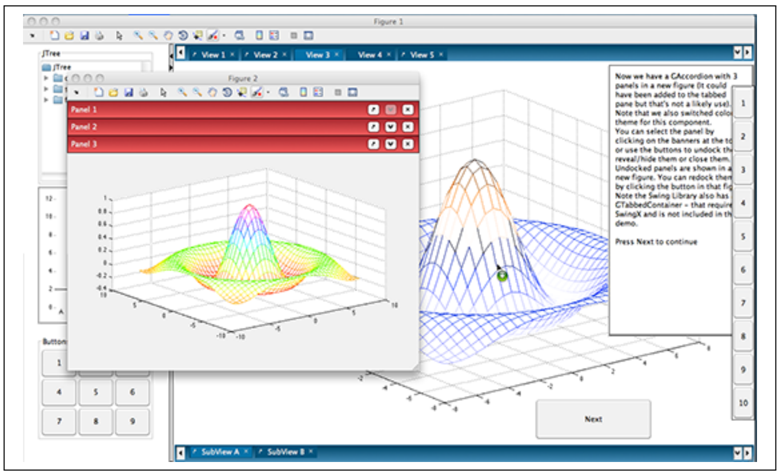


Now we have a GAccordion with 3 panels in a new figure (it could have been added to the tabbed pane but that's not a likely use). Note that we also switched colo theme for this component.
You can select the panel by clicking on the banners at the to or use the buttons to undock th reveal/hide them or close them. Undocked panels are shown in a new figure. You can redock them by clicking the button in that fig Note the Swing Library also has GTabbedContainer - that require SwingX and is not included in th demo.

Press Next to continue

Making MATLAB® Swing

Summary of the Project Waterloo Swing Library for MATLAB

Malcolm Lidierth
Wolfson Centre for Age-Related Diseases

http://sigtool.sourceforge.net/

Revised: 25th September 2011

Acknowlegements:

MATLAB code in this document was styled using Florian Knorn's
M-code LaTeXPackage.
http://www.mathworks.com/matlabcentral/fileexchange/8015-m-code-latex-package

The document was prepared using Pascal Brachet's TEXMaker
http://www.xm1math.net/texmaker/

Contents

The Project Waterloo Swing Library provides widgets and tools to create a richer desktop experience than is available as standard in your MATLAB code. The tools mix MATLAB graphics with Java Swing and, occassionally, its extensions from the SwingLabs SwingX Project at java.net. Simple-to-use wrappers are provided for these components in MATLAB.

Unlike the standard MATLAB uicontrol family of tools, the GTool wrappers expose the underlying Java Swing objects for further programmatic use. They are therefore far less restrictive and allow much richer GUIs to be developed. This document provides an overview of the library. Each function is fully documented in its in-line code although a basic knowledge of MATLAB, Java and OOP is assumed.

Requirements

The library is platform-independent but requires MATLAB R2008a or later. It can be accessed from:

http://sigtool.sourceforge.net/

License

Setting up

Unzip the project folder and place the main folder named ”waterloo” on your MATLAB path. This contains an m-file, waterloo.m, that will do the rest of the setup on demand in each MATLAB session. This avoids adding the subfolders and Java code to your MATLAB paths unless they are needed. Just run
waterloo()
at the MATLAB prompt or include the command in any code that uses the Library.

The jcontrol class

The jcontrol class allows Java Swing components to be added and used easily in MATLAB figures and other MATLAB graphics containers. The jcontrol class makes use of an undocumented MATLAB function, javacomponent, which does most of the work. Full details are provided in Yair Altman's forthcoming book[1]. While the jcontrol class is used extensively within the Swing Library, the wrappers largely hide them from the end-user, so only a brief description is given here. To add a Swing component to a MATLAB container such as a figure or uipanel invoke jcontrol providing the MATLAB HG handle of the parent component as the first argument. The Swing component can be provided

1. as an instance or
2. specified using a string

If a string is used, the component must have a default constructor with no input arguments. In all cases you need also to provide the required position in the parent container as a 4-element vector [*x y width height*]. Normalized units are assumed by default and positions follow the MATLAB convention where the origin is bottom-left e.g.

```
j=jcontrol(figurehandle, javax.swing.JButton('Button 1'),...
                    'Position', [0.1 0.2 0.2 0.1]);
```

[1]Y. Altman,Undocumented Secrets of MATLAB-Java Programming, CRC Press, 2011, ISBN 9781439869031

The jcontrol class lets you add pretty much anything supported in Java Swing to a MATLAB graphics container such as a figure or uipanel. You are not restricted to the standard Swing components - custom Java Beans can be added. The figure below shows Kai Toedter's JCalendar from

http://www.toedter.com/en/jcalendar/index.html

drawn in a MATLAB figure window: You can also, in effect, create new

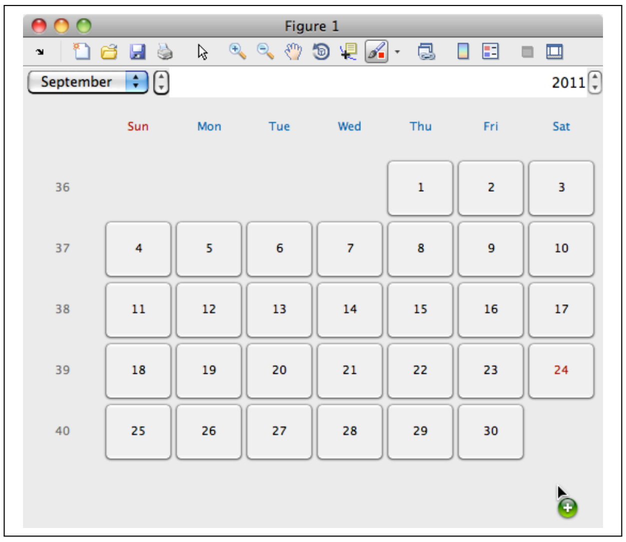


Figure 1: A JCalendar in a MATLAB figure window

components from within MATLAB by creating many jcontrol objects. In general though, a better approach is to create these inside a single jcontrol that has a Java Swing container as its component. So, if you need 5 buttons, create a JPanel and add 5 JButtons to that:

```
button=cell(1,5);
j=jcontrol(figure(), javax.swing.JPanel(),...
     'Position', [0.2 0.8 0.6 0.15]);
for k=1:5
     button{k}=j.add(javax.swing.JButton(num2str(k)));
end
```

Try resizing the MATLAB figure and see how the buttons in the JPanel behave. The layout is being managed by a center aligned java.awt.FlowLayout which is the default layout manager for the JPanel. You could change this if you wanted, e.g. try

```
j.setLayout(java.awt.GridLayout(1,5));
```

and see how the resizing behaviour changes.

In the example above, the buttons were added to a cell array. While you can make an array of jcontrols, it is often easiest to use a cell array instead. There may be some efficiency advantages to this also with large numbers of jcontrols- cell arrays do not require a single contiguous memory block in MATLAB.

Directly access the buttons using normal MATLAB syntax, e.g.:

```
button{1}.setText('OK');
```

The GTool superclass

GTool is the superclass for most other classes in this package. It provides methods used in common by other components and a set of static methods for controlling the color schemes. GTool has no constructor. Instead, you create instances of GTool subclasses by calling the constructors for those subclasses.
In general, each instance of GTool subclass has an object that does the work and a set of components that the object controls. The object is usually a Java component but sometimes it is a MATLAB component container. A few classes have no object - in those cases the components do the work.

GTool classes exist in parallel with MATLAB's graphical hierarchy. While each instance of a GTool class has a MATLAB parent container such as a figure or uipanel[2] , the GTool does not form part of the MATLAB graphics hierarchy. Here are some of the methods:

obj.getParent()
Returns the handle of the associated MATLAB graphics container

obj.getObject()
This returns the object that is central to the component - it either controls it or contains it. You will not need this unless you want to coerce the object into behaving in a non-standard way

obj.getComponents()
Returns the components this object contains as a cell array.

obj.getComponent(n)
Returns the component specified by n. Components are numbered from 1 following the usual MATLAB convention.

obj.getComponentCount()
Returns the number of components in this object.

[2]GWait and GProgressBars are exceptions: they use a JFrame and exist outside the normal MATLAB hierarchy

GSplitPane

The GSplitPane allows you to divide a MATLAB container into 2 components that can be resized using the mouse. Create one using:
g=GSplitPane(figure(), 'vertical');[3]

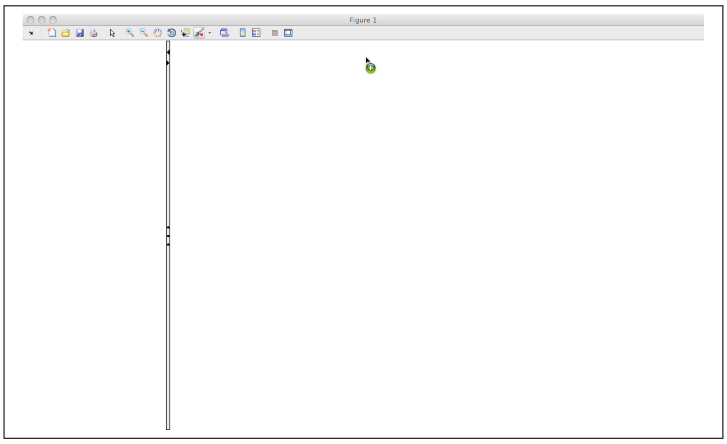


Figure 2: A GSplitPane with nothing added to the uipanels

A split pane has two components, a uipanel on each side of the divider. You can populate these as usual with graphics. With the GSplitPane *g* from above
myaxes=axes('Parent', g.getComponent(2));
will add a set of MATLAB axes to the right side uipanel (component 2).

Plot to the axes in the usual way:
line (1:10, 1:10, 'Parent', myaxes)
GSplitPanes, like most GTools, can be nested. Here we use nested panes and set the positions of the dividers programmatically by calling setProportion:

```
fig=figure();
pane1=GSplitPane(fig, 'vertical');
pane1.setProportion(0.2);
% Now use the right side uipanel for another split pane
pane2=GSplitPane(pane1.getComponent(2), 'horizontal');
pane2.setProportion(0.2);
ax1=axes('parent', pane1.getComponent(1));
```

[3]Remember you need to run waterloo first to set-up the path

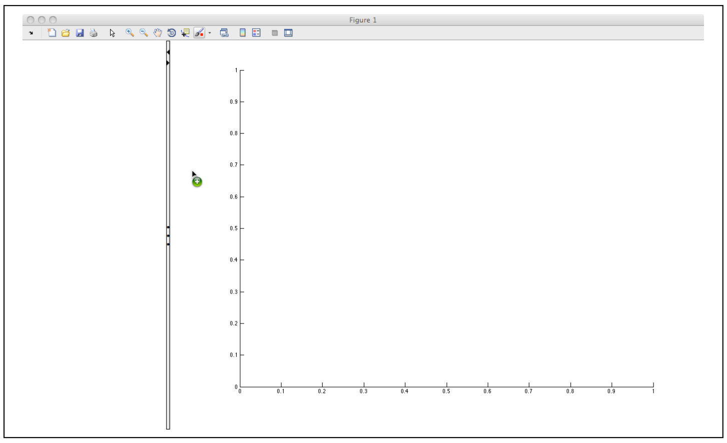


Figure 3: A GSplitPane with MATLAB axes added on the right

```
plot(ax1, 0:0.1:2, sin(0:0.1:2));
ax2=axes('parent', pane2.getComponent(2));
plot(ax2, 0:0.1:5, sin(0:0.1:5)+(0:0.1:5));
ax3=axes('parent', pane2.getComponent(1));
plot(ax3, 0:0.1:20, sin(0:0.1:20)-(0:0.1:20));
```

Here is the result:

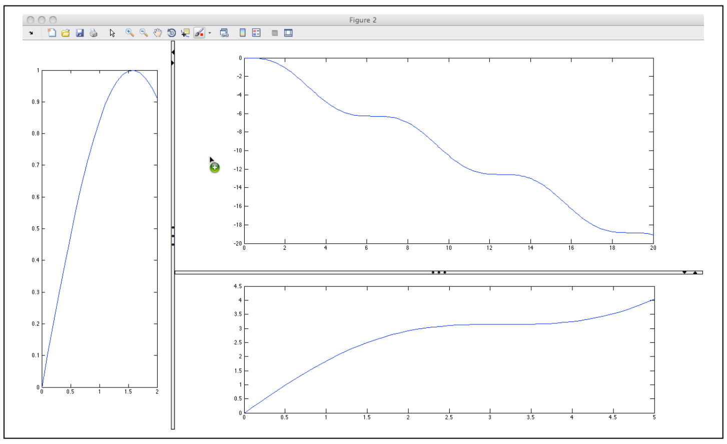


Figure 4: Mutiple split panes in a MATLAB figure window

Note that, when you drag the dividers, the screen continually updates. This is fast, but maybe not fast enough in all cases. GSplitPanes, and many other components have a setAnimated method to turn off the continous updates. Try it pane2.setAnimated(false);

The divider will move as before, but the remaining graphics is only updated at the end.

As well as the GSplitPane, there is a GElasticPane. This allows the user to resize it with the mouse but springs back to its programmed position when the mouse is released. For GElasticPanes, only one side is usually populated and that drags over, and obscures, all other graphics when it is moved.

GCardPane

The split panes allow a MATLAB figure to be split into regions. GCardPanes add depth. With a GCardPane, uipanels can be placed in a stack like a deck of cards with only one of the cards being visible at any time.

The GCardPane does not provide GUI-based control of which card is showing. Instead, GCardPanes are added to other classes that do provide this control. For the GTabbedPane, control is implemented using a standard Swing JTabbedPane (or an extension of it such as the JideTabbedPane which is included in the MATLAB distribution). Better for MATLAB use are the GTabContainer and its partner GAccordion. These use custom controllers combining Swing and MATLAB.

GTabContainer

Creating a tabbed container is easy:

```
GTool.setTheme('blue');
g=GTabContainer(figure(), 'top');
g.addTab('Panel 1');
g.addTab('Panel 2');
axes('Parent', g.getComponentAt(1));
surf(peaks(30));
axes('Parent', g.getComponentAt(2));
contour(peaks(30));
```

Note here, that components are added by calling a custom method, in this case addTab(). This creates a new component, a uipanel, that can be populated as before. The only difference is that the panels each fill the parent

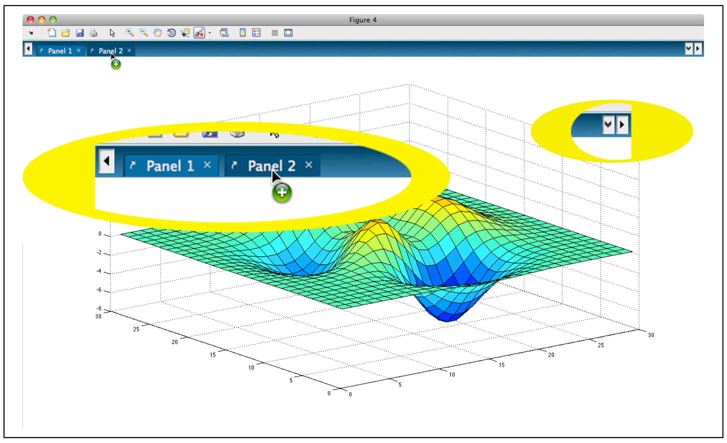


Figure 5: A GTabContainer with two tabs. The expanded sections views show the controls for scrolling through the tabs and for undocking and closing them

container and only one is visible at any time. Also, we called the GTool static method setTheme here. This sets up the color scheme.

The tabs at the top control which uipanel is visible. You can also control that programmatically withg.setSelectedIndex(n);. The tabs also have controls that allow a uipanel to be undocked from the container (and opened in a new figure) or closed. With many tabs, you can scroll through those available using the arrows or the dropdown list that appears when you click the button. These features are programmable.

GAccordion

A GAccordion is much like a GTabContainer but instead of a single row of tabs, each added uipanel has its own banner. These baners are always visible and you can select which associated uipanel is visible by clicking on the banner or using the button associated with each banner. The banners take up space, so a GAccordion will usually only be useful when you have a few uipanels in the card pane.

Construct a GAccordion as you would a GTabContainer, e.g.:

```
g=GAccordion(figure());
g.addTab('Panel 1');
```

```
g.addTab('Panel 2');
g.addTab('Panel 3');
```

As before, individual cards can be undocked to a new figure or closed and this ability is user-programmable.

Note: With both GTabContainers and GAccordions, undocked panels can be redocked into the original container by clicking the associated redock button - though this behaviour can be deselected by programmers. When a card is redocked, it is placed at the index location it occupied when it was undocked- so the cards will be shuffled if redocking is not done in exact reverse order.

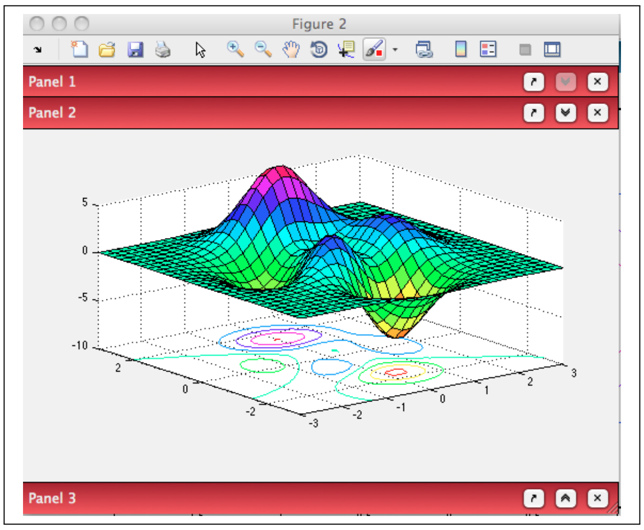


Figure 6: A GAccordion with three tabs. The theme for this GAccordion has been set to 'red'

GTaskPaneContainer

Java-savvy users might also use the GTaskPaneContainer. This requires that the SwingLabs SwingX library from Oracle - so that needs to be on your MATLAB javaclasspath. Only Swing components can be added to a GTaskPaneContainer - not MATLAB graphics. The addTab() and getComponent(n) methods return org.jdesktop.swingx.JXTaskPane objects, not MATLAB HG handles.

GFlyoutPanel

GFlyoutPanels extend the general-purpose GHotSpot class that provides components that are sensitive to mouse-movement. With these you can create panels that appear only when the mouse is near the edge of a figure - rather like the dock on a Mac.

GFlyoutPanels can be placed at any edge of a figure e.g. to place a panel on the right

```
g=GFlyoutPanel(figure(), 'right');
```

A GFlyoutPanel has just one component, a uipanel, that can be retrieved as usual with getComponent(1) and populated as required. If you need a broader panel than the default, use setWidth() to make it wider (or deeper for horizontal panels). The code below simply adds a GFlyoutPanel with 10 Swing JButtons.

```
 % Add a GFlyoutPanel to the right of figure
flyoutPanel=GFlyoutPanel(figure(), 'right');
% Add a Swing JPanel to the uipanel associated with the
% GFlyoutPanel
panel=jcontrol(flyoutPanel.getComponent(1), ...
    javax.swing.JPanel(), 'Position', [0 0 1 1],...
          'Background', GColor.getColor('w'), 'Visible', 'off');
panel.setBorder(javax.swing.border.LineBorder(...
         java.awt.Color.black,1,true));
panel.setLayout(java.awt.GridLayout(10,1));
% Put some buttons in the JPanel and have them alter the text in
% the message box through their callbacks
for k=1:10
    button=panel.add(javax.swing.JButton(num2str(k)));
    button=handle(button, 'callbackproperties');
    % For a real application we would need to add a callback ...
        for each button
    %set(button, 'MouseClickedCallback', {@LocalCallback, k});
end
```

The GFlyoutPanel depends for its function on the MouseMotionHandler class which is part of the library. Most of the components in this library have a Java Swing object which provide for detecting mouse clicks and movements within them. For MATLAB graphics containers, only figures have mouse-motion sensitivity. MouseMotionHandler effectively extends this to all other MATLAB components using the existing MATLAB WindowButtonMotion-Fcn callback as a hook.

GScroller and GScrollPane

So far, the graphics have been added within the limits of the standard MAT-LAB graphics window - although we have added depth to the 2D window using the GCardPane and its derivatives. Two classes allow plotting outside of the standard [0 0 1 1] delimited window. These are GScroller, which provides a view that is scrollable in one direction only and GScrollPane which allows 2D scrolling.

GScroller is simplest because most of the work is done internally by the code in its methods. To illustrate, load some data from a MATLAB file (its included as part of the MATLAB distribution):

```
colormap('gray');
x=load('durer.mat');
```

Create a figure. This time we will create a uipanel and add the GTool to that rather than directly to the figure as in the previous examples.

```
fh=figure();
h=uipanel('Parent', fh, 'Position', [0 0 0.3 1]);
```

Add the GScroller as usual; 'left' here indicates that the scrollbar should be placed on the left side of the parent uipanel

```
g=GScroller(h, 'left');
```

Now we can add uipanels, place axes in them and plot the image loaded above from the MATLAB data file

```
p1=g.add();
```

```
axes('Parent', p1);
imagesc(x.X);
axis('off');
p2=g.add();
axes('Parent', p2);
imagesc(x.X);
axis('off');
p3=g.add();
axes('Parent', p3);
imagesc(x.X);
axis('off');
p4=g.add();
axes('Parent', p4);
imagesc(x.X);
axis('off');
p5=g.add();
axes('Parent', p5);
imagesc(x.X);
axis('off');
colormap('gray');
```

The output is shown below (to examine the component better, you might want to add further panels as above).

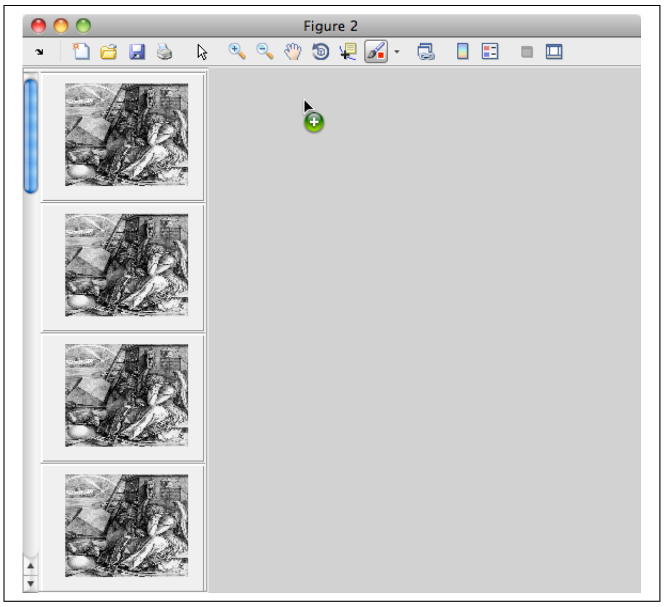


Figure 7: A GScroller with 4 visible panels and a scrollbar to left

Note that the added panels are all the same size. You can control how many panels will be visible at the time of construction using:

```
g=GScroller(h, 'left', n);
```

where n defaults to 4. Control which n of the available panels is visible using the scrollbar or programmatically using the scrollTo method. This accepts an index on input or the handle of the required uipanel e.g.:

```
g.scrollTo(3)
g.scrollTo(p2)
```

The panel that slides in a GScroller can be accessed with the getView() method, though it is simpler to use the provided add() and remove() methods to control its contents. For the more complex GScrollPane though, you will need to call getView() to access the scrolling panel and add/remove components yourself.
GScrollPane offers 2D scrolling and more flexibility but requires more programmer effort. In the simplest case we might add just one set of axes - but these should be put in a uipanel, not added directly to the view.

```
g=GScrollPane(figure());
% Note here the added panel exceeds the usual position limits
h1=uipanel('Parent', g.getView(), 'Position', [0 -1 2 2]);
g.revalidate();
axes('Parent', h1);
x=load('durer.mat');
imagesc(x.X);
colormap('gray');
```

There are a couple of new features here. The revalidate() method was called to update the screen positions. This is called automatically when the parent container is resized. Also, the uipanel was added outside of the normal [0 0 1 1] limits. GScrollPane coerces MATLAB's graphics into painting these regions, hopefully as intended. To have more than one panel:

```
g=GScrollPane(figure());
h1=uipanel('Parent', g.getView(), 'Position', [0 0 1 1]);
h2=uipanel('Parent', g.getView(), 'Position', [0.25 -0.75 0.5 ...
    0.5]);
```

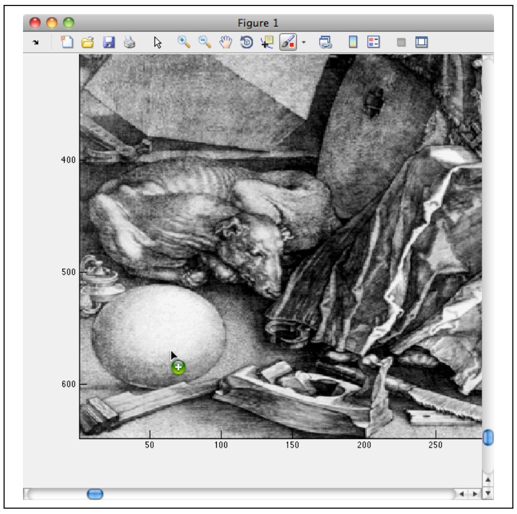


Figure 8: A GScrollPane. The view contains a single panel and has been positioned to show the axes' origin

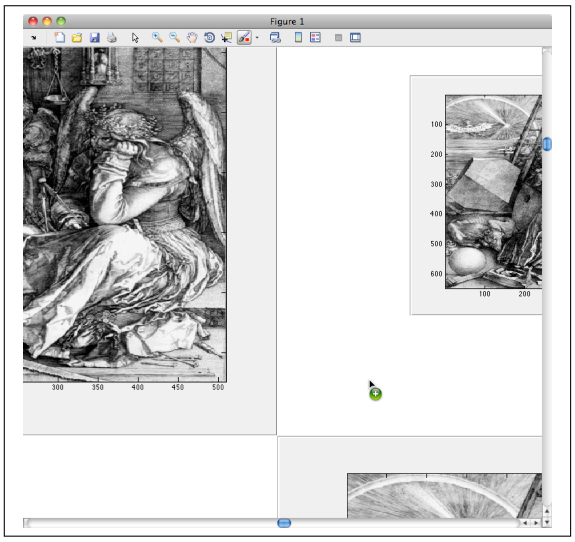


Figure 9: A GScrollPane. The view contains 4 panels and has been positioned to the centre

```
h3=uipanel('Parent', g.getView(), 'Position', [1.25 0.25 0.5 ...
    0.5]);
h4=uipanel('Parent', g.getView(), 'Position', [1 -1 1 1]);
set(get(g.getView(), 'Parent'), 'BackgroundColor', 'w');
g.revalidate();
x=load('durer.mat');
axes('Parent', h1);
imagesc(x.X);
axes('Parent', h2);
imagesc(x.X);
axes('Parent', h3);
imagesc(x.X);
axes('Parent', h4);
imagesc(x.X);
colormap('gray');
```

GWait and GProgressBars

Displaying a progress bar when performing lengthy calculations is useful for the user but including support for these in a loop can be problematic: when a short loop executes many times the code for managing the progress bar can take up a substantial part of the cpu time. The Swing library offers two components to help with this. Both use JFrames rather than MATLAB graphics - they sit in the Java graphics hierarchy of the MATLAB desktop but not in the usually accessible MATLAB graphics hierarchy. You can provide a MATLAB container to the constructor, in which case the position of the GWait will be centered on the container. Otherwise specify 0, the MATLAB root.

The GWait class offers a simple wait bar that shows a message and an image - which can be an animated GIF. There is then no need to control this at all from within your code. although the text can be updated through the setText method. If you do not specify an icon, a default animation from the collection at http://www.sevenoaksart.co.uk will be used. The user can close the GWait bar as usual, and they can be deleted programatically with delete(obj).

```
% Use a default icon
g=GWait(gcf, 'MyFunction', 'Calculation in progress...');
% Use the specified file for the image - this should be ...
    specified with its path or be on the MATLAB
```

```
% search path
g=GWait(gcf, 'MyFunction', 'Calculation in progress...', ...
    'MyImageFile.gif');
```

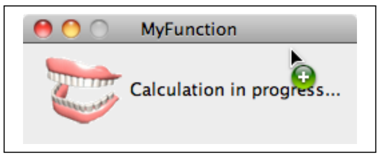


Figure 10: A GWait bar with on of the sillier default animated GIFs from Sevenoaks Art - a set of gnashers.

A GProgressBar adds a progress bar to the display. The bar is updated on a timer, so the graphics is not updated on every loop - GProgressBars do not therefore take up much cpu time or block the MATLAB computational thread. GProgressBars also display an estimate of the time left to completion. This is calculated internally - there is no need to program this yourself.

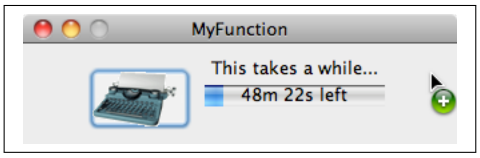


Figure 11: A GProgressBar

Create a GProgressBar with:

```
g=GProgressBar(gcf, 'MyFunction', 'Calculation in progress...');
```

As with the GWait bar, you can add your own preferred icon.

```
g=GProgressBar(gcf, 'MyFunction', 'Calculation in ...
    progress...', 'MyPicture.gif');
```

To use the bar you need to alter the value property:

```
for k=1:100
g.Value=k;
% Your code goes here
% ....
end
```

Setting the minimum and maximum

By default, the progress bar has limits of 0 to 100. Change these using setMinimum(x) and setMaximum(y) .

Controlling the clock

The internal clock is started by default when the progress bar is constructed. Often, you will want to reset it and start it again just before you loop begins:

1. reset() Resets the timer and graphics displaying an "indeterminate" waitbar.

2. start() Sets the clock running. It does not perform a reset.

3. stop() Stops the clock.

```
g=GProgressBar(gcf, 'MyFunction', 'Calculation in progress...')
g.setMinimum(0);
g.setMaximum(50000);
g.reset();
g.start();
    for k=1:50000
    g.Value=k;
    % Your code goes here
    %...
end
```

Keeping things fast

Note that the value in the GProgressBar was set by direct assignment. This is much faster than calling the setValue() method and is therefore preferred when in a loop. Even then, assignment to a property of an object is slow compared to setting the value of a primitive data type. You might prefer to upate the value only on some iterations of the loop:

```
for k=1:50000
if rem(k,1000)==0
    g.Value=k;
end
% Your code goes here
% ...
end
```

Ordinarily, conditional statements are something to minimize in loops, but here the if statement saves time by ensuring that g.Value is updated only once every thousand iterations. The graphical display will be updated only on the next tick of the GProgressBar's internal clock (once every 0.75s by default).

Keeping code tidy

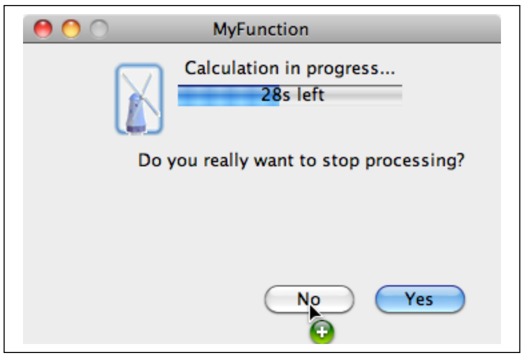
What would happen if the user closed the GProgressBar with the code above? That deletes our object so the call to g.Value=x would throw an error and the loop would exit. In MATLAB, that may be good enough because control will return to the command window, but the GProgressBar has some mechanisms to let you control this process better:

QueryOnClose

Did the user really want to stop the code loop? Good software generally checks this my issuing a prompt to the user to confirm the request. The QueryOnClose property of the GProgressBar provides support for this. This is set true by default. To switch the behaviour off call:

g.setQueryOnClose(false);

When true, the GProgressBar displays a message asking if this was really intended. If the user answers 'No' nothing happens (apart from getting rid of the question). If yes, the GProgressBar is deleted. Note that programmatically deleting a GProgressBar does not invoke the query - it is seen only when the JFrame close icon is clicked.



Calculation in progress...

Do you really want to stop processing?

Figure 12: A GProgressBar displaying the default question presented when the close icon is clicked.

Controlling closing

Setting QueryOnClose to true just issues a prompt. If the user answers 'Yes' the GProgressBar will be deleted and the error in the code loop above will be thrown as before. There are several main ways improve on this. You could just include a call to isvalid in the loop.

```
for k=1:50000
    if rem(k,1000)==0
        if isvalid(g)
            g.Value=k;
        else
            % Clean up code can go here
            % ....
            % Then break out of the loop
            break
        end
    end
    % Your code goes here
    % ...
end
```

This is simple, but adds code in the loop which takes time to execute on each iteration. Better is to include the loop in a try/catch sequence. This adds no code within the main loop so does not slow down the code.

```
g=GProgressBar(gcf, 'MyFunction', 'Calculation in progress...');
g.setMaximum(5000000);
g.reset();
g.start();
try
    for k=1:5000000
        if rem(k,1000)==0
            g.Value=k;
            pause(0.01); % add a pause for demo purposes only
        end
        % Your code for normal operation goes here.....
    end
catch ex
    if strcmp(ex.identifier,'MATLAB:class:InvalidHandle')...
            && isvalid(g)==false
           % Cleanup code goes here....
    else
        rethrow(ex);
    end
end
```

Note that the catch block tests not just for the cause of the exception (an invalid handle) but also that the GProgressBar has been deleted i.e. that isvalid returns false (as other objects may have been the source of the exception - there is no getSource method for MATLAB MExceptions). This code may seem a bit too lengthy to use often. In any case, what happens if the user tries to break out of the loop by hitting control-C instead if closing the progress bar? That has not been accounted for. If we make use of MATLAB's onCleanup class, we can achieve much the same result as above with less code, account for the use of control-C and do less typing:

```
g=GProgressBar(gcf, 'MyFunction', 'Calculation in progress...');
% Set up MATLAB's onCleanup
c = onCleanup(@()LocalCleanup(g));
g.setMaximum(5000000);
g.reset();
g.start();
try
```

```
    for k=1:5000000
        if rem(k,1000)==0
            g.Value=k;
        end
        % Your loop code goes here.....
    end
catch %#ok<CTCH>
end
```

Note that a catch is included here but the catch block does nothing but allow the function to terminate - thus invoking the onCleanup code. All that is left is to define the cleanup routine:

```
function LocalCleanup(g)
if isvalid(g);delete(g);end;
return
end
```

Add any other code, exception checks etc you like to this function. Note we tested the validity of *g* again. For a control-C, *g* will not have been deleted. With closure of the GProgressBar it will have been (by default anyway).

Nesting GTool components

For the most part, GTool components can be nested within each other. For example, the figure below shows a split pane with a GTabContainer on its right and a uipanel containing a GAccordion on its left. The GAccordion has been added to the component list for the GTabContainer. As a result, clicking the buttons in the tab container controls which card is displayed in both the tab container and the accordion.

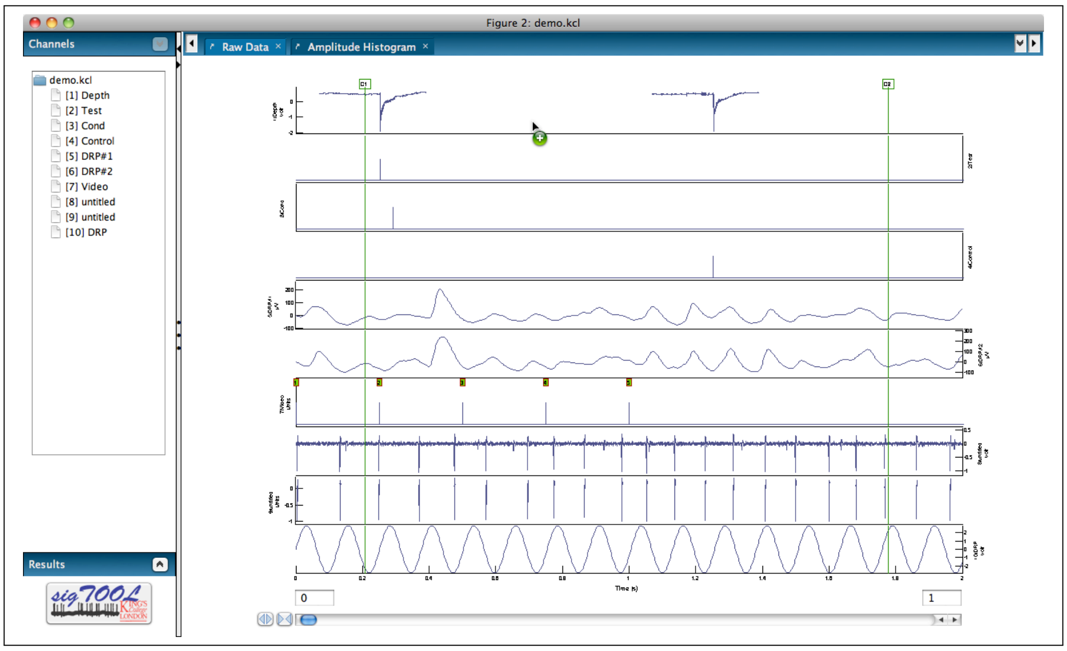


Figure 13: Nesting GTool components. This shows a window from the sigTOOL Project: the MATLAB figure contains a GSplitPane. On the left of the divider is a GAccordion and, on the right, a GTabContainer

Future developments

The Swing Library is one part of a larger Project Waterloo. This includes

1. a set of miscellaneous utilities dealing, amongst other things, with project development and large data sets in MATLAB
2. a Swing-based graphics library that allows scientific graphics and Swing to be easily mixed

The graphics library is Java-based and can be called from Java or MATLAB code (first release expected early 2012). See the project website for further information:

http://sigtool.sourceforge.net/

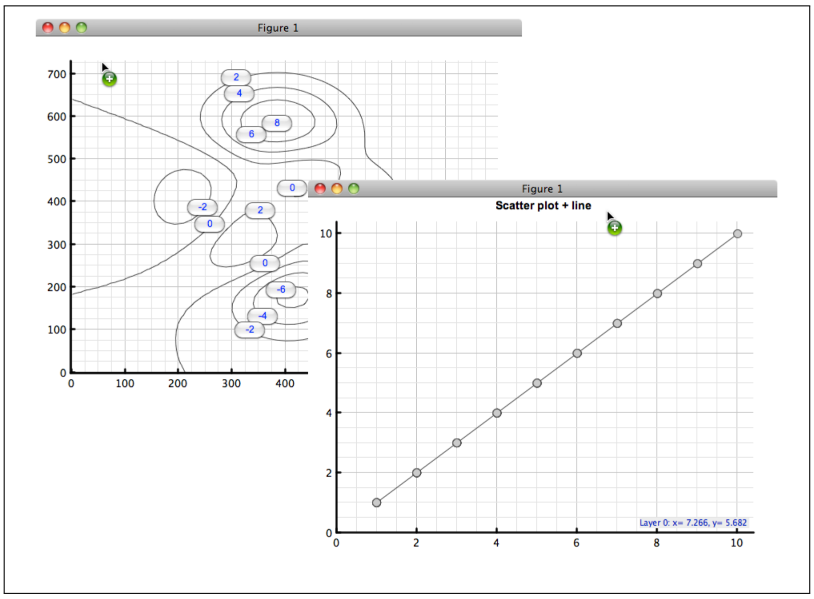


Figure 14: Scatter/line and contour plots in MATLAB created with the Project Waterloo Graphics Library.