Creación de Interfaces Gráficas de Usuario (GUI) con MatLab

Gonzalo Fernández de Córdoba Martos
Salamanca, Septiembre de 2007.

1. Objetivo del curso

El objetivo de este curso es aprender a realizar interfaces gráficas de usuario, a las que nos referiremos como GUI-s que es como se normalmente se conocen.
El lenguaje más habitual para crear GUI-s es Java, ya que tiene la enorme ventaja de funcionar en cualquier máquina, sin embargo Java resulta muy lenta para hacer cálculos eficientemente, y es aquí donde MatLab es más poderoso. Por otro lado, las GUI-s creadas con MatLab pueden ser entregadas al ordenador del cliente (quien posiblemente no tenga más que un navegador) y ser ejecutadas en el ordenador de quien creó la interfaz en MatLab (y que por supuesto tiene un MatLab funcionando), de modo que la ventaja relativa de Java esta parcialmente ofertada también por MatLab.
Las GUI-s son herramientas muy útiles para entregar aplicaciones a aquellas personas que no saben lo suficiente de programación y que quieren beneficiarse de las ventajas de un programa.

2. Organización del material.

El curso está estructurado de la siguiente manera: i) Diseño de una GUI, ii) Creación de una GUI sencilla y iii) Analizando una GUI más complicada.

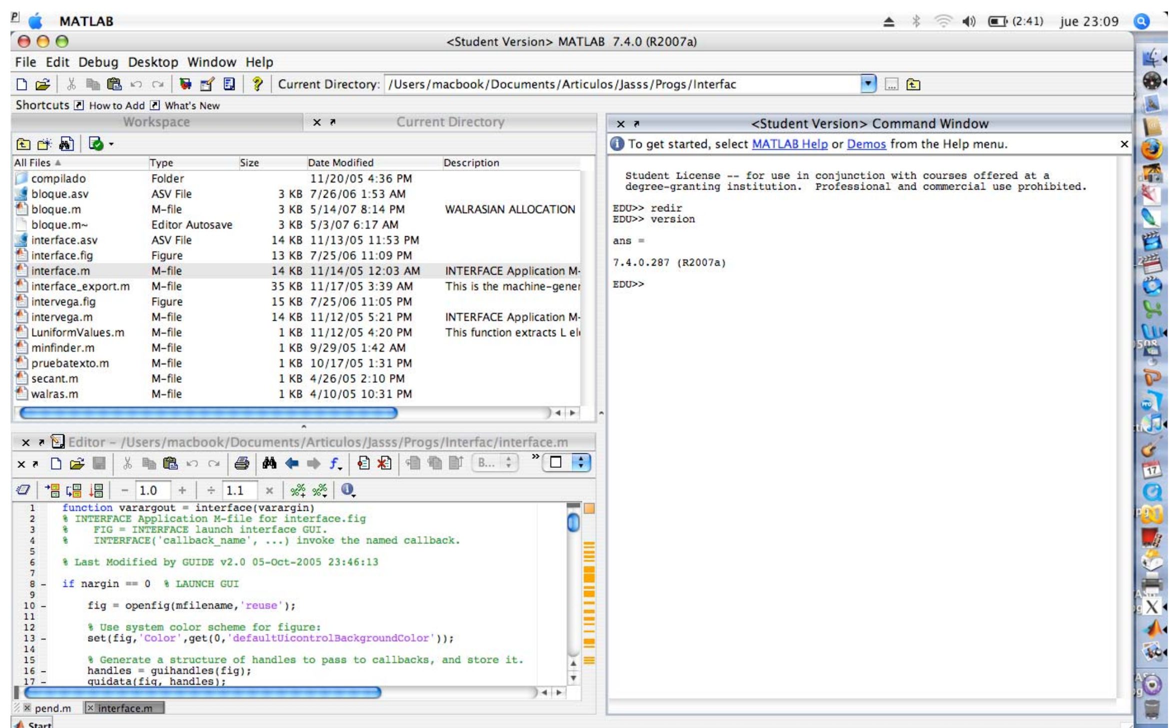
2.1 Diseño de una GUI.

Antes de empezar a programar es imprescindible hablar con el usuario final de la GUI . Es importantísimo entender cuáles son las necesidades exactas que tienen que ser cubiertas por la aplicación. Para ello es necesario entender el tipo de datos y variables que son introducidas por el usuario, así como las excepciones que puedan producirse, los casos que ocurren pocas veces pero que hay que tener en cuenta, etc. También es necesario saber cómo quiere el cliente que se presenten los datos; si se necesitan gráficos o tablas que salgan por impresora, o cómo se guardan los resultados, dónde se guardan y en qué formato lo hacen. La parte del diseño es, con mucha diferencia, la más importante desde el punto de vista del usuario y por tanto también lo es desde el punto de vista empresarial.
Para diseñar correctamente una GUI, lo mejor es hacerlo con papel y lápiz. Presentar un boceto al cliente y mejorarlo con él es la mejor opción. De esta manera se consigue que no haya sorpresas y evita que después de haber realizado un montón de trabajo luego haya que tirarlo a la basura y que encarece mucho los proyectos, y además se consigue que el cliente se implique en el proyecto poniendo su talento y sus preferencias en la herramienta que al final usará él mismo.
Las GUI-s tienen que hacerse de modo que los botones estén donde la gente espera que estén. Si nuestra GUI tiene varias páginas distintas y en cada una de ellas hay un botón que dice “Guardar” es conveniente que ese botón esté localizado en el mismo sitio siempre. Todo esto parece ser de un sentido tan

común que parece innecesario hacer notar que el papel y el lápiz son la mejor herramienta, sin embargo al hacer GUI-s sólo el sentido común tiene algún sentido.
Una vez que tenemos claro qué objetos tendrá la GUI, gráficos, textos, radio buttons, check boxes, edición de texto, entrada de valores, lectura de matrices, etc, y una vez que tengamos claro de qué forma aparecerán en la interfaz (el *layout*) es necesario hacer un programa de tipo *script* que tenga la misma funcionalidad que la GUI que queremos programar. Antes de incorporar el programa a la GUI, es necesario hacer todo tipo de pruebas con él hasta estar completamente seguros de que el programa que vamos a incorporar en la GUI es el programa que queremos. Para hacer las necesarias pruebas lo mejor es hacerlas sobre un *script* y no directamente sobre la GUI.
Una vez que tengamos el *script* guardado podremos incorporar los distintos trozos del *script* en la GUI, de modo que al hacer las pruebas sobre la GUI podamos contrastar los resultados con los que obtenemos del *script*.
Una vez hayamos acabado con los tests sobre la GUI definitiva y estemos completamente seguros de su correcto funcionamiento, la GUI puede ser entregada al cliente.

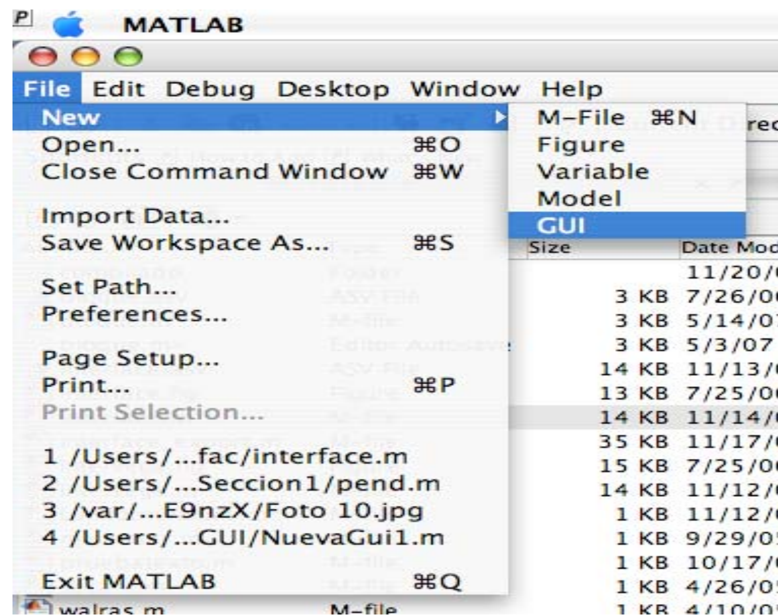
2.2 Creación de una GUI sencilla.

Para crear una GUI sencilla primero tenemos que llamar a la ventana de comandos de MatLab. Esto se hace picando sobre el icono de MatLab, que al abrirse nos mostrará la siguiente ventana:

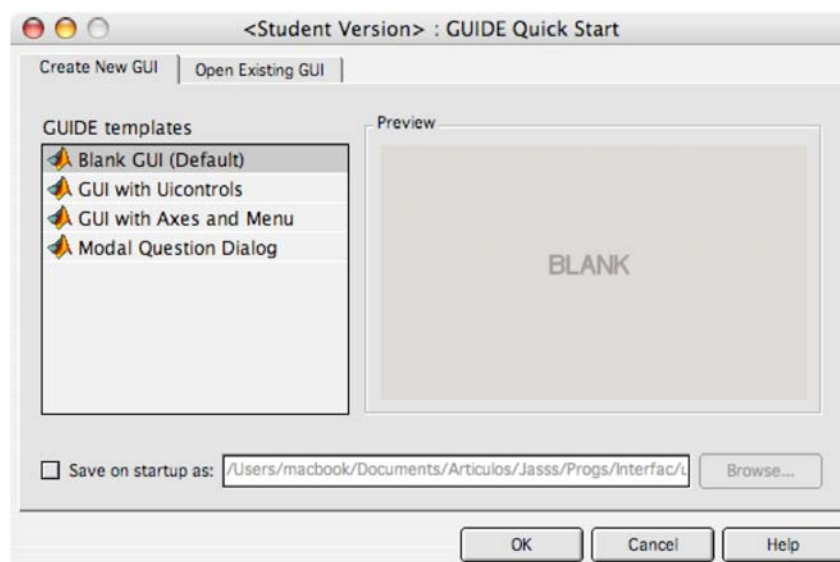


Como se puede ver en la ventana de comando la versión que estoy usando es la nueva versión de MatLab para estudiantes 7.4.0.287 (R2007a). Es muy similar a las últimas versiones para empresas y únicamente tiene restricciones legales que impiden usar esta versión con fines empresariales.

La ventana de comandos que aparece aquí tiene el comando **redir**, que es un programita que me he hecho para que me lleve a directorios seleccionados. Los que no se hayan hecho un programa como ese deben acudir al desplegable que aparece sobre la ventana de comandos como **Current Directory** y llevar a la ventana de comandos al directorio donde desees trabajar.
Para empezar a crear una GUI vamos a File-> New->GUI como en:

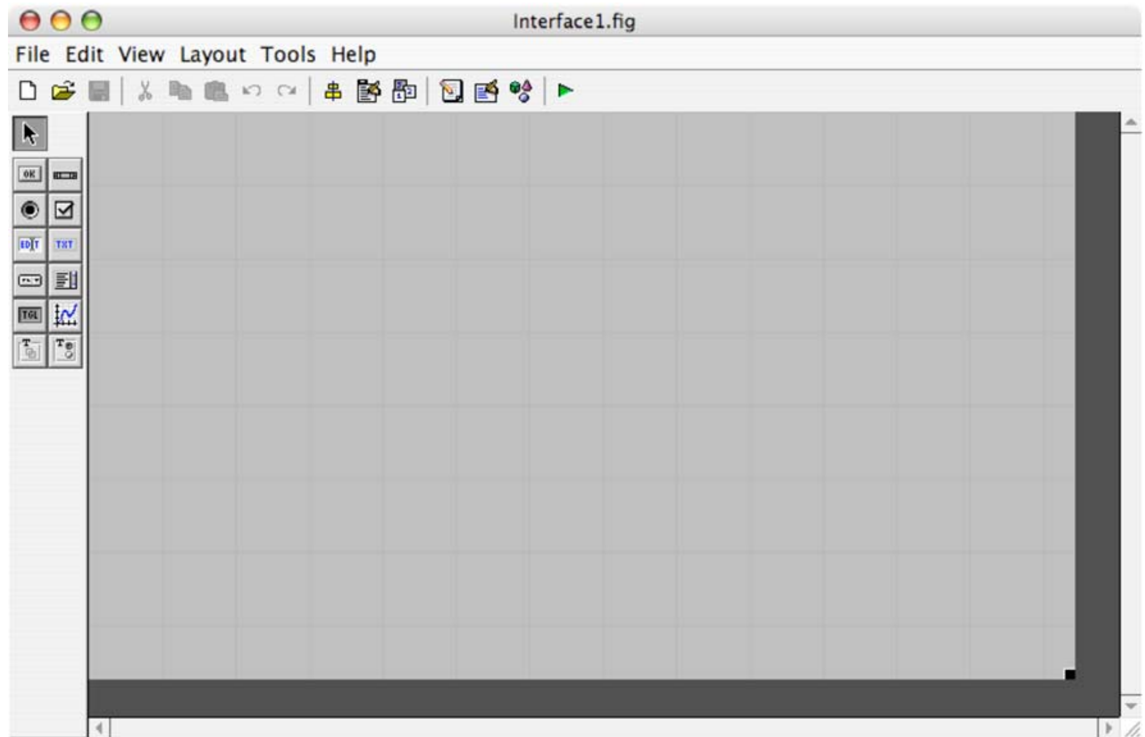


Al seleccionar que queremos una GUI se nos abre una ventana de diálogo en la que tenemos unas serie de opciones.



Podemos abrir una GUI que ya exista, lo cual haremos después de haber guardado nuestra primera GUI o comenzar con una GUI en blanco. Las otras ofrecen una plantilla en la que ya existen algunos controles (Uicontrols), o en la que ya hay incorporado un gráfico y un menú y finalmente incorporar un ventana de diálogo. No obstante todas esas cosas las haremos en estas secciones, de modo que elegimos tener una GUI en blanco.

Abajo, en la ventana de diálogo, observamos un *Check Box* que dice *Save on startup as:* y una ventana que te permite seleccionar un nuevo directorio. Mi sugerencia es marcar la casilla para activar la opción *Browse* y meter el nombre de una interfaz en un directorio en el que no haya nada. El nombre que yo he elegido es **interface1** en un directorio nuevo y obtengo lo siguiente:



Como puedes ver, el nombre de la interfaz es el elegido, y por supuesto está en el directorio que yo he seleccionado. De no haberlo guardado, MatLab habría titulado provisionalmente a la interfaz como **untitled.fig** y la habría guardado en el directorio donde se hallaba la ventana de comandos.
La cuestión más importante es que al guardar la interfaz en el mismo comienzo nos encontramos con unos archivos nuevos en nuestro directorio elegido. Un archivo es **interface1.fig** y el otro es **interface1.m**. Antes de comenzar a manipularlos es conveniente echarles un vistazo para ver las diferencias que se van a ir produciendo a medida que vayamos introduciendo elementos dentro de la interfaz.

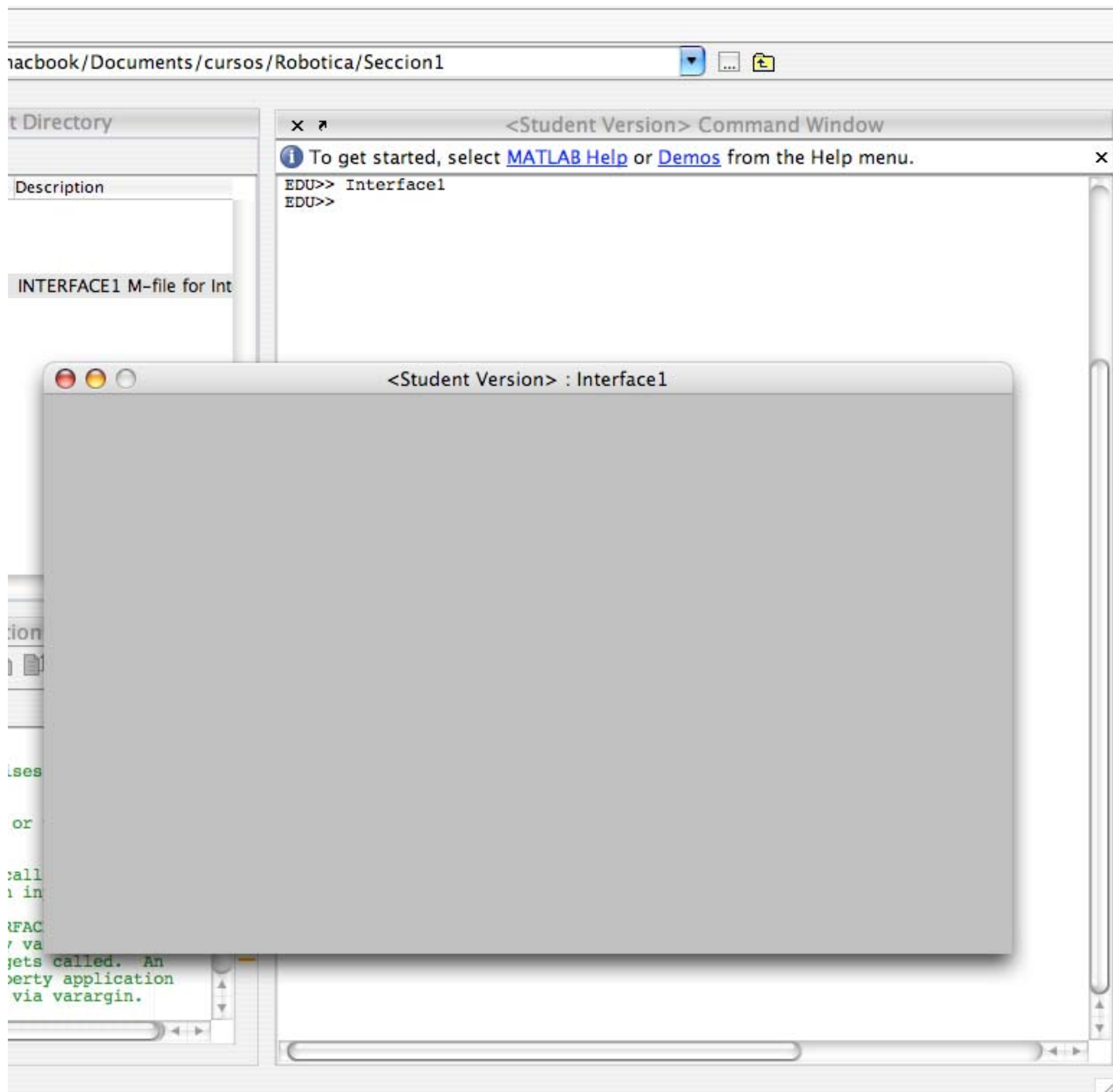
/Users/macbook/Documents/cursos/Robotica/Seccion1/Interface1.m

```
function varargout = Interface1(varargin)
% INTERFACE1 M-file for Interface1.fig
%      INTERFACE1, by itself, creates a new INTERFACE1 or raises the existing
%      singleton*.
%
%      H = INTERFACE1 returns the handle to a new INTERFACE1 or the handle to
%      the existing singleton*.
%
%      INTERFACE1('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in INTERFACE1.M with the given input arguments.
%
%      INTERFACE1('Property','Value',...) creates a new INTERFACE1 or raises the
%      existing singleton*.  Starting from the left, property value pairs are
%      applied to the GUI before Interface1_OpeningFunction gets called.  An
%      unrecognized property name or invalid value makes property application
%      stop.  All inputs are passed to Interface1_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help Interface1

% Last Modified by GUIDE v2.5 30-Aug-2007 23:41:47

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @Interface1_OpeningFcn, ...
                   'gui_OutputFcn',  @Interface1_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before Interface1 is made visible.
function Interface1_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
```

Interface1 Ln 15 Col 51

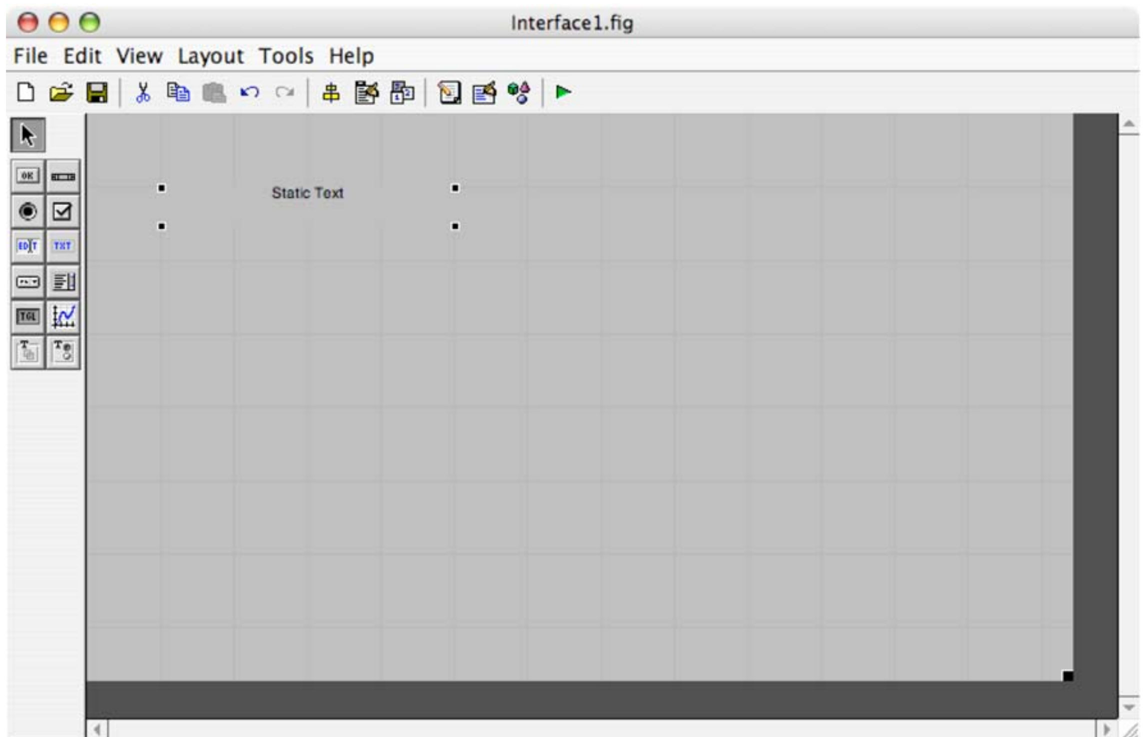
Lo que vemos aquí es el archivo de tipo *function* con la extensión característica de MatLab **.m**. Este archivo es construido automáticamente por MatLab y las líneas de código que aparecen son las que crean la interfaz que aparece en el archivo **interface1.fig**. El archivo de inicialización define los parámetros básicos de la interfaz y crea un conjunto de *handles* para cada uno de los objetos que vayan apareciendo sobre la interfaz.

Si hacemos clic sobre el archivo **interface1.fig**, o invocamos a **interface1.m** en la ventana de comandos obtenemos los siguiente:

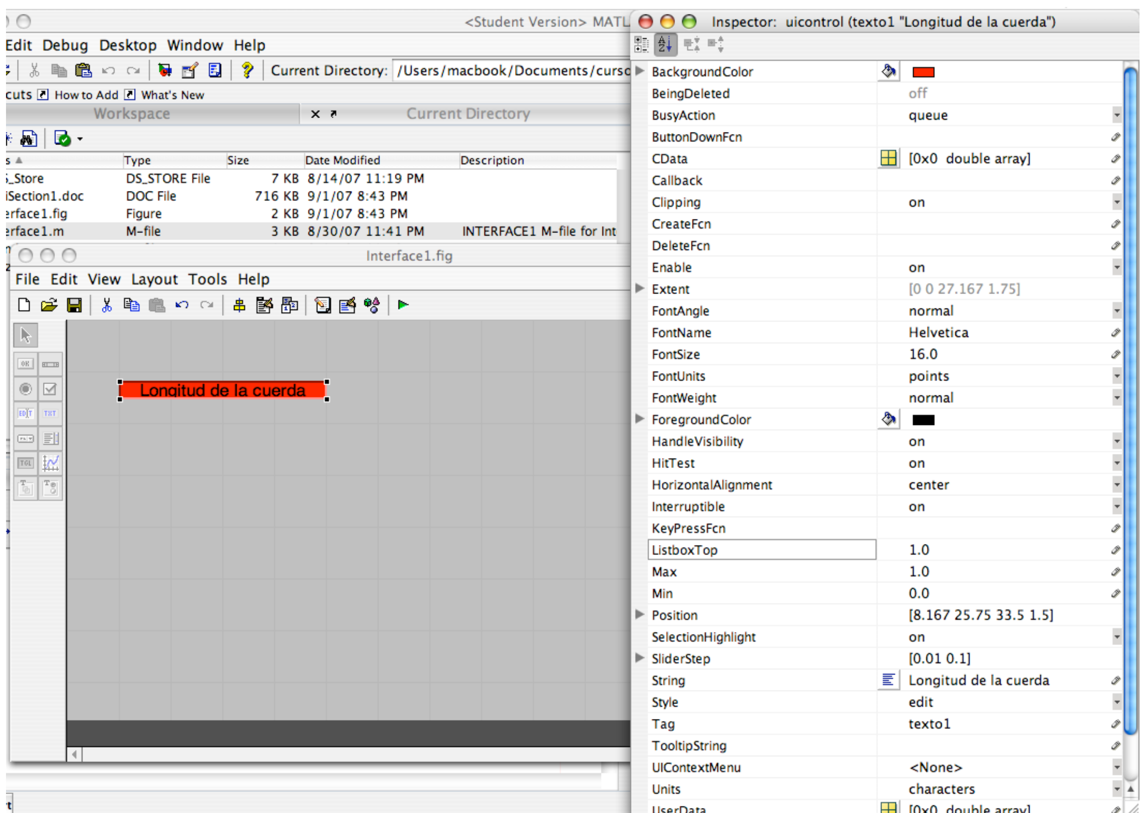


Como se puede apreciar, la interfaz no contiene nada más que un lienzo vacío sin ninguna funcionalidad. El código de inicialización que está en este momento escrito en **Interface1.m** se encarga de crear la interfaz tal y como está, y por esa razón ese código no debe ser modificado.

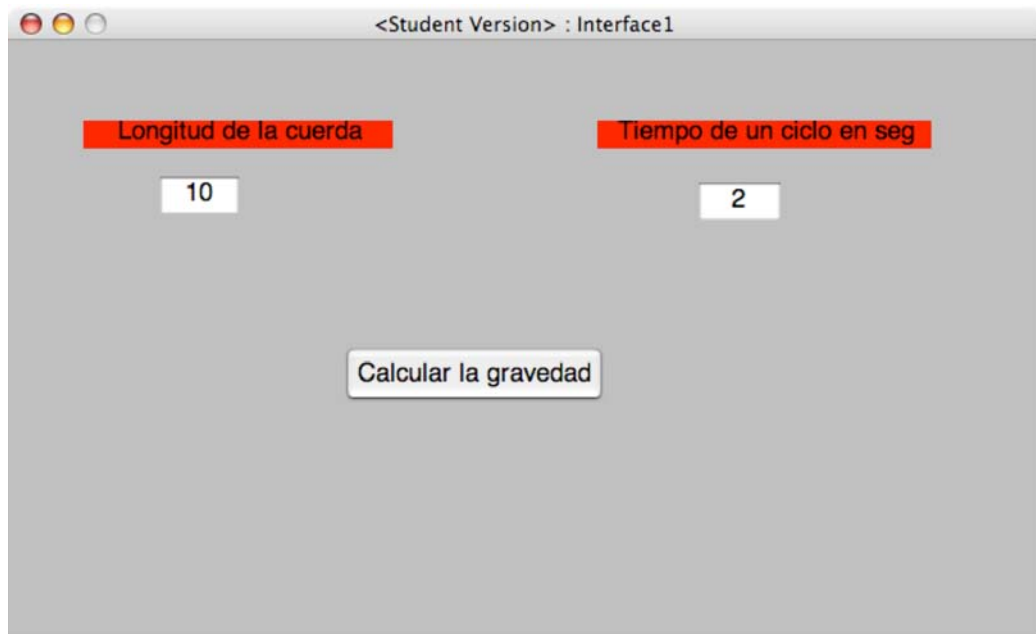
Ahora vamos a seleccionar con el ratón el botón que dice *text* sobre la ventana para manipular el lienzo de la GUI que estamos creando. Al hacerlo, el ratón nos permite seleccionar un área sobre la que colocaremos el texto que deseamos que aparezca sobre el lienzo de la GUI. Así:



Una vez que tenemos el espacio de texto en el lugar deseado, podemos ir al menú view-> property inspector, o bien hacer clic sobre el objeto de texto y obtenemos una ventana desde la que podemos cambiar las propiedades del espacio de texto que hemos creado:



Se aprecia que hemos cambiado el color del fondo del campo de texto a través de la propiedad BackgroundColor que aparece sobre el inspector de propiedades del objeto de texto. La fuente elegida ha sido Helvética de 16 puntos. El *String* o cadena de texto introducida a través del menú ha sido "Longitud de la Cuerda", y al objeto entero le hemos puesto un nombre de texto1, que es como aparece referido en la cabecera de la ventana del inspector de propiedades, ese es su **Tag**. Procedemos de la misma forma hasta completar la interfaz tal y como aparece a continuación:



Esta interfaz toma dos valores observados de un péndulo, que son la longitud de la cuerda y el tiempo que tarda en hacer un ciclo. Si el péndulo se balancea con un ángulo pequeño de modo que el seno del ángulo sea aproximado al valor del propio ángulo, entonces el periodo puede ser aproximado a través de:

$$T \approx 2\pi\sqrt{\frac{l}{g}}$$

Dado que el periodo es una dato y también lo es la longitud de la cuerda del péndulo, despejamos el valor de g, y así obtenemos un valor para la gravedad en el lugar en el que hagamos el experimento. El valor de g extraído de esta fórmula es lo que nuestra GUI calculará. Para ello, de acuerdo con lo dicho en la anterior sección necesitaremos de un pequeño programa, un *script*, que haga los cálculos para nosotros.
Ese programa podría ser así:

```
clear

T   =   10;     %periodo en segundos
l   =   2;      %longitud en metros
g   =   l/(T/(2*pi))^2;
```

que hace exactamente lo que nosotros queremos. Ahora necesitamos que al pinchar con el ratón sobre el botón que calcula la gravedad el programa que acabamos de escribir se ejecute, es decir, tenemos que describir el *callback* asociado al *pushbutton* que hemos llamado “Calcular la Gravedad”.
Para ver los *callbacks* asociados a cada uno de los objetos que hemos creado podemos editar el archivo **Interface1.m** para ver en qué estado ha quedado al llegar al punto en el que estamos. Vemos que unas líneas adicionales al código inicial han aparecido:

```
function texto2_Callback(hObject, eventdata, handles)
% hObject    handle to texto2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of texto2 as text
%        str2double(get(hObject,'String')) returns contents of texto2 as a double


% --- Executes during object creation, after setting all properties.
function texto2_CreateFcn(hObject, eventdata, handles)
% hObject    handle to texto2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function texto4_Callback(hObject, eventdata, handles)
% hObject    handle to texto4 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of texto4 as text
%        str2double(get(hObject,'String')) returns contents of texto4 as a double


% --- Executes during object creation, after setting all properties.
function texto4_CreateFcn(hObject, eventdata, handles)
% hObject    handle to texto4 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'), get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in Calcula.
function Calcula_Callback(hObject, eventdata, handles)
% hObject    handle to Calcula (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

Vemos cómo, por ejemplo, el objeto que hemos llamado texto2 tiene dos funciones asociadas, una que es su *callback* y le da funcionalidad, y otra que es la que crea al objeto y le da apariencia. El propio programa de MatLab nos da automáticamente una sugerencia sobre las cosas que podemos hacer con el texto editado y es que a través del comando **get**, seremos capaces de capturar el texto que el usuario haya introducido en el campo correspondiente. Así, si escribimos dentro del espacio definido para la función *callback* del texto2:

a_texto = get(hObject, ‘String’);

entonces MatLab creará una variable con nombre **a**, y esa variable será del tipo *String* (texto), y su valor será el texto introducido en ese espacio.

Como con la variable **a** no se pueden realizar operaciones aritméticas (por ser texto) es necesario hacer un *casting* para la variable y transformarla de texto a número doble. La sentencia:

a_numero = str2double(a);

hace precisamente esto. Por supuesto **a = str2double(get(hObject, ‘String’));** es la forma de hacerlo todo al mismo tiempo. Una vez que hemos realizado el *casting*, podemos utilizar el valor de las variables para hacer nuestros cálculos, es decir, podemos leer el valor introducido como texto por el usuario en cada una de las dos casillas de edición, y después del correspondiente *cast* usar los valores numéricos para incorporarlos al pequeño programa que hemos escrito.

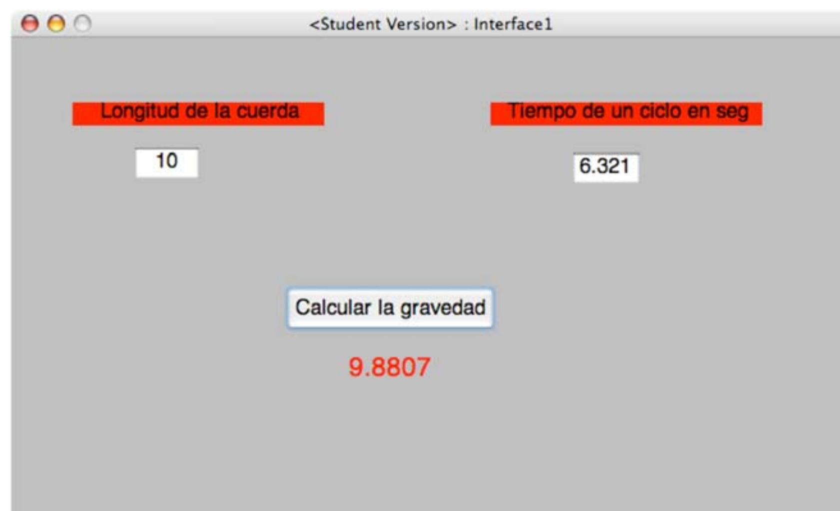
```
121
122     % --- Executes on button press in Calcula.
123     function Calcula_Callback(hObject, eventdata, handles)
124     % hObject    handle to Calcula (see GCBO)
125     % eventdata  reserved - to be defined in a future version of MATLAB
126     % handles    structure with handles and user data (see GUIDATA)
127
128 -   l   =   str2double(get(handles.texto2, 'string'));
129 -   T   =   str2double(get(handles.texto4, 'string'));
130 -   g   =   l/(T/(2*pi))^2;
131
132
```

pend.m | Interface1.m | pend2.m | interface.m

Insertamos el programa que hemos hecho en un *script* y que tenemos convenientemente probado y lo insertamos, asegurándonos de que hacemos las llamadas correctas de los sitios correctos. Como puedes ver, el argumento **hObject** de la función **Calcula_Callback(hObject, eventdata, handles)** es ni más ni menos que la variable estructurada **handles.texto2**. Es importante entender que la variable l o T debe ser invocada siempre de esta manera en cada función que las requiera. Esto es así porque como ya sabes todas las variables dentro de las funciones son variables locales, y por tanto "invisibles" para las funciones en las que no se haya hecho el correspondiente **get**.
Para acabar necesitamos un espacio en nuestra GUI en el que presentar el resultado, o sea, **g**. Para ello vamos a crear un espacio de texto que dejaremos en blanco (con un *string* vacío en la ventana del inspector de propiedades) y usaremos el comando **set** (lo contrario de **get**), para imponer el *string* que haya resultado de la operación realizada con la longitud de la cuerda y el periodo y que nos da la gravedad local.
El *Tag* (o propiedad del objeto a través de la cual MatLab se refiere a él) del objeto de texto que he dejado con un *string* vacío es **mostrar**, por tanto escribimos al final del programa, en la línea 132, el comando:

```
set(handles.mostrar, 'String', num2str(g))
```

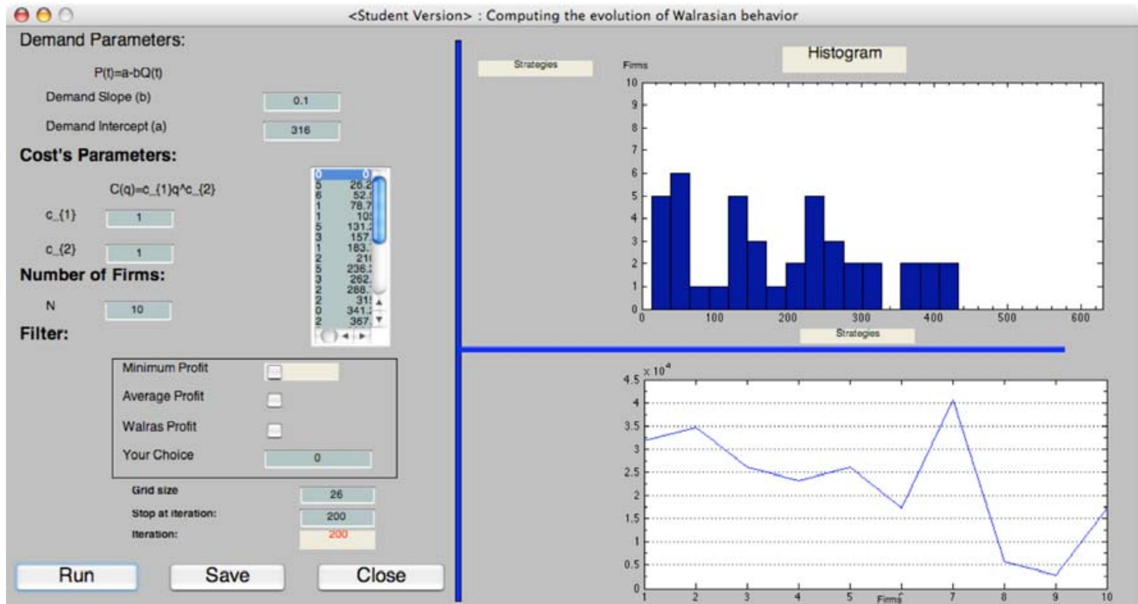
Es decir, con **set** fijamos el valor de la variable asociada al objeto **mostrar**. Dado que en ese objeto sólo podemos escribir texto hacemos el *casting* contrario y pasamos de número a cadena de texto y de ahí el comando **num2str** (que se leería como *number to string,* o en español, "de número a texto").
Ya tenemos nuestra calculadora de la gravedad lista y con este aspecto:



Desde luego más fea no puede ser, pero tiene casi todos los elementos necesarios de una buena interfaz. Dado que sería muy largo y tedioso ir elemento por elemento sugiero que veamos una interfaz algo más complicada y miremos con detenimiento qué cosas hace.

2.3 Estudio de una GUI más complicada.

En esta sección vamos a estudiar una GUI que tiene algunos elementos adicionales sobre la GUI que ya hemos visto: gráficos, botones de cierre, scrolls y chechboxes. La interfaz tiene el siguiente aspecto:



Esta interfaz muestra el comportamiento de un número N (10 por defecto) de empresas que compiten en un mercado con función de demanda dada por la relación lineal inversa entre precio de venta y cantidades vendidas. La función de costes depende de los parámetros c_[1] y c_[2], en la función de costes que aparece en la interfaz. Las empresas pueden lanzar una cantidad al mercado elegida de una malla finita y común a todas ellas (al comienzo de la simulación) de números reales positivos. En la primera iteración cada empresa elige al azar una cantidad individual para lanzar al mercado, soportando el coste de su producción. La suma de todas las cantidades individuales genera la producción total, y esta a su vez induce un precio de venta de acuerdo con la función de demanda. El precio de venta multiplicado por la cantidad individual de cada empresa induce una distribución de ingresos, en tanto que la producción induce unos costes, que sustraídos a los ingresos produce unos beneficios.
La interfaz quiere ser una ayuda para ver qué sucede cando en un entorno empresarial como el descrito, las empresas deciden eliminar determinadas acciones (volúmenes de producción) dependiendo de los resultados relativos que obtienen. Por ejemplo, si en la zona de la interfaz que dice **Filter**, seleccionamos la casilla Average Profit (beneficio medio), esto querrá decir que las empresas comparan sus beneficios después de haber elegido una cantidad al azar, si esa cantidad elegida al azar no le genera a la empresa un beneficio al menos tan alto como el beneficio medio, entonces esa empresa nunca jamás volverá a elegir ese volumen de producción en el futuro. La pregunta es: en entornos tan sencillos

como el planteado, ¿cuál es la distribución de producción individual de largo plazo?. Eso es lo que muestra el histograma que aparece dentro de la interfaz. Por qué los economistas piensan que esta pregunta es interesante se sale del objetivo del curso pero lo cierto es que hay una literatura enorme en relación a este tema.

Lo que a nosotros nos interesa es ver la conexión entre la interfaz y el código.

El código asociado a esta interfaz es el siguiente:

```
function varargout = interface(varargin)
% INTERFACE Application M-file for interface.fig
%    FIG = INTERFACE launch interface GUI.
%    INTERFACE('callback_name', ...) invoke the named callback.

% Last Modified by GUIDE v2.0 05-Oct-2005 23:46:13

if nargin == 0  % LAUNCH GUI

    fig = openfig(mfilename,'reuse');

    % Use system color scheme for figure:
    set(fig,'Color',get(0,'defaultUicontrolBackgroundColor'));

    % Generate a structure of handles to pass to callbacks, and store it.
    handles = guihandles(fig);
    guidata(fig, handles);

    if nargout > 0
        varargout{1} = fig;
    end

elseif ischar(varargin{1}) % INVOKE NAMED SUBFUNCTION OR CALLBACK

    try
        if (nargout)
            [varargout{1:nargout}] = feval(varargin{:}); % FEVAL switchyard
        else
            feval(varargin{:}); % FEVAL switchyard
        end
    catch
        disp(lasterr);
    end

end


%| ABOUT CALLBACKS:
%| GUIDE automatically appends subfunction prototypes to this file, and
%| sets objects' callback properties to call them through the FEVAL
%| switchyard above. This comment describes that mechanism.
%|
%| Each callback subfunction declaration has the following form:
%| <SUBFUNCTION_NAME>(H, EVENTDATA, HANDLES, VARARGIN)
%|
%| The subfunction name is composed using the object's Tag and the
%| callback type separated by '_', e.g. 'slider2_Callback',
%| 'figure1_CloseRequestFcn', 'axis1_ButtondownFcn'.
%|
%| H is the callback object's handle (obtained using GCBO).
%|
%| EVENTDATA is empty, but reserved for future use.
%|
%| HANDLES is a structure containing handles of components in GUI using
%| tags as fieldnames, e.g. handles.figure1, handles.slider2. This
%| structure is created at GUI startup using GUIHANDLES and stored in
%| the figure's application data using GUIDATA. A copy of the structure
%| is passed to each callback.  You can store additional information in
%| this structure at GUI startup, and you can change the structure
%| during callbacks.  Call guidata(h, handles) after changing your
%| copy to replace the stored original so that subsequent callbacks see
%| the updates. Type "help guihandles" and "help guidata" for more
%| information.
%|
%| VARARGIN contains any extra arguments you have passed to the
%| callback. Specify the extra arguments by editing the callback
```

```
%| property in the inspector. By default, GUIDE sets the property to:
%| <MFILENAME>('<SUBFUNCTION_NAME>', gcbo, [], guidata(gcbo))
%| Add any extra arguments after the last argument, before the final
%| closing parenthesis.
% --------------------------------------------------------------------
function varargout = slope_Callback(h, eventdata, handles, varargin)

b    =   str2double(get(handles.slope,'string'))
    if isnan(b)
        errordlg('You must enter a numeric value', 'Bad Input', 'modal')
    end
     if b<=0
         errordlg('You must enter a positive value', 'Bad Input', 'modal')
     end


% --------------------------------------------------------------------
function varargout = intercept_Callback(h, eventdata, handles, varargin)

a    =   str2double(get(handles.intercept,'string'))
    if isnan(a)
        errordlg('You must enter a numeric value', 'Bad Input', 'modal')
    end

    if a<=0
        errordlg('You must enter a positive value', 'Bad Input', 'modal')
    end




% --------------------------------------------------------------------
function varargout = UnitCost_Callback(h, eventdata, handles, varargin)

c1   =   str2double(get(handles.UnitCost,'string'))
    if isnan(c1)
        errordlg('You must enter a numeric value', 'Bad Input', 'modal')
    end

    if c1<0
        errordlg('You must enter a positive value', 'Bad Input', 'modal')
    end




% --------------------------------------------------------------------
function varargout = expCost_Callback(h, eventdata, handles, varargin)

c2   =   str2double(get(handles.expCost,'string'))
    if isnan(c2)
        errordlg('You must enter a numeric value', 'Bad Input', 'modal')
    end

    if c2<0
        errordlg('You must enter a positive value', 'Bad Input', 'modal')
    end


% --------------------------------------------------------------------
function varargout = numFirms_Callback(h, eventdata, handles, varargin)

N    =   str2double(get(handles.numFirms,'string'));
    if isnan(N)
        errordlg('You must enter a numeric value', 'Bad Input', 'modal')
    end

    if N<2
        errordlg('You must enter at least two firms', 'Bad Input', 'modal')
    end

    if (N==round(N))==0;
        N    =   round(N);
        warndlg('Rounding the number of firms to nearest integer', 'Bad Input', 'non-
modal')
        set(handles.numFirms, 'String', round(N));
    else
    end
```

```
% --------------------------------------------------------------------
function varargout = minProfit_Callback(h, eventdata, handles, varargin)

if (get(handles.minProfit, 'Value') == get(handles.minProfit, 'Max'))

    selection =1;
else
end


% --------------------------------------------------------------------
function varargout = averageProfit_Callback(h, eventdata, handles, varargin)

if (get(handles.averageProfit, 'Value') == get(handles.averageProfit, 'Max'))

    selection =2;
else
end




% --------------------------------------------------------------------
function varargout = walrasProfit_Callback(h, eventdata, handles, varargin)

if (get(handles.walrasProfit, 'Value') == get(handles.walrasProfit, 'Max'))

    selection = 3;
    else
    end


% --------------------------------------------------------------------
function varargout = yourChoice_Callback(h, eventdata, handles, varargin)

    selection = 4;
yc  =   str2double(get(handles.yourChoice,'string'))
    if isnan(yc)
        errordlg('You must enter a numeric value', 'Bad Input', 'modal')
    end


% --------------------------------------------------------------------
function varargout = gridSize_Callback(h, eventdata, handles, varargin)

gs  =   str2double(get(handles.gridSize,'string'))
    if isnan(gs)
        errordlg('You must enter a numeric value', 'Bad Input', 'modal')
    elseif (gs==round(gs))==0;
        gs    =   round(gs);
        warndlg('Rounding the number of firms to nearest integer', 'Bad Input', 'non-
modal')
        set(handles.gridSize, 'String', round(gs));
    else

    end

    if gs<=2
        warndlg('Minimum grid size is 3', 'Bad Input', 'modal')
        set(handles.gridSize, 'String', 3);
    end

% --------------------------------------------------------------------
function varargout = stopIter_Callback(h, eventdata, handles, varargin)

si  =   str2double(get(handles.stopIter,'string'))
    if isnan(si)
        errordlg('You must enter a numeric value', 'Bad Input', 'modal')
    elseif (si==round(si))==0;
        si    =   round(si);
        warndlg('Rounding number of iterations to nearest integer', 'Bad Input', 'non-
modal')
        set(handles.stopIter, 'String', round(si));
    else

    end
```

```
% -------------------------------------------------------------------
function varargout = numIter_Callback(h, eventdata, handles, varargin)



% -------------------------------------------------------------------
function varargout = Ejecutar_Callback(h, eventdata, handles, varargin)

a       =   str2double(get(handles.intercept,'string'));
b       =   str2double(get(handles.slope,'string'));
c1      =   str2double(get(handles.UnitCost,'string'));
c2      =   str2double(get(handles.expCost,'string'));
N       =   str2double(get(handles.numFirms,'string'));
nu      =   str2double(get(handles.gridSize,'string'));


%WALRASIAN ALLOCATION
param   =   [a b c1 c2 N];
crit    =   1e-4;
maxit   =   1000;
x0      =   a/(2*b*N);
qjWal   =   secant('walras'  , x0, param, crit, maxit);

%STRATEGY SETs: S=[0:delta:2*qjWal]
Z       =   (nu/2-floor(nu/2)==0);
delta   =   qjWal/((nu+1-Z)/2-1);
S       =   [0:delta:2*qjWal];
S       =   sort([S]);
T       =   1000; %length(S)*N;
rand('state',sum(100*clock));

%INITIAL STATE
for i=1:N
    eval(['S' num2str(i) '=S';]);                %Creacion de N conjuntos S iguales
end


%WALRAS
QWal    =   N*qjWal;
PWal    =   a-b*QWal;
PiWal   =   PWal*qjWal-c1*qjWal^c2;
Rescued =   [];
single  =   0;
t       =   1;

while single<N

    for i=1:N
        q(i)     =   LuniformValues(eval(['S' num2str(i)]),1); %Seleccion uniforme sobre
cada uno de los conjuntos
    end

        if mean(q)~=q(1)
            Q(t)        =   sum(q);
            P(t)        =   max(a-b*Q(t), 0);
            Pi(:,t)     =   P(t).*q'-c1*q'.^c2;

                if get(handles.minProfit, 'Value') == 1;
                        y           =   minfinder((Pi(:,t)));
                    elseif get(handles.averageProfit, 'Value') == 1;
                        y           =   find(Pi(:,t)<mean(Pi(:,t)));
                    elseif get(handles.walrasProfit, 'Value') == 1;
                        y           =   find(Pi(:,t)<=PiWal);
                    else
                        yc          =   str2double(get(handles.yourChoice,'string'));
                        y           =   find(Pi(:,t)<=yc)
                end

                for i=1:length(y)
                    A       =   eval(['S' num2str(y(i));]);
                    if length(A)>1;
                        I(i)     =   find(A==q(y(i)));
                        eval(['S' num2str(y(i)) '=setdiff(A,A(I(i)))';]);
                    else
                    end
                end
        else
            Rescued =   [Rescued q'];
```

```
        end


    single  =   0;
    for i=1:N
        if length(eval(['S' num2str(i);]))==1; single=single+1; end
    end

    Ac   =   [];
    for i=1:N
        A   =   eval(['S' num2str(i);]);
        Ac  =   [Ac A];
    end
    axes(handles.frequency_axes)
    hist(Ac,S);
    [m,n]=hist(Ac,S);
    set(handles.frequency_axes, 'XLim', [0 S(length(S))])
    set(handles.frequency_axes, 'YLim', [0 N])
    set(handles.frequency_axes,'XMinorTick','on')
    grid off
    pause(0.01)

    axes(handles.profit_axes)
    plot(Pi(:,t))
    set(handles.profit_axes,'YMinorTick','on')
    set(handles.profit_axes, 'XLim', [1 N])
    set(handles.profit_axes, 'YGrid', 'on')
    %grid off
    pause(0.01)

    set(handles.mostrar, 'String', num2str([m; n]'))

    t   =   t+1;
    if length(Ac)==N & length(y)~=0; break; end
    set(handles.numIter, 'String', num2str(t), 'ForegroundColor', [1 0 0])
    if t == str2double(get(handles.stopIter, 'String')); break; end



end


% ----------------------------------------------------------------------
function varargout = mostrar_Callback(h, eventdata, handles, varargin)



% ----------------------------------------------------------------------
function varargout = Save_Callback(h, eventdata, handles, varargin)




% ----------------------------------------------------------------------
function varargout = Close_Callback(h, eventdata, handles, varargin)

delete(handles.figure1)


% ----------------------------------------------------------------------
function x=secant(func, x0, param, crit, maxit)
del=diag(max(abs(x0)*1e-4, 1e-8));
n=length(x0);
for i=1:maxit
   f=feval(func,x0,param);
   for j=1:n
      J(:,j)=(f-feval(func,x0-del(:,j),param))/del(j,j);
   end
   x=x0-inv(J)*f;
   if norm(x-x0)<crit; break; end
   x0=x;
end
if i>=maxit
        warndlg('The number of iterations is too large', 'Bad Input', 'non-modal')
end


% ----------------------------------------------------------------------
function f = walras(x0, param)
```

```
a   =   param(1);
b   =   param(2);
c1  =   param(3);
c2  =   param(4);
N   =   param(5);
xj  =   x0;
f   =   (a-b*N*xj)-c1*c2*xj^(c2-1);    %Precio = Coste Marginal


% --------------------------------------------------------------------
function f = LuniformValues(x,l)

%This function extracts L elements of vector x according to a uniform distribution over
%x
n   =   length(x);
for i=1:l
r(i)   =   ceil(n*rand);
end
f   =   x(r);


% --------------------------------------------------------------------
function posvec=minfinder(x)
minimo  =   min(x);
x       ==  minimo;
y       =   ans;
posvec  =   [];

for i=1:length(y)
        if y(i)==1;
            pos = i;
        else
            pos = [];
        end
        posvec=[posvec pos];
end
```