Modelit
Elisabethdreef 5
4101 KN Culemborg
+31(345)531717

info@modelit.nl
www.modelit.nl

Modelit Application Framework for Matlab

Version: 2008_01
Date: August 13, 2008

Version:	2008_01
Date:	August 13, 2008
Manual:	Modelit Application Framework for Matlab
Authors:	Nanne van der Zijpp Kees-Jan Hoogland
Copyright:	2008, Modelit
Contact:	info@modelit.nl www.modelit.nl

Table of Contents

1 Introduction

1.1 Objective of the Modelit Application Framework for Matlab

The Modelit Application Framework for Matlab (in short: Application Framework) has been designed for Matlab developers who want to provide their Matlab applications with intuitive and powerful user-interfaces.

Matlab developers can benefit from the Application Framework in two ways:

- Using the framework cuts costs for development and maintenance;
- Applications built with the framework have a number of user friendly features that come at no cost.

Efficiency gains in application development and maintenance

The Application Framework separates data storage and visualization. Proper use of the framework will lead to efficiency gains that grow exponentially with the complexity of the interface one is creating. The following factors contribute to these gains:

- Less technical design decisions;
- Less code and less complexity leading to time savings at implementation;
- Flexibility to expand the interface with extra workspace variables or interface components;
- Reduce risks for error and facilitate testing;
- Designing and building applications in a uniform way facilitates working together on one project and transfering maintenance from one person to another.

Features implemented by the framework

All applications based on the framework automatically benefit from a number of built in features:

- Undo en redo;
- Load and save mechanism for databases;
- Automated update mechanism for the user interface after the data are changed;
- Timed backups;
- Crash recovery;
- Automatic restore interface settings when interface is closed and re-opened.

1.2 Introducing the application framework

The best way to find out what the application framework is about is to consider a few examples.

First example

Consider the commands entered to the Matlab prompt below and their output:

```
>> a=1:10
a =
     1     2     3     4     5     6     7     8     9    10
>> a=undoredo(a)
Undoredo object
     1     2     3     4     5     6     7     8     9    10
>> a(5)=50
Undoredo object
     1     2     3     4    50     6     7     8     9    10
>> a=undo(a)
Undoredo object
     1     2     3     4     5     6     7     8     9    10
```

This example uses the function **undoredo**. In fact **undoredo** is the constructor for the **undoredo** object that is central to the Modelit Application Framework for Matlab. The example illustrates the following properties of the **undoredo** object:

- You can transform any Matlab variable to an undoredo object. In other words an **undoredo** object is initialized with a Matlab variable;
- After initialization you can modify or refer to undoredo objects as if they were ordinary Matlab variables;
- Modifications made to an **undoredo** object may be undone by applying the **undo** method.

Second example
Create a function view.m that contains the following lines:

```
function view(signature,data,ind)
plot(data)
```

Now type the following commands on the Matlab prompt:

```
 >> a=1:10;
 >> a=undoredo(a,'display',@view);
 >> dummy=flush(a);
 >> a(3)=10;
 >> a(7)=[];
 >> dummy=flush(a);
```

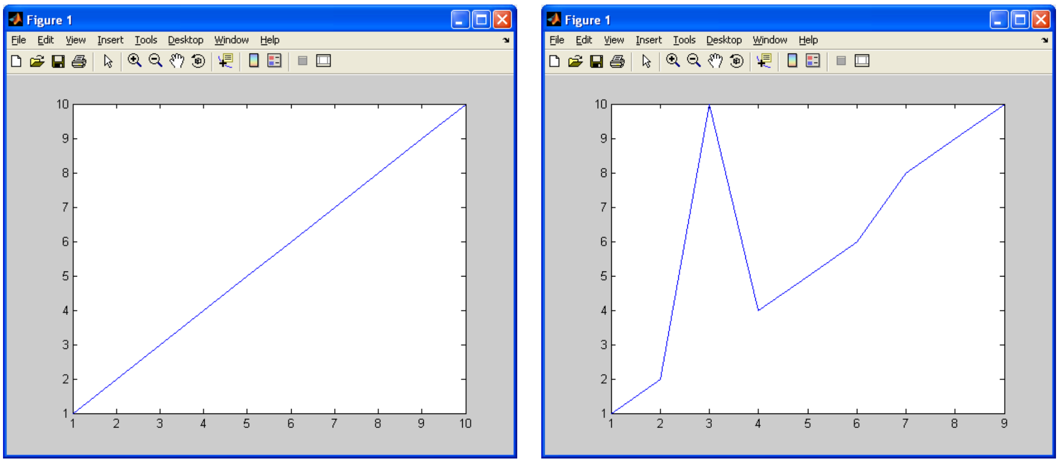


Figure 1
File Edit View Insert Tools Desktop Window Help
10
9
8
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8 9 10
Figure 1
File Edit View Insert Tools Desktop Window Help
10
9
8
7
6
5
4
3
2
1
1 2 3 4 5 6 7 8 9

Figure 1: *Figures that appear with the example*

This example reveals another property of the undoredo object: A displayfunction that visualizes or otherwise deals with the data contents may be attached to it. In the example the displayfunction is specified when the undoredo object is initialized. A flush command is needed to trigger the display function, this makes it possible to specify a group of changes before the interface is updated.

- End of examples -

In a nutshell these examples explain what the Application Framework is about: it allows a Matlab programmer to separate the database of an application from the visualization. In more complex applications the vector "a" of the example will be replaced with a more complex variable, usually a structure, and the display function is more complex, but the basic ideas remain the same.

1.3 On-line help

PDF documentation
The current document provides the user guide and reference manual and is provided as a PDF document.

Manual pages
Large parts of the reference manual included in this document are available as manual pages of the m-functions included in the Application Framework. They can be retrieved by typing the following command on the command line.

```
>> help <functionname>
```

1.4 System requirements

The Modelit Framework for Matlab has been tested with Matlab R2006b, R2007a and R2007b. The framework does not use any undocumented Matlab features and therefore should also work with future Matlab versions.

Any application based on the Application Framework can be compiled to standalone applications with the Matlab Compiler, and works with other builder products, such as the Matlab Builder for Java.

The Application Framework works seamlessly with other toolboxes and subroutine libraries provided by Modelit. Two toolboxes are particularly useful when creating user interfaces:

- *Modelit Layout Manager*. This toolbox supports the design of modular and resizable GUI's. The toolbox allows specifying the position of GUI elements relative to frames using normalized coordinates, pixel coordinates or grid position. It also automates the computation of frame sizes and the nesting of frames;
- *Modelit User Interface Components Toolbox for Matlab*. This toolbox offers access to interface elements such as tabbed panes, sortable tables, trees, comboboxes with autocompletion and so forth without the need to switch to a different programming language or complex code.

1.5 How to proceed from here

Over the years the Application Framework has been applied in many applications. Experience has learnt that all applications can be fit in a specific template. This template is described in chapter 3. It is highly recommended to apply this template for your first applications.

Depending on your needs you may read the following chapters:

- Chapter 2 contains all information for installing the Application Framework;
- Chapter 3 explains the ideas of the framework, gives examples and provides templates;
- Chapter 4 explains the backgrounds of dependency trees;
- Chapter 5 contains the reference manual;
- Chapter 6 discusses a number of advanced topics.

2 Installation

Installing the Modelit Application Framework consists of copying the directories included in the m-file distribution to target directories on your system and including these directories in your Matlab path. For convenience a file install.m that sets the path is included.

Follow the next steps to install the Modelit Application Framework for Matlab:

1. Unzip the files from the MAF.zip file.
   This creates a folder ‘Modelit’ with subdirectories;
2. Find and run install.m. This prints Matlab code that will include the required directories in your Matlab path on the console. These lines should be copied to your startup.m file;
3. Copy these lines to the startup.m file;
4. Run startup.m or restart Matlab.

You may verify the installation by typing:

```
a=undoredo(1:3)
a(2)=20
a=undo(a)
```

This is what you should see if you type the commands one by one:

```
>> a=undoredo(1:3)
Undoredo object
     1     2     3

>> a(2)=20
Undoredo object
     1    20     3

>> a=undo(a)
Undoredo object
     1     2     3
```

3 Templates for applications based on the framework

3.1 Simple template

Characteristics:

- Only utilizes separation between database and interface;
- No undo/redo;
- No automatic detection of interface elements that require an update;
- No figure settings.

The template contains the following elements:

- Create an application entrypoint;
- Initialize a database;
- Create a figure and define callbacks;
- Define a display function;
- Initialize of the undoredo object;
- Force painting the interface for the first time.

Below each element of the template is explained in some more detail. In the next section a complete m-file listing of a simple example application is given. This file is also included in the source code distribution as the file simple.m

Create an application entry point
An application is started by running a specific function. Let's call this function myApplic. We create a file myApplic.m and enter the first line:

```
function myApplic
```

Initialize a database
It is recommended to store the database in a structure. Probably your application will have menu items titled "load" and "save" that load and save the workspace structure as well. When the application starts, an empty database needs to be created. Typically your application will contain a function that takes care of this, let's call this function "init_db". Your code will contain a line that refers to this function and looks like:

```
dbdata=init_db;
```

Create a figure and define callbacks
Your application will probably open a figure when its starts. It is generally a good idea to create a function that takes care of initializing this figure. This function creates the figure and all elements in it including their callbacks. The next line we add to our function is:

```
HWIN=create_fig;
```

The callbacks need to be able to retrieve the central database of the application, for this purpose it is recommended to store this datastructure as the userdata of the main figure window and define a function that retrieves this data. For example:

```
function db=get_db
HWIN=findobj('tag','MAINWINDOW');
db=get(HWIN,'userdata');
```

A typical callback looks as follows:

```
function callback(obj,event)
%retrieve database
```

```
db=get_db;
%modify database
db.field=...
%display modified data
db=flush(db);
%store the database again
store(db);
```

Define a display function

The display function is called every time the central database, which will be defined in the next step, is modified. A display function is always called with 3 arguments: "signature", "data" and "index". For a simple application you only need the "data" argument. This argument passes the database to the display function. So your code must contain a function with the following signature:

```
function view(signature,data,upd)
```

Initialize the undoredo object

We need to initialize an undoredo object in specify the following information:

- The initial data. This is the structure dbdata that was created above;
- The display function. This is the function view as defined above;
- The location where the database is stored. The most common place is to store the database is in the userdata of the main figure of the application. For this purpose we must specify the windowhandle HWIN.

The code that does the job is:

```
db=undoredo(dbdata,...
'display',@view,...
'storehandle',HWIN)
```

Force painting the interface for the first time

Normally changes in the database will trigger the display function, but when the interface is initialized we need to do it ourselves. Therefore we will call the flush method.

```
db=flush(db);
```

3.2 Advanced template

In this section we present a template that contains most of the functionality of the Application Framework. You may not need all this functionality directly, but parts of the template may be omitted.

The template contains the following elements:

- Create an application entry point;
- Create a figure and define callbacks;
- Initialize a database;
- Initialize the undoredo object for the database;
- Define a dependency tree for the database;
- Initialize user preferences;
- Initialize the undoredo object for the user preferences;
- Define a dependency tree for the user preferences;
- Define a display function;
- Define undo and redo functions;
- Define load and save functions;
- Provide a function that is called when the figure is closed;
- Provide a function that is called when the figure is deleted;

- Force painting the interface for the first time.

In the table below the function "main" initializes the interface. A function "create_fig" will create all handle graphic objects and install the callbacks and needs to be supplied by you. Examples of GUI callbacks are included in the template. The functions in the following table are needed by the function “main”.

In the source code distribution a file advanced.m is included in the examples directory. This file includes the template, complemented with a few lines of code that implement a simple GUI.

At first sight the template might look somewhat elaborate, but if you copy and adapt the template, you'll find that most of the template can be re-used, so that the amount of code that you need to provide yourself to get your first example running is very limited.

Function name	Description and Example code
main	The function "main" initializes the interface. The functions the rows below will be called

```
function main
%this function creates a GUI

%check if the GUI already exists:
if ~isempty(mainWinHandle)
    %the GUI already exists, no further action is needed
    return
end
%rhe GUI does not yet exist, create the GUI
HWIN=create_fig;
%register closerequestfunction and deletefunction:
set(HWIN,'closereq',@closereq,...
    'deletef',@deletef);

%create undoredo object for user preferences:
opt=undoredo(initopt,...
    'displayfunction',@displaysettings,...
    'mode','simple',...    % no undo for userpref settings
    'storeh',HWIN,...      % store with current figure
    'storef','userprefs'); % store in application data
"userprefs"
%register dependency tree for user preferences
setdepend(HWIN, opt, settings2win);
%maker user preferences available to future callbacks
store(guiopt);

%initialize or recover workspace data
[data,COMMITTED]=emptyDb;
%create undoredo object for workspace data
db=undoredo(data,...
    'backupfile',autoSaveFile,... % apply periodic backups
    'displayfunction',@displaydata,...
    'storehandle',HWIN,...        % store with current figure
    'storefield','userdata',...   % store in "userdata"
    'mode','memory');             % allow undo, do not cache
undo history
%for very large workspaces consider:
%     'cachefile','cacheFile',...
%     'mode','cached');

%change COMMITTED status if required:
db=setcommitted(db,COMMITTED);
```

	%register dependency tree for workspace data:
	setdepend(HWIN, db, data2win);
	
	%Draw interface for the first time. Add the 'all' argument to flush to
	%force that all user interface elements will be drawn:
	db=flush(db,'all');
	%make workspace data available to future callbacks:
	store(guiopt);
	%end of inialization of interface

mainWinHandle	This function returns the handle of the application startup figure
	
	function HWIN=mainWinHandle
	HWIN=findobj('tag','MAINWINTAG');
	set(HWIN,'closereq',@closereq);

create_fig	This is a user specified function that creates the main window of the application and all the handle graphic objects in it. function HWIN=create_fig
initOptFile	This function returns the name of the file that stores the user preferences. Choose a unique file for each type of figure. function str=initOptFile str='userpref.stt';
autoSaveFile	This function returns the constant with the name of autosave file. function str=autoSaveFile str='autoSave.tmp';
get_db	This function retrieves the database. function db=get_db db=get(mainWinHandle,'userd');

get_opt	This function retrieves the user preferences.
	
	function opt=get_opt;
	opt=getappdata(mainWinHandle,'userprefs');

check_exit	This function is called when there might be any present data that need saving before exiting the application or loading new data from file.
	
	function status=check_exit
	% OUTPUT
	% status
	% 0: no unsaved data
	% 1: unsaved data. these have been saved.
	% 2: unsaved data. these have not been saved.
	% -1: unsaved data. user pressed cancel
	db=get_db;
	if(iscommitted(db))
	  %All data are already saved
	  status=1;
	  return
	end
	
	%Unsaved data exist
	switch questdlg('Save data?',...
	        'Close workspace','Yes','No','Cancel','Yes')
	    case 'Yes'
	        if saveWorkspace;
	            status=1;

	else
	            status=-1;
	        end
	    case 'No'
	        status=2;
	    case 'Cancel'
	        status=-1;
	end

saveWorkspace	This is a typical callback for the uimenu item with label "Save as".
	
	function ok=saveWorkspace(obj,event)
	saved=false;
	db=get_db;
	fname=askFileName %function not provided in template
	if isempty(fname)
	  return
	end
	data=getdata(db);
	save(fname,'data');
	db=setcommitted(d);
	store(db);
	saved=true;

loadWorkspace	This is a typical callback for the uimenu item with label "Load".
	
	function loadWorkspace(obj,event)
	%check if present data require save
	if check_exit==-1
	    %user cancels
	    return;
	end
	%ask filename
	fname=askFileName %function not provided in template
	
	if isempty(fname)
	    %user cancels
	    return
	end
	
	%load data from file
	s=load(fname);
	%replace data content of undoredo object
	db=get_db;
	db=setdata(db,s.data);
	%display data. Note that all items need to be redisplayed
	store(flush(db,'all'));

closereq	This is the closerequest function of the application startup figure. It will be called when the user attemps to close the figure. If there are any unsaved data, ask the user what to do. If the user cancels, the figure will not be closed.
	
	function closereq(obj,event)
	if check_exit==-1
	  %user has cancelled
	  return
	end
	delete(mainWinHandle);

deletef	This is the deletefunction of the application startup figure. It will be called when the the figure is closed. In this function the following tasks will be performed:

<table>
<tr>
<td></td>
<td>

- The user preferences for this figure will be retrieved and stored, so that when the figure is opened again these can be imported again;
- Any cache files and autobackup associated with the workspace data will be removed.

```
function deletef(obj,event)
%exit function of application main screen

%save user preferences
%enclose in try catch just in case
try
    guiopt=get_opt;
    opt=getdata(guiopt);
    %remove any items that do not need saving
    %(function not provided in template)
    opt=removeItems(opt)
    save(initOptFile,'opt');
catch
    %continue
end

%Close all figures
delete(findobj('type','figure'));

%remove autosave and cache files, if present
try
    db=get_db;
    cleanupdisk(db);
catch
    %db was corrupt
end
```

</td>
</tr>
<tr>
<td>initOpt</td>
<td>

This function initializes the data structure that represents the user preferences. This is done in 3 steps:

- Create a structure that contains the factory defaults. This is to make sure that no errors will occur if the figure is openened for the first time. Generally the user preferences are initialized with a number of fields already present. Otherwise the "isfield" check will be needed everytime a field is used or modified;
- Load the user preference s as saved when the figure was closed last time (see function template "deletef"), and overwrite the factory defaults with the values that are loaded from file;
- Set any values that are specific for the current session. For example, you may store object handles in the user-preference structure.

```
function initOpt
%assign factoryDefaults (function not provided in template)
opt=factoryDefaults;
%apply saved user preferences
opt=ur_getopt(opt,initOptFile);
%ovverid options where applicable
opt.currentFile='untitled.mat';
```

</td>
</tr>
<tr>
<td>initDb</td>
<td>

Like initOpt this function provides an inital value for an undoredo object, in this case the database. Generally the workspace is initialized with a number of fields already present. Otherwise the "isfield" check will be needed everytime a field is used or modified. Unlike the user preferences the workspace data from the previous session should generally not be reloaded. The exception is if the application has not been terminated in a normal manner, for example by a power failure or a user pressing Ctrl+Alt+Delete. In

</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td>

this case the autosave file will still be present and should be loaded.

```
function [data,COMMITTED]=initDb
%initialize empty workspace
%
% OUTPUT
%     data:      data for workspace
%     COMMITTED: if false: data have been recovered and have
not yet been
%                saved
if exist(autoSaveFile,'file')
    %recover data after crash
    disp(sprintf('Recover workspace from %s',autoSaveFile));
    s=load(autoSaveFile);
    data=s.data;
    COMMITTED=false; %these data have not yet been saved
    return
end
%call user specified function emptyDb for initialization
db=emptyDb;
COMMITTED=true;
```

</td>
</tr>
<tr>
<td>emptyDb</td>
<td>

```
function data=emptyDb
%this function must be specified by user
%and is not part of template
```

</td>
</tr>
<tr>
<td>settings2win</td>
<td>

This function returns a dependency tree for the user preferences. A dependency tree is a structure that resembles the database of an application. At each node of this structure a field "updobj" may be added. This field should contain a cell array with the name or names of the update actions that are required when an assignment is made that affects this node or any of its children.

```
function upd=settings2win
%This function is specific for each application
%Example
upd.currentFile.updobj={'displayfilename'};
```

</td>
</tr>
<tr>
<td>data2win</td>
<td>

Return the dependency tree for the database.
Example:

```
function upd=data2win
%This function is specific for each application
%Example
upd.data.updobj={'graph'};
upd.data.x.updobj={'domain'};
upd.data.y.updobj={'reach'};
```

</td>
</tr>
<tr>
<td>displaysettings</td>
<td>

This function will be called when the flush method is invoked on the undoredo object that holds the user preferences. The update structure is evaluated using the evaldepend function. Subsequentlty a view function will be called that is shared with the display function of the database.

```
function displaysettings(signature,opt,ind)
HWIN=gcf;
upd = evaldepend(HWIN, ind, signature)
db=get_db;
view(getdata(db),opt,upd);
```

</td>
</tr>
<tr>
<td>displaydata</td>
<td>

This function will be called when the flush method is invoked on the undoredo object that holds the database. The update structure is evaluated using the evaldepend function. Subsequentlty a view function will be called that is shared with the display function of the user preferences.

</td>
</tr>
</table>

	`function displaydata(signature,db,ind)`
	`HWIN=gcf;`
	`upd = evaldepend(HWIN, ind, signature)`
	`opt=get_opt;`
	`view(db,getdata(opt),upd);`

view	This is the function that actually displays the interface, given the database *and* the user preferences. It will be called from either displaysettings or displaydata (see above).
	`function view(db,opt,upd)`
	`%This function is specific for each application`

3.3 Example application: the game of GO

Games make good example applications because readers know the rules of the game and hence what type of functionality the example implements. In the current case we use the board game Go (http://en.wikipedia.org/wiki/Go_(board_game)) as an example.

The source code playgo.m is included in the source code distribution and creates the interface using the Application Framework. The playgo example resembles the "simple" template more than the "Advanced template".

As an exercise one may add extra features to the playgo example by using parts of the "Advanced template".

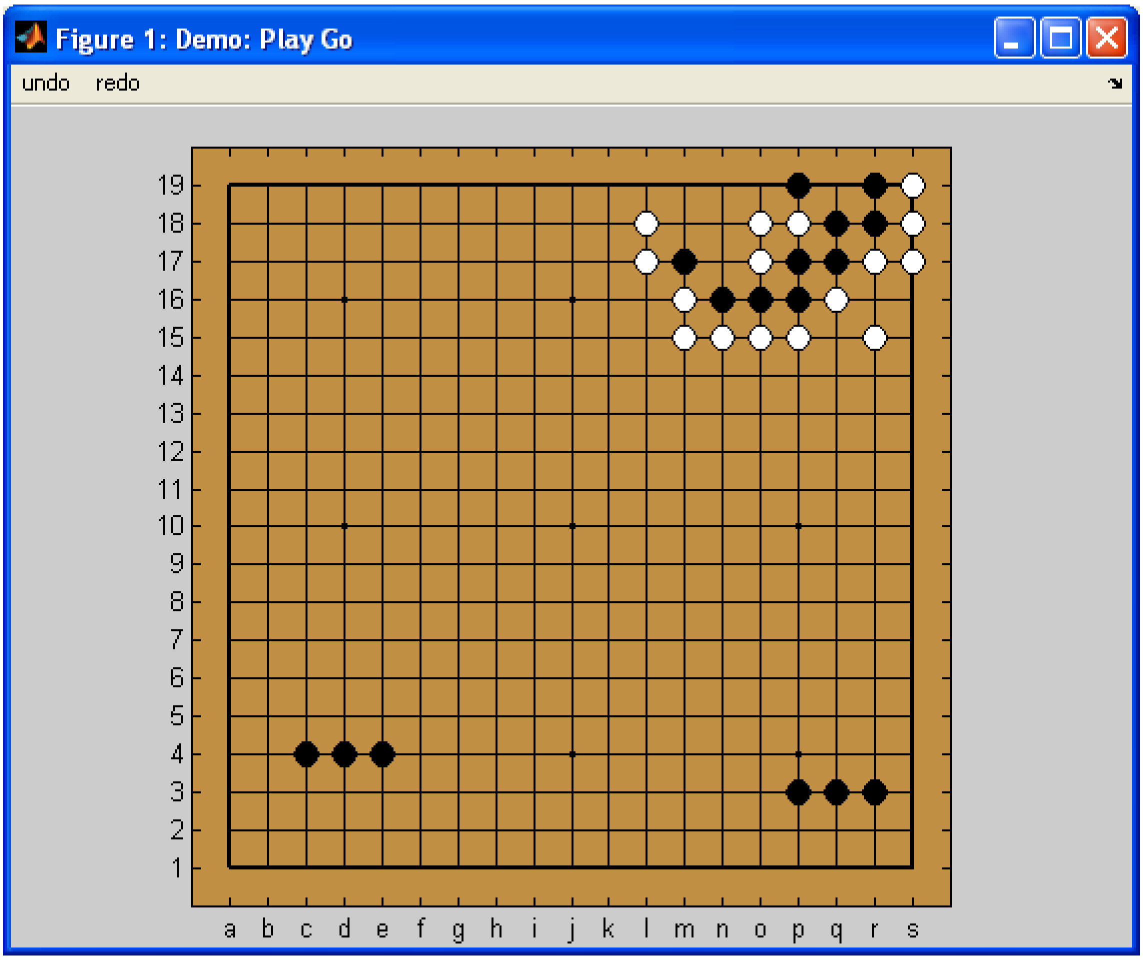


Figure 2: *The function "playgo" creates GUI that allows users to play the well known GO game on a computer.*

4 Specifying dependency trees

4.1 Introduction

The complexity of a GUI may be characterized by the number of *fields* that make up the data model and the number or *elements* that are displayed in the GUI. When a field in the datamodel changes GUI elements that depend on it must be updated. We call this an *interaction*. The potential number of interactions increase with the product of the number of fields and the number of elements, therefore twice the amount of objects and elements mean four times the amount of interactions.

Of course one does not need to code each interaction separately. Instead one may repaint the entire interface if any of the data fields change. However, this approach comes at the cost of high response times and sluggish interfaces. The Application Framework offers a mechanism to work around this: dependency trees.

When a GUI is implemented using the Application Framework it will contain a *displayfunction* that is responsible for displaying any element of the GUI. The general structure of a displayfunction looks like:

```
function ok=displayfunction(signature,data,ind)
upd=evaldepend(gcf,ind,signature)
if upd.element1
     %<paint element 1>
end
if upd.element2
     %<paint element 2>
end
%etc
```

In the code above the *update structure* "upd" is returned by the function **evaldepend**. The update structure contains a number of boolean fields. These fields are used as flags that indicate whether or not a specific part of the displayfunction is bypassed or not. The function **evaldepend** computes the update structure based on a *dependency tree* that is stored by the function **setdepend**. This is done when the application initializes a figure.

Together the function **setdepend** and **evaldepend** provide functionality that is useful for creating complex interfaces that still respond quickly. In this chapter we describe the background of dependency trees.

Note:
If you are creating lightweight interfaces only, you do not need **setdepend** and **evaldepend** , and you may skip the remainder of this chapter.

4.2 Dependency matrix

Ideally, the display function should only update those elements that depend on the modified data and leave other interface elements unaffected. To accomplish this we need to complete two tasks:

- Define a number of GUI elements. You may opt for a refined method, for example by identifying each line in a graph as a separate element, or apply a more aggregate approach, for example by identifying a complete graph as an element;
- Define the dependencies between the fields in the datamodel and the GUI elements. Again you may choose a very detailed approach, for example that field "db.a.b.c" impacts element 1 and "db.a.b.d" impacts element 2, or a more aggregate approach, for example that field "db.a" impacts element 1 and element 2.

Generally it is a good idea to start with an aggregate approach that will be refined as part of the profiling process.

The matrix below shows a typical dependency matrix. The vertical axis shows the fields that make up the datamodel. Essentially this is a list of fields, but these may be organized in groups, and the groups are part of either workspace data or user preference data. Although not indicated in the matrix, groups may again be organized in a hierarchical manner. In practice data will be organized in structures.

Basically the dependencies are defined at the level of GUI elements and fields, where each GUI element depends on one or more fields. This is indicated with blue-shaded cells in the matrix below. This matrix is called a *dependency matrix*.

Table 1: *Dependencies specified at field level*

Category	Field	GUI elements																			
workspace data	fld1																				
	+----fld11		■											■			■	■	■		
	+----fld12		■														■	■	■		
	fld2																				
	+----fld21			■	■	■	■	■	■	■					■			■	■		
	+----fld22														■						■
user preferences	fld3																				
	+----fld31		■					■	■	■	■	■	■	■	■	■	■	■	■	■	
	+----fld32		■																		
	fld4																				
	+----fld41				■	■	■	■					■						■		
	+----fld42				■	■	■	■					■						■		

■ dependency defined at field level

4.3 Specifying the dependency matrix: dependency tree

In this section we will show how the information needed for creating the dependency matrix is specified. A condition for this is that the datamodel of our application is represented by a Matlab structure. Consider the structure displayed in Figure 3. Note that only the lowest level nodes (the leaves) hold data. The other nodes do not store data directly.

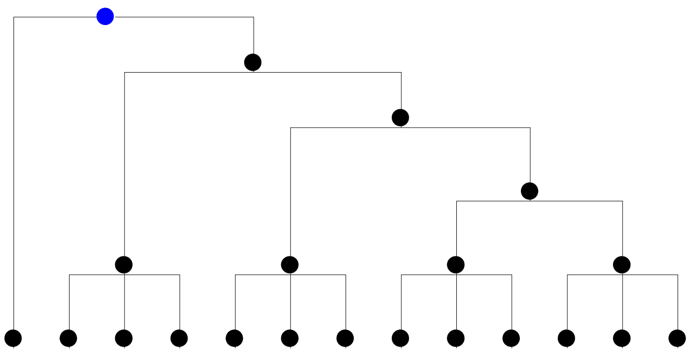


Figure 3: *Workspace data organized in a structure.*

A dependency tree is created by adding a cell array with the name "updobj" as a field to nodes of the datamodel. Specifying these fields is optional. Omitting them is equivalent to adding an empty cell array.

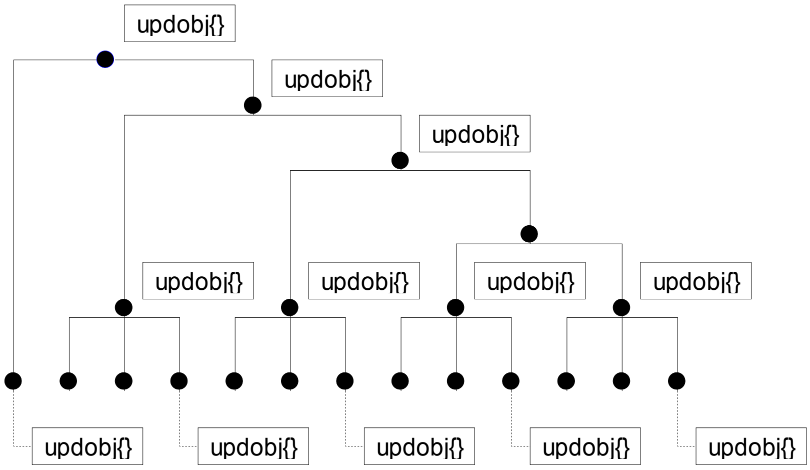


Figure 4: Specifying a dependency tree

The cell array "updobj" contains the names of the GUI elements that need updating when any structure field that is a direct or indirect child of the current node is modified. These names will then reappear in the update structure that is evaluated by **evaldepend**.

Note that the updobj" array can be specified at any level and not only the lowest level. Specifying "updobj" at a specific node is equivalent to specification at all its child nodes. This speeds up the specification process .

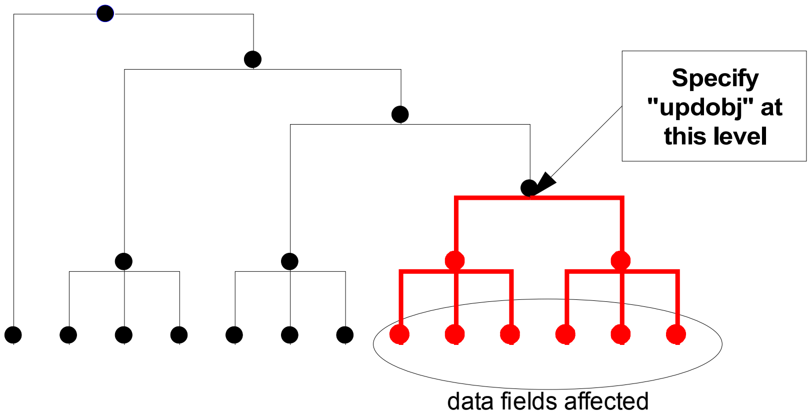


Figure 5: *Instead of specifying the cell array "updobj" at the lowest level, on may save time by specifying it at a higher level.*

4.4 Evaluating the update tree

When a field in the datamodel changes the update structure can be computed by **evaldepend**. The function **evaldepend** is called from the displayfunction using the arguments: figure handle, signature and subsasgn structure(s). The latter are stored in a cell array.

The figure handle is used to retrieve all dependency trees that are registered with the figure. This makes it possible to initialized the update structure with all required fields having the value "false". In this way we may use statements like

```
if upd.element1
```

```
    %<paint element 1>
end
```

instead of:

```
if isfield(upd,'element1') && upd.element1
    %<paint element 1>
end
```

The signature tells the evaldepend function which dependency tree is corresponds to the subsagn argument that is considered

The subsasgn argument is a cell array that corresponds with the assignments that have been done to the undoredo object since the last update. It tells evaldepend which node of the datamodel has been modified. Type "help subsasgn" for Matlab help on this topic.

If we consider Figure 6, modifying a node implies that all child nodes are changed. Hence the update actions specified in the "updobj" cell arrays with the child nodes should be executed. Also the update actions specified at the parent nodes will be executed. The reason for this is that specifying an update object at a parent node implies that this action is applicable to all its child nodes.

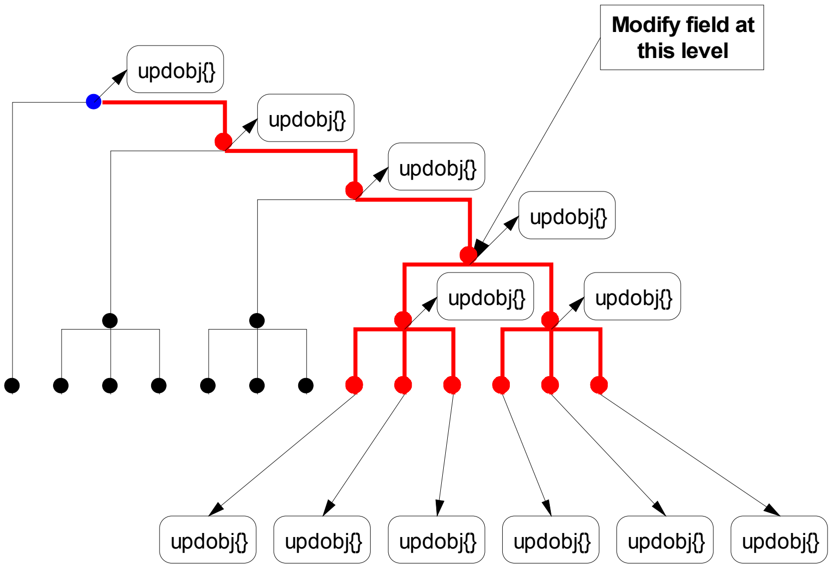


Figure 6: *When an assignment is made to a node in the datamodel, the "updobj" cell arrays associated with all parent- and child nodes are concatenated to obtain the list of GUI elements that should be repainted.*

4.5 Overview

Figure 7 Illustrates once more how the **evaldepend** function can be used to evaluate the update structure.

In short the idea of the update structure is that with each node a set of dependent GUI-elements can be specified. When the content of the GUI-data is *modified at the level of a specific node* these GUI-elements will be updated.

GUI-data is said to be modified at the level of a specific node if either:

- Direct attributes of the nodes are changed, added, or removed. Direct attributes are the fields of the node;
- Child attributes of the nodes are changed, added, or removed. Child attributes are the fields of children or children’s children of the node;

- A parent or parent's parent of the node is deleted or replaced

Figure 8 shows the diagram that is applicable when no conditional updating is required. In this case the entire interface is repainted every time a change is made in the data model. Comparing Figure 7 and Figure 8 show how the "simple" approach can be extended to the "optimized" approach. Because the optimization process is carried out by appending rather than replacing the current source code, this approach can be part of a software development process where early prototypes already display much of the functionality and look-and-feel of the end product.

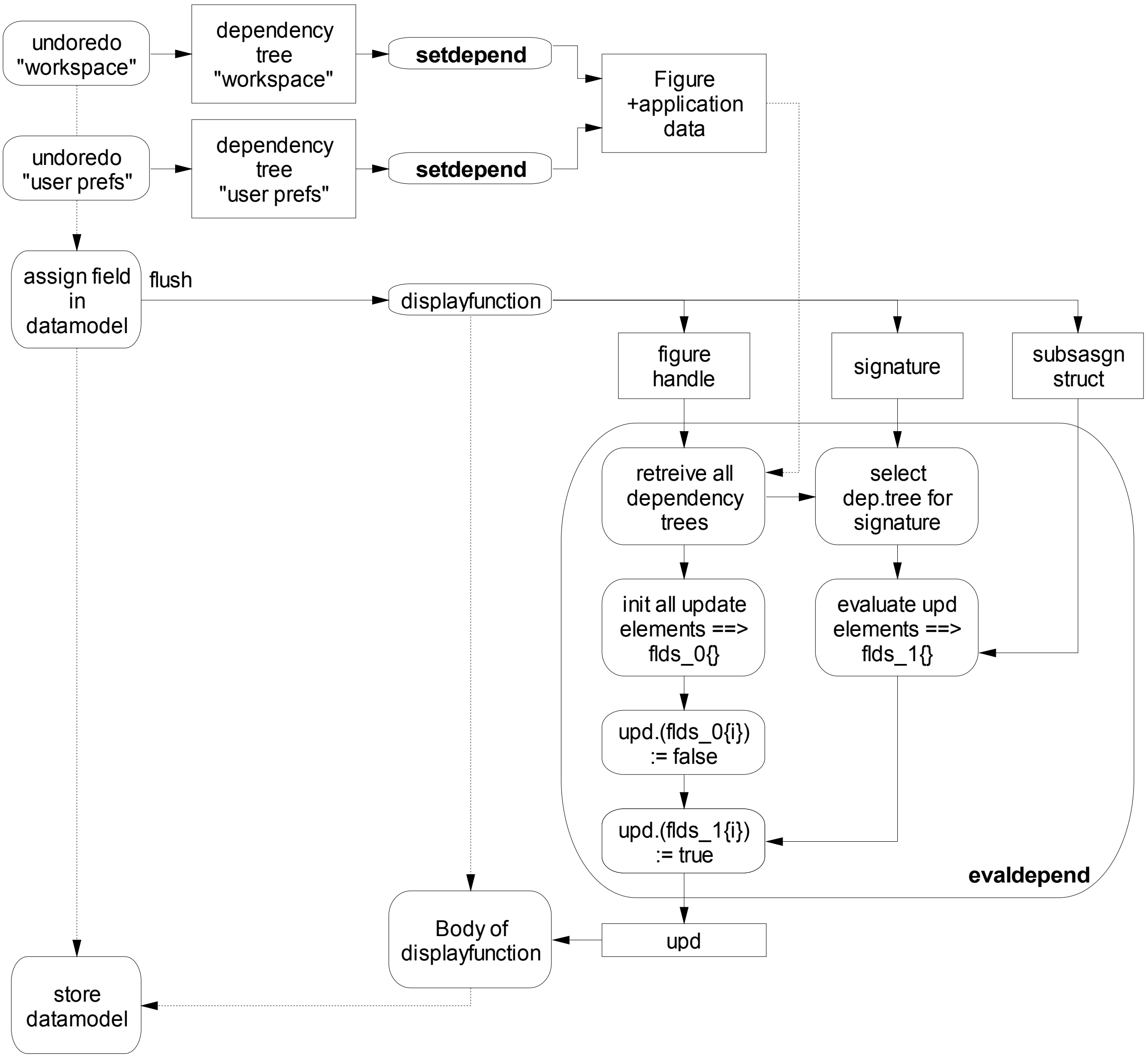


Figure 7: *Illustration of the interplay between setdepend, displayfunction and evaldepend*

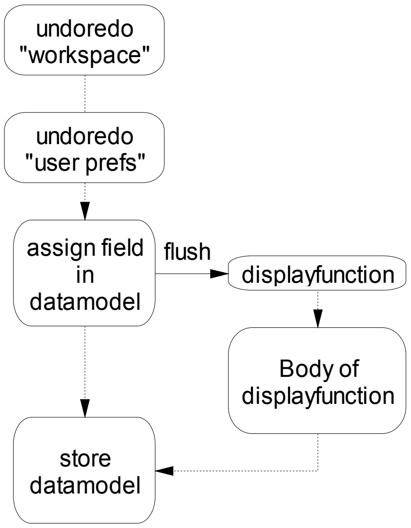


Figure 8: *When no conditional updating is required the code structure simplifies to the diagram above*

5 Reference manual

5.1 Introduction

This reference manual consists of two parts:

- *Methods of the undoredo object*. Where possible, the functionality of the Application Framework is offered through methods of the undoredo object (see section 5.2);
- *Auxiliary functions*. In order to invoke a method of an undoredo object, an undoredo object must be available as a variable. This is not always the case. Therefore at least a few separate functions need to be available (see section 5.3).

5.2 Methods of the undoredo object

5.2.1 cleanupdisk

SUMMARY	Depending on the properties specified upon creation of the **undoredo** object, different files may be associated with this object, such as cache files and backup files. **cleanupdisk** removes these files and should be called when the **undoredo** object is no longer needed.
CALL	`cleanupdisk(db)`
INPUT	db: **undoredo** object.
OUTPUT	This function returns no output arguments.

EXAMPLE	`%Insert somewhere in main body:`
	`set(gcf,'deletef',@deleteFcn);`
	
	`function deleteFcn(HWIN,event)`
	`% deleteFcn - application delete function`
	`% retrieve undoredo object:`
	`db = get_db; %(get_db must be provided)`
	`% remove all files associated with this undoredo object:`
	`cleanupdisk(db);`
	`%Destroy figure:`
	`delete(HWIN);`

5.2.2 closegroup

SUMMARY	In the undo/redo menu, all transactions in a group are presented in one line. The **closegroup** command is used to separate different groups of transactions. Normally the **closegroup** command is not needed as the **store** method closes a group of transactions before storing the **undoredo** object. **closegroup** is needed: • In the specific case where you are performing a series of operations that should appear separately in the undo list, but there is no reason to store the database in between these operations; • When the **store** method is not used for committing transactions in an **undoredo** object.
CALL	`db=closegroup(db)`
INPUT	db: **undoredo** object.

OUTPUT	db: **undoredo** object after update.
SEE ALSO	**store, setlabel**

5.2.3 deletequeue

SUMMARY	Empty the display queue without calling display function. The **undoredo** object keeps track of the items that should be updated by the displayfunction by storing substruct arguments passed on to the **subsasgn** method in a cell array. When the **flush** method is called this queue is passed on to the display function and the queue is made empty. If you are working on an application that uses two **undoredo** objects that can be modified independently, for example one for data and one for user preferences, situations might occur where: • Both **undoredo** objects have a nonempty queue; • Calling flush for one of the **undoredo** objects makes calling the flush method for the other object no longer needed. In this case you can invoke **deletequeue** to tell the other object that it can empty its queue. If you omit this, no real harm will be done, but the next time **flush** is called for this object. Some object will be repainted, causing an undesired user experience.
CALL	`db=deletequeue(db)`
INPUT	db: **undoredo** object.
OUTPUT	db: **undoredo** object after update.

EXAMPLE	`%load workspace from file "fname"`
	`%store current filename in user preferences undoredo object`
	`opt=get_opt;`
	`opt.filename=fname;`
	`%no need for update: next lines will update screen`
	`store(deletequeue(opt));`
	
	`%retrieve and store database:`
	`db=get_db`
	`db.data=load(fname);`
	`store(flush(db));`

SEE ALSO	**flush**

5.2.4 display

SUMMARY	The **undoredo** object is designed so that the analogies with a "normal" Matlab variable are maximized. When the function **display** is invoked on an **undoredo** object, `disp(db.data)` will be called. Also a line with "undoredo object" is displayed to notify the user of the class of the object.
CALL	`display(db)`
INPUT	db: **undoredo** object.
OUTPUT	This function returns no output arguments.
SEE ALSO	**fieldnames, subsasgn, subsref, isfield, isempty**

5.2.5 fieldnames

SUMMARY	The **undoredo** object is designed so that the analogies with a "normal" Matlab variable are maximized. When the function **fieldnames** is invoked on an **undoredo** object, `fieldnames(db.data)` will be called.
CALL	`flds=fieldnames(db)`
INPUT	db: **undoredo** object.
OUTPUT	flds: Cellstring with the fields of the **undoredo** object.

5.2.6 flush

SUMMARY	Perform all paint actions that are required for the transactions since last **flush**. Invoking the **flush** method on an **undoredo** objects causes the displayfunction to be called. This function will be called as follows: `abort=displayfunction(signature,data,queued)` with input: signature: Unique number for each **undoredo** object. The value of the signature is needed by certain auxiliary functions. data: Data contents of the object. queued: Cell array containing substructs. and output: abort: This is an optional output argument for the display function. If the display function sets abort to TRUE, a subsequent call to **store** will have no effect. This implements a mechanism for error checking. You may verify certain conditions in the displayfunction and return abort=TRUE if required conditions are not met.

CALL	`db=flush(db)`
	`db=flush(db,'all')`
	`db=flush(db,extra)`

INPUT	db: **undoredo** object extra: Extra item to be passed on in cell-array 'queued' Typical use: `db=flush(db,'all')`: update all elements Note that the displayfunction that is used should be able to deal with this extra argument.
OUTPUT	db: **undoredo** object after update.
EXAMPLE	Typical use:

	`db=get_db;       %retrieve database`
	`db.field=value; %update database`
	`db=flush(db);    %update display`
	`store(db);       %store database`

SPECIAL CASES	EXAMPLE 1: Paint all screen elements with changes in underlying data:
	`flush(d)`
	
	EXAMPLE 2: Paint all screen elements:
	`flush(db,'all')`
	
	EXAMPLE 3: Mimic change of specific field without actually changing data:
	`flush(guiopt,substruct('.',field));`
	
	EXAMPLE 4: complex argument checking:
	
	The next code fragment shows how complex argument checking can be implemented. If a user enters a number of arguments that are mutually inconsistent. The user should be warned and the previous GUI state must be restored.
	
	`function someCallback`
	`db=get_db;`
	`db=processUserInput;`
	`db=flush(db)`
	`if isempty(db)`
	`    %repaint interface based on previous data`
	`    warndlg(<somewarning>);`
	`    db=get_db;`
	`    db=flush(db,'all'); %you might want to refine here`
	`end`
	`store(db)`

5.2.7 getdata

SUMMARY	In most cases data is retrieved from an object by subscripting, for example:
	`db=undoredo(1:8)`
	`a=db(3)`
	➔ `a=3`
	However there is no subscript that retrieves the complete datastructure. For this purpose the use the getdata method:
	`db=undoredo(data1)`
	`data2=getdata(db)`
	➔ `data2 is an exact copy of data1.`

CALL	`data=getdata(db)`
INPUT	db: **undoredo** object
OUTPUT	Data content of **undoredo** object.
EXAMPLE	If OBJ is an **undoredo** object, the following example shows how to clear the undoredo history of an object: `OBJ=undoredo(data(OBJ));`
NOTE	There is a subtle difference between `data=getdata(db)`and `data=db(:)`. The `(:)` operator always returns a Mx1 vector with `M=numel(db)`. If db contains data with size m x n, **getdata** is needed to retrieve data content and data size.

5.2.8 getsignature

SUMMARY	Signature is an internal field of the **undoredo** object, and hence not visible from outside. The present function returns the value of the signature. **Getsignature** is a specialized function. You may need it if you want to recreate an **undoredo** object with a specific signature that matches the reference made to a previous **undoredo** object, for example when you load new workspace data. However in most cases it will be easier to use the **setdata** command.
CALL	`signature = getsignature(db)`
INPUT	db: **undoredo** object.
OUTPUT	signature of the **undoredo** object (double).
SEE ALSO	**setdata**

5.2.9 iscommitted

SUMMARY	When an **undoredo** object is created or its contents are reassigned using setdata, the status "committed" is set to TRUE. Each command that changes the data content also sets the status "committed" to FALSE. The status "committed" is used in the application program to decide if the user should be asked to save data when the application is closed. This is typically implemented in the closerequest function. When the user saves intermediate results the application should include a statement that sets the committed status to TRUE.
CALL	`committed=iscommitted(db)`
INPUT	db: **undoredo** object.

OUTPUT	committed:
	        Commit status (TRUE or FALSE)
	        Committed=TRUE
	        ➔ All transactions have been committed (saved to disk or stored otherwise)
	        Committed=FALSE
	        ➔ One or more transactions have not been committed.

EXAMPLE	(code example save data)
	
	`ud = getdata(db);`
	`save(fname,'ud');`
	`db=setcommitted(db);`
	`store(db);`
	
	(code example close request function)
	
	`function closereq(hFig,event)`
	`db = get_db;`
	`if isempty(db)||iscommitted(db)`
	`    delete(hFig);`
	`    return`
	`end`
	
	`%Ask and store unsaved data`
	`switch questdlg('Save data?,'Close`
	`application','Yes','No','Cancel','Yes')`
	`    case 'Yes'`
	`        savedata(db);`
	`        delete(hFig);`

	`    case 'No'`
	`        delete(hFig);`
	`        return`
	`    case 'Cancel'`
	`        return;`
	`end`

SEE ALSO	**Setcommitted, subsasgn, setvalue**

5.2.10 isempty

SUMMARY	An overloaded function for **undoredo** objects. It calls isempty on the data component of the **undoredo** object.
CALL	`rc=isempty(db)`
INPUT	db: **undoredo** object.
OUTPUT	rc: Boolean return code.

5.2.11 isemptyqueue

SUMMARY	The **undoredo** object stores a queue that contains all substruct parameters that were passed on to the **subsasgn** method (see below) since the last call to **flush**. This queue will be passed on to the display function as its third argument, and allows this function to decide which screen elements to update. A call to the **flush** method will empty the queue. **isemptyqueue** is a specialized utility that tells whether or not the queue is empty.
CALL	`rc=isemptyqueue(db)`
INPUT	db: **undoredo** object.
OUTPUT	rc: Boolean return code.

5.2.12 isfield

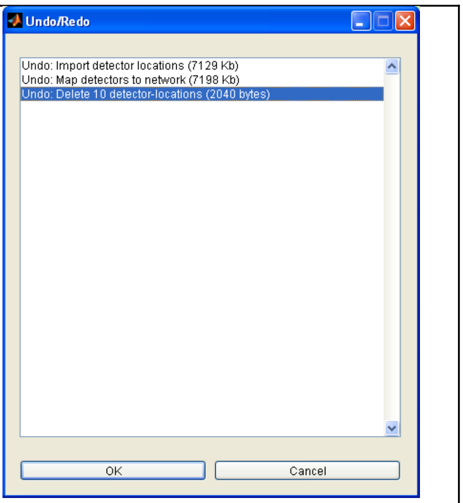
SUMMARY	An overloaded function for **undoredo** objects. This method calls **isfield** on the data component of the **undoredo** object.
CALL	`rc=isfield(db,fld)`
INPUT	db: **undoredo** object.
OUTPUT	rc: Boolean return code.

5.2.13 isopen

SUMMARY	A specialized utility that normally should not be needed. It allows one to check if modifications have been made, but not yet stored, since the last invocation of the **store** method. For further explanation and backgrounds we refer to the summary of the **label** method.
CALL	`rc=isopen(db)`
INPUT	db: **undoredo** object.

OUTPUT	rc: Boolean return code rc=1: The group is open. This means that since the last call to **store** at least one modification was made. The next modification will be added to the present group. rc=0: The group is closed. No modifications have been made since the last call to **store**. The next modification will initialize a new group.

5.2.14 label

SUMMARY	Attach a label to a group of modifications applied to an **undoredo** object. This label will appear in the list presented in the undo and redo menu, see the figure on the right. Apart from displaying a string in the undo/redo dialog the label has no function. Specifying the label is optional. The label is attached to a *group* of modifications. *What is a group?* A group is initialized with a modification. Each next modification will be added to the present group. A group is closed by invoking the **store** method. **Undo** and **redo** commands will always on all elements of a group simultaneously. It is not possible to **undo** one element of a group and leave the other elements unaffected.  Undo: Import detector locations (7129 Kb) Undo: Map detectors to network (7198 Kb) Undo: Delete 10 detector-locations (2040 bytes)
CALL	`db=label(db,labelstr)`
INPUT	db: **undoredo** object. labelstr: Label to attach to a group of operations.
OUTPUT	db: **undoredo** object after update.
SEE ALSO	**undomenu, isopen**

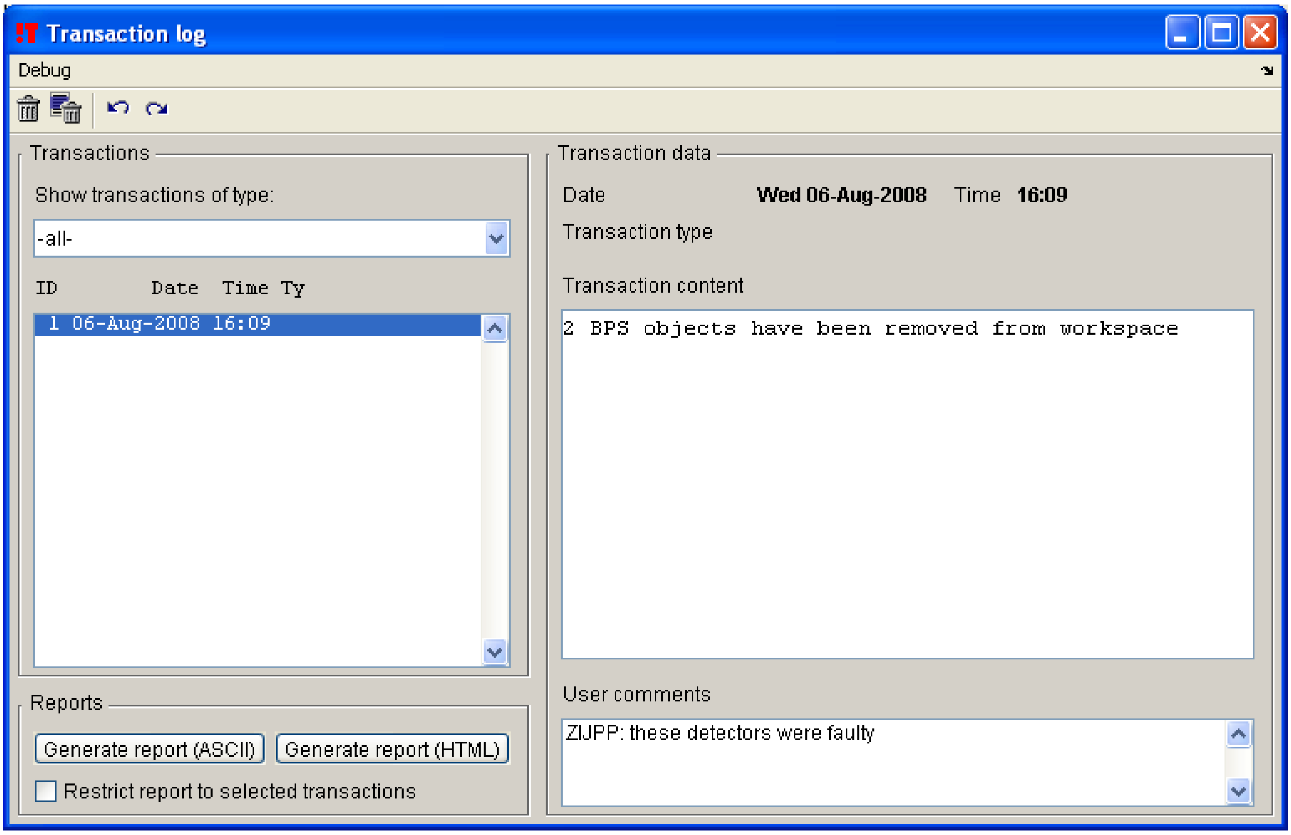
5.2.15 logbookentry

SUMMARY	Applications that involve inspecting or manipulating data often have a need for a logbook that stores information about the data-handling process and has room for user comments. For this purpose the Application Framework contains the logbook. You can add information to the logbook of your application by using the **logbookentry** command. The method **logbookgui** can be used to start the logbook GUI as shown in Figure 9.
CALL	`db = logbookentry(db,content,type,undolabel,comment)`

INPUT	db: undoredo object (to be updated). content: Value for content property (CELL or CHAR array). type Optional argument (defaults to empty), value for type property. undolabel: Optional argument (defaults to empty), label for undo menu. Include this if **logbookentry** defines the only element of an update group. comment: Optional argument (defaults to empty), value for comment property.
OUTPUT	db: **undoredo** object after update The field "transaction" is uninitialized or appended with the following data structure: transaction +----date (double) +----content (char array) +----type (char array) +----comment (char)

EXAMPLE	`data.value=1:10;`
	`db=undoredo(data);`
	`data.value(5)=50;`
	`db=logbookentry(db,'element 5 has been updated','updated');`
	`disp(getdata(db));`

SEE ALSO	**transact_gui, logbookgui**



-all-

1 06-Aug-2008 16:09

Date Wed 06-Aug-2008 Time 16:09

Transaction type

Transaction content

```
2 BPS objects have been removed from workspace
```

User comments

ZIJPP: these detectors were faulty

Figure 9: *Logbook GUI. This logbook contains one report and one manual edit.*

5.2.16 redo

SUMMARY	The **redo** command rolls back the last modification to an **undoredo** object. When done, the **flush** method is invoked.
CALL	`db=redo(db,N)`
INPUT	db: **undoredo** object. N: Number of actions to redo (defaults to 1).
OUTPUT	db: **undoredo** object after update.
SEE ALSO	**undo**

5.2.17 setcommitted

SUMMARY	This function works together with **iscommitted**. See manual page of **iscommitted** for summary and examples.

CALL	`db=setcommitted(db)`
	`db=setcommitted(db,committed)`

INPUT	db: **undoredo** object committed: TRUE or FALSE. (Defaults to TRUE). The committed property of the **undoredo** object will be overwritten with this value.
OUTPUT	db: **undoredo** object after update.
EXAMPLE	See **iscommitted** for an example.
SEE ALSO	**iscommitted**

5.2.18 setdata

SUMMARY	The **subsasgn** method provides no way to replace the datacontent of an **undoredo** object with a new Matlab variable. This method does the job. The extra argument may be used to indicate whether or not the undo history should be cleared.

CALL	`db=setdata(db,data)`
	`db=setdata(db,data,reset)`

INPUT	db: **undoredo** object. data: New data for **undoredo** object. reset: (Optional, defaults to TRUE), if TRUE reset the undo history of the object.
OUTPUT	db: **undoredo** object after update.

EXAMPLE	This function is particularly useful when the database is replaced when a new workspace is loaded from file.
	
	`db=get_db;`
	`if ~iscommitted`
	`  -provide code that prompts user to save data-`
	`end`
	`db=setdata(load(fname),true);`
	`%remember to paint all objects`

	`store(flush(db,'all'));`
SEE ALSO	**getdata**, **subsasgn**

5.2.19 setdepend

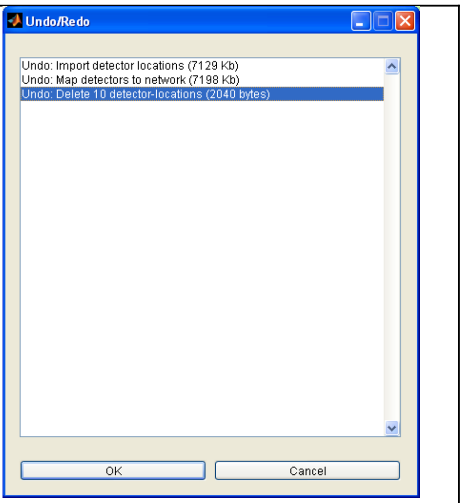
SUMMARY	A dependency tree is used by **evaldepend** to derive the so-called update structure using the substruct arguments used in one or more assignments to an **undoredo** object. A dependency tree is specified in a user defined function that returns a structure that resembles the database of an application. At each node of this structure a field "updobj" may be added. This field should contain a cell array with the name or names of the update actions that are required when an assignment is made that effects this node or any of its children. See the example for an illustration.
CALL	`setdepend(db,HWIN,deptree)`
INPUT	HWIN: Figure handle. db: **undoredo** object. deptree: Dependency tree.
OUTPUT	This function returns no output arguments, but registers the dependency tree in the application data of the figure.

EXAMPLE	`function example`
	`%initialize`
	`data.a=1;`
	`data.b.c=2;`
	`%create application figure`
	`HWIN=figure;`
	`%define database`
	`db=undoredo(data,'disp',@view,'storeh',HWIN,'storef','userdata');`
	
	`deptree.a.updobj={'update_a'};`
	`deptree.b.updobj={'update_b'};`
	`deptree.b.c.updobj={'update_c'};`
	`deptree.b.d.updobj={'update_d'};`
	`setdepend(HWIN,db,deptree); %register dependency tree`
	`%end of initialize`
	
	`%do some assignments and view what happens, make sure the function`
	`%"view" (see below) is available`
	`db.b.c=1;`
	`db=flush(db);`
	`%      ==>upd =`
	`%      update_a: 0`
	`%      update_b: 1`
	`%      update_c: 1`
	`%      update_d: 0`
	`db.b=1;`
	`db=flush(db);`
	`%      ==>upd =`
	`%      update_a: 0`
	`%      update_b: 1`
	`%      update_c: 1`
	`%      update_d: 1`
	`db.a=1;`
	`db=flush(db);`
	`%      ==>upd =`

	`%     update_a: 1`
	`%     update_b: 0`
	`%     update_c: 0`
	`%     update_d: 0`
	
	`function view(signature,S,ind)`
	`upd=evaldepend(gcf,ind,signature)`

SEE ALSO	**evaldepend**

5.2.20 setlabel

SUMMARY	Attach a label to a group of modifications applied to an **undoredo** object. This label will appear if the list presented in the undo and redo menu, see the figure on the right. Apart from displaying a string in the undo/redo dialog the label has no function. Specifying the label is optional. The label is attached to a *group* of modifications. *What is a group?* A group is initialized with a modification. Each next modification will be added to the present group. A group is closed by invoking the **store** method. **Undo** and **redo** commands will always work on all elements of a group simultaneously. It is not possible to undo one element of a group and leave the other elements unaffected. *What happens if no call is made to* **setlabel***?* Calling **setlabel** is optional. If no call is made to **setlabel**, the label will remain empty. *What happens if multiple calls are made to* **setlabel***?* When **setlabel** is called more than once between initializing and closing a transaction the last call will override the earlier one.  Undo: Import detector locations (7129 Kb) Undo: Map detectors to network (7198 Kb) Undo: Delete 10 detector-locations (2040 bytes)
CALL	`db=label(db,labelstr)`
INPUT	db: **undoredo** object. labelstr: Label to attach to a group of operations.
OUTPUT	db: **undoredo** object after update.
SEE ALSO	**undomenu, isopen, closegroup**

5.2.21 store

SUMMARY	When an **undoredo** object is created the properties "storehandle" and "storefield" may be specified. This allows an **undoredo** object to store itself. The **store** operator is similar to the commit action known in databases. Before an **undoredo** object is stored the group of transactions is closed, so that the next change will initialize a

	new group.
CALL	`store(db)`
INPUT	db: **undoredo** object.
OUTPUT	This function returns no output arguments, but updates the userdata or applicationdata of the handle specified in the **undoredo** object.
EXAMPLE	Usually store is called in combination with **flush**: `store(flush(db))`
SEE ALSO	**closegroup, flush**

5.2.22 subsasgn

SUMMARY	An overloaded function for **undoredo** objects. It calls **subsagn** on the data component of the **undoredo** object.

CALL	(examples)
	`db.a=value`
	`db.a(2)=value`
	`db.b=[]`

INPUT	db: **undoredo** object. value: Matlab variable.
OUTPUT	db: Updated **undoredo** object.

5.2.23 subsref

SUMMARY	An overloaded function for **undoredo** objects. It calls **subsref** on the data component of the **undoredo** object.

CALL	(examples)
	`value=db.a`
	`value=db.a(2)`

INPUT	db: **undoredo** object.
OUTPUT	value: Matlab variable.

5.2.24 undo

SUMMARY	The **undo** command rolls back the last modification to an **undoredo** object. When done, the **flush** method is invoked.
CALL	`db=undo(db,N)`
INPUT	db: **undoredo** object. N: Number of actions to undo (defaults to 1).
OUTPUT	db: **undoredo** object after update.
SEE ALSO	**redo**

5.2.25 undoredo

SUMMARY	Constructor of the **undoredo** object. This function transforms a Matlab variable into an **undoredo** object.		
CALL	`db=undoredo(data,<property1>,<value1>,<property2>,<value2>,...)`		
INPUT	The first input parameter is the Matlab variable that is encapsulated in the **undoredo** object. All other parameters are passed as property-value pairs. Properties are not case sensitive and need not be fully named, as long as the property is uniquely identified.		
	property [class]	**default value**	**usage**

	displayfunction [function]	empty	This function will be called every time the method **flush** is invoked on the **undoredo** object.
			The function call looks like
			`abort=displayfunction(...`
			`    signature, data, queued)`
			with input:
			signature:
			Unique number for each **undoredo** object. The value of signature is needed by certain auxiliary functions.
			data:
			Data contents of the **undoredo** object.
			queued:
			Cell array containing substructs.
			and output:
			abort:
			This is an optional output argument for the display function. If the display function sets abort to TRUE, a subsequent call to store will have no effect. This implements a mechanism for error checking. You may verify certain conditions in the displayfunction and return abort=TRUE if required conditions are not met.

	signature [double]	now()	Approximate time of object creation. Used as a reference to the **undoredo** object (without requiring to pass on the full object and its data). Signature is passed on to the objects displayfunction. Normally, this property is not specified and the **undoredo** constructor assigns this field. see also: **setdata**
	mode [char]	'simple' {'memory'} 'cached'	The mode parameter determines how the "undo" and "redo" functionality will be implemented. “simple”: Use this option is no undo is required. “memory”:

			Undo info stored in memory. Use this option if no massive datasets are needed. “cached”: Undo info cached to disk if needed. This option is only needed for very large datasets.
	cachefile [char]	'urcache'	(applies only if mode==”cached”) The cachefile names will start with this parameter. Administrative info will be appended. To prevent a mix-up between cache files and other files it is recommended to avoid names that might be in use by other applications. There is no real need to specify the cachefile parameter, unless you run multiple cached applications from the same directory.
	maxbytes [integer]	64.000.000	Maximum number of bytes stored in memory before saving to disk.
	storehandle [double]	required parameter	Handle of GUI object with which **undoredo** object is saved.

	storefield
	[char]
		required parameter	Setting storefield==”userdata” causes **undoredo** data to be saved using:
			`set(storehandle,'userdata',obj)`
			setting storefield~=”userdata” causes **undoredo** data to be saved using:
			`setappdata(storehandle,storefield,obj)`

	backupfile [char]	`''`	Use filename as base for the autobackup file. If this parameter is not specified or left empty, no automatic backups will be created. The backup file will be updated periodically (see parameter "backupinterval"). When using automatic backup, you must delete the backup files when the application closes.
	backupinterval [double]	10/1440	This parameter is passed as a **datenum** value (1 equals one day). Every time a modification to an **undoredo** object is made, the internal parameter "timeoflastbackup" is checked and updated.
OUTPUT	db	**undoredo** object.	

5.3 Auxiliary functions

5.3.1 dispupd

SUMMARY	For debugging purposes it may be useful to include a line with `dispupd(upd)` in the display function. When the display function is called a list of items that will be updated is printed on the console to facilitate troubleshooting. In a deployed environment no debug information will be printed.
CALL	`dispupd(upd)`
INPUT	upd: The update structure, as returned from **evaldepend**.
OUTPUT	This function returns no output arguments but displays information in the command window.

EXAMPLE	`function view(signature,data,ind)`
	`upd=evaldepend(gcf,ind,signature);`

	`dispupd(upd);`
SEE ALSO	**evaldepend**

5.3.2 evaldepend

SUMMARY	This function evaluates the update structure for the combination of: • One or more **undoredo** objects registered with **setdepend**; • A figure. Prior to calling this function the dependency tree must be specified using the **setdepend** command. *Why does this function rely on a signature instead of on an* **undoredo** *object?* The dependency tree is defined and registered separately for each *combination* of **undoredo** object and figure. Hence, one **undoredo** object may be registered with more than one figure, each time with a different dependency tree. This is because different figures contain different updatable elements. When the **undoredo** object is registered with a figure, not the full object is stored as a reference, but only its signature. The **undoredo** signature uniquely defines each **undoredo** object. Using the full object as a reference would imply undesired redundancy.
CALL	`upd = evaldepend(HWIN, ind, signature)`
INPUT	HWIN: Figure handle. ind: Subscripts applied to modify object data. signature: Signature of modified **undoredo** object.
OUTPUT	upd: A structure that contains the screen elements that should or should not be updated: upd.property=0 ➔ do not update screen element. upd.property=1 ➔ update screen element.

EXAMPLE	The code below provides a template for usage of dependency trees
	
	`%Include in the main body of the application:`
	`db=undoredo(initdb,'disp',@dispdata);`
	`setdepend(HWIN, db, data2applic);`
	`opt=undoredo(initopt,'disp',@dispsettings);`
	`setdepend(HWIN, opt, settings2applic);`
	
	`function s=initdb`
	    `-user defined function-`
	`function s=initopt`
	    `-user defined function-`
	`function db=get_db`
	    `-user defined function-`
	`function opt=get_opt;`
	    `-user defined function-`
	
	`%the next function is called when application data change:`
	`function dispdata(signature,db,ind)`
	`upd = evaldepend(HWIN, ind, signature)`
	`opt=get_opt;`
	`view(db,opt,upd);`

	%the next function is called when user preferences change:
	function dispsettings(signature,opt,ind)
	upd = evaldepend(HWIN, ind, signature)
	db=get_db;
	view(db,opt,upd);
	
	function view(db,opt,upd)
	-user defined function-
	if upd.element1
	    -user defined action-
	end
	if upd.element2
	    -user defined action-
	end

SEE ALSO	**evaldepend**

5.3.3 getpostponed

SUMMARY	The methods **getpostponed** and **setpostponed** are useful in situations where the display depends both on workspace data and user-preferences. A typical situation that might occur is that the visibility of a specific panel, say a panel that displays a complex graph, is controlled by a parameter that is part of the user preferences and the content of this complex graph depends on a dataset that is part of the workspace data. Suppose that this dataset changes. This implies that the graph must be updated. If the panel that holds the graph is visible, everything is straightforward: the graph will be updated and the display and the database will be consistent again. Now, suppose the user has disabled the graph. We might still update the axes so that once the user enables the graph; it will be up to date immediately. However, this is a waste of resources: the user might never enable the graph. Another option is to force painting the graph each time the panel is made visible. This is also a waste of resources in some cases: hiding and showing the frame will now take a long time: the graph will be updated every time the visibility state of the panel is toggled, even if the graph was already up to date. **Setpostponed** and **getpostponed** provide a workaround for these cases. When a graph needs to be updated but the panel that displays it is hidden, one may call **setpostponed** instead of actually drawing the graph. The next time the displayfunction is called; use **getpostponed** to detect any postponed drawing actions if the visibility of the panel has changed otherwise **setpostponed** will be called again.
CALL	upd=getpostponed(upd,HWIN,NAME)
INPUT	upd: Update structure (see also **evaldepend**). HWIN: Handle of window for which the update is being postponed; NAME: Storage name (defaults to "ppupdate");
OUTPUT	upd: Update structure, expanded with postponed actions.

EXAMPLE	function display(signature,data,ind)
	%retrieve user preferences:
	opt=get_opt;
	%check which elements need an update based on last modification:
	upd=evaldepend(gcf,ind,signature);
	%check which element may need updated based on
	%earlier modifications:
	upd=getpostponed(upd,gcf);

```
%reset list of postponed update elements
postponed={};
if upd.showcomplexdata
  if opt.panelisvisible
    <paint complex graph>
  else
    %action "showcomplexdata" cannot be completed now
    postponed{end+1}='showcomplexdata';
  end
end
%Register list of postponed update elements
setpostponed(postponed,gcf);
```

SEE ALSO	**setpostponed**

5.3.4 isregistered

SUMMARY	Returns TRUE if undoredo object is registered on figure.
CALL	`b = isregistered(obj, fig)`
INPUT	props: Cell array containing list of postponed actions. HWIN: Handle of window for which the update is being postponed. NAME: Storage name (defaults to "ppupdate").
OUTPUT	This function returns no output arguments, but changes the application data "ppupdate" of the figure with handle HWIN.
SEE ALSO	**evaldepend**

5.3.5 setpostponed

SUMMARY	See **getpostponed**.
CALL	`setpostponed(props,HWIN,NAME)`
INPUT	props: Cell array containing list of postponed actions. HWIN: Handle of window for which the update is being postponed. NAME: Storage name (defaults to "ppupdate").
OUTPUT	This function returns no output arguments.
EXAMPLE	See **getpostponed**.
SEE ALSO	**getpostponed**

5.3.6 logbookgui

SUMMARY	The auxiliary function **logbookgui** starts the logbook GUI as shown in Figure 9. Usually this function is defined as a callback of a toolbar button.
CALL	`logbookgui(obj,event,fp_getdata,C)`
INPUT	obj,event: Only for internal use. These arguments are passed by Matlab when an object callback is specified as `{@logbookgui,fp_getdata,C}`

	fp_getdata:
	Function pointer to function that returns **undoredo** object. This can be a three-line function like:
	`function db=getdata`
	`HWIN=findobj('tag','MAINWIN');`
	`db=get(WIN,'userdata');`
	C:
	Structure with GUI constants (colors, fontsize, etc.). If not specified, default settings are used.

OUTPUT	This function returns no output arguments but creates a GUI.
SEE ALSO	**logbookentry**

EXAMPLE	The following code adds a button to the toolbar that opens the logbook:
	`uipushtool('tooltip','Inspect or edit logfile',...`
	`    'clicked',{@logbookgui,@get_db,C});`

5.3.7 undomenu

SUMMARY	Execute undo, redo of undo-menu.
CALL	`undomenu(obj,event,operation,fp_getdata,HWIN)` NOTE: This is not a method of **undoredo** objects, but a general accessible function.

INPUT	obj,event:
	Standard Matlab callback arguments. These arguments are not used in the present function.
	operation:
	Type of operation required.
	operation==1 ➔ undo.
	operation==2 ➔ redo.
	operation==3 ➔ multiple undo/redo. A popuplist appears that allows users to choose to which state the application should reverse.
	operation==4 ➔ reset undo/redo history.
	fp_getdata:
	Function pointer to user-specified function that returns database structure. This can be a three-line function like:
	
	`function ud=getdata`
	`global MAINWIN %handle of application's main window`
	`ud=get(MAINWIN,'userdata');`
	
	HWIN:
	Input argument for fp_getdata (usually a figure handle).

OUTPUT	This function returns no output arguments.

SEE ALSO	**undo, redo**

5.3.8 ur_getopt

SUMMARY

User preferences are settings that apply to the appearance of a specific application. When an application is closed and opened later, users typically expect that the application reappears with identical settings. To accomplish this, the user preferences should be saved when the application closes and loaded again when the application is started. Saving the data can best be done in the application's main figure's deletefunction. Loading the data typically is done in a function that by convention has the name "initOpt" (but any other name is allowed). This function initializes the data structure that represents the user preferences. This is done in three steps:

- Create a structure that contains the factory defaults. This is to make sure that no errors will occur if the application is started for the first time;
- Load the user preferences as saved when the application was closed last time (see function template "deletef"), and overwrite the factory defaults with the values that are loaded from file. This is done by the function **ur_getopt**;
- Set any values that are specific for the current session. For example, you may store object handles in the user-preference structure.

CALL

```
opt=ur_getopt(defopt,OPTFILE,varname)
```

INPUT

defopt:
Default options (current function overwrites these).

OPTFILE:
Binary file in which options have been saved earlier.

varname:
Variable name in which options are stored (defaults to "opt”).

OUTPUT

opt:
Options structure in which data from defopt and OPTFILE are combined

EXAMPLE

```
function initOpt
%assign factoryDefaults (function not provided in template)
opt=factoryDefaults;
%apply saved user preferences
opt=ur_getopt(opt,initOptFile);
%override options where applicable
opt.currentFile='untitled.mat';
```

6 Advanced topics

6.1 Design decisions

The state of an application is made up by a multitude of variables; data on file en handle graphic objects. A user changes the state by issuing commands.

When a new application is implemented a number of design decisions must be made with regard to how the state will be stored:

- Use the Application Framework or not? Very simple applications may not need the framework. However if it is likely that the current -simple- application will be extended in the future, it might be a good idea to use the framework anyway;
- Implement a single undoredo object that holds workspace data and user preferences or two separate undoredo objects. Adding an extra undoredo object for the user prefernces requires extra work but provides a better user experience ;
- Which data need to be stored where? The state of an interface can be stored in many ways, for example:
  - In one or more undoredo objects;
  - By reference to data that is stored in a file or in a database;
  - In global Matlab variables;
  - As userdata or applicationdata of handle graphic objects;
  - As other properties of handle graphic objects, such as 'String" or 'Checked' and 'Value'.

In particular the poperties of graphic applications need attention if you are designing a GUI that allows "undo" and "redo" commands. In many applications these properties make up an important part of the storage. For example if a user enters a filename in an edit box, there is no need to store this variable in a redundant variable.

When applying the Application Framework, this is typically what happens. The reason for this is that the field should also display correctlty after undo, redo or initialization.

You may not want to extend this to all properties. For example if you application contains a list, you may not want to store the selected items in this list redundantlty becaus ther is no point is keeping track of the lists selections in an undoredo object.
When the user clicks in the list, you still need to call the displayfunction, for example to display the selected listietems in some manner. This needs to be done without making a change to the object. A typical trick you may apply is to extend the dependency tree of the user preferences with a dummy field an make modification to this field.

For example:

```
%include this line in dependency tree
upd.dummy.updobj = {'plotmarker'}

%include these lines in list callback:
opt=get_opt;
opt.dummy=[];
%paint interface,catch output to suppress warning
unusedout=flush(opt);
```