer graphics, so here comes a short review. A vector is an equivalence class (think set) of directed line segments that share the same length and direction. One of the segments is a representative of the vector.
The left image shows some representatives of the vector.



There are infinitely many representatives. A point, however, is a unique object. `P` and `Q` are two points (right image).

Two points define a vector: $v = Q - P$ is the vector which starts in P and goes to Q. A point and a vector defines a new point: $Q = P + v$. A single point, P (or Q) does not define a vector. A vector does not define a point, either.



A basis is a set of linearly independent vectors such that all vectors (in the space) can be written as a linear combination of the basis vectors. A vector has a coordinate representation in such a system. The left image shows a basis. It is common to draw the representatives starting at the same point.
Note, however, that we still do not have an origin.

Let us now forget the basis for a while, and instead introduce a special, fix point, the origin, $\mathcal{O}$.

Given the origin we can get a 1-1 correspondence between vectors and points by using the representative starting in $\mathcal{O}$ and ending in the P (Sw. ortsvektor). In the right image v corresponds to the point P.

A coordinate system is an origin together with a basis. A point and a vector has a coordinate representation in such a system.
We will use ON-systems (orthogonal and normalized basis).