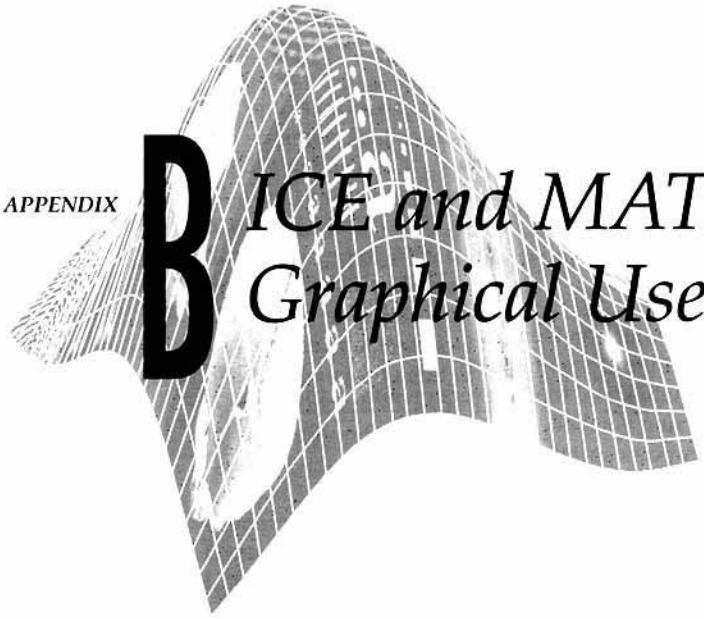


APPENDIX B

ICE and MATLAB Graphical User Interfaces

Preview

In this appendix we develop the `ice` *interactive color editing* (ICE) function introduced in Chapter 7. The discussion assumes familiarity on the part of the reader with the material in Section 7.4. Section 7.4 provides many examples of using `ice` in both pseudo- and full-color image processing (Examples 7.5 through 7.9) and describes the `ice` calling syntax, input parameters, and graphical interface elements (they are summarized in Tables 7.7 through 7.9). The power of `ice` is its ability to let users generate color transformation curves interactively and graphically, while displaying the impact of the generated transformations on images in real or near real time.

B.1 Creating ICE's Graphical User Interface

MATLAB's *Graphical User Interface Development Environment* (GUIDE) provides a rich set of tools for incorporating *graphical user interfaces* (GUIs) in M-functions. Using GUIDE, the processes of (1) laying out a GUI (i.e., its buttons, pop-up menus, etc.) and (2) programming the operation of the GUI are divided conveniently into two easily managed and relatively independent tasks. The resulting graphical M-function is composed of two identically named (ignoring extensions) files:

1. A file with extension `.fig`, called a *FIG-file*, that contains a complete graphical description of all the function's GUI objects or elements and their spatial arrangement. A FIG-file contains binary data that does not need to be parsed when the associated GUI-based M-function is executed. The FIG-file for ICE (`ice.fig`) is described later in this section.
2. A file with extension `.m`, called a *GUI M-file*, which contains the code that controls the GUI operation. This file includes functions that are called

when the GUI is launched and exited, and *callback functions* that are executed when a user interacts with GUI objects—for example, when a button is pushed. The GUI M-file for ICE (`ice.m`) is described in the next section.

To launch GUIDE from the MATLAB command window, type

```
guide filename
```

where `filename` is the name of an existing FIG-file on the current path. If `filename` is omitted, GUIDE opens a new (i.e., blank) window.

Figure B.1 shows the GUIDE *Layout Editor* (launched by entering `guide ice` at the MATLAB >> prompt) for the Interactive Color Editor (ICE) layout. The Layout Editor is used to select, place, size, align, and manipulate graphic objects on a mock-up of the user interface under development. The buttons on its left side form a *Component Palette* containing the GUI objects that are supported—*Push Buttons*, *Sliders*, *Radio Buttons*, *Checkboxes*, *Edit Texts*, *Static Texts*, *Popup Menus*, *Listboxes*, *Toggle Buttons*, *Tables*, *Axes*, *Panels*, *Button Groups*, and *ActiveX Controls*. Each object is similar in behavior to its standard Windows' counterpart. And any combination of objects can be added to the figure object in the layout area on the right side of the Layout Editor. Note that the ICE GUI includes checkboxes (`Smooth`, `Clamp Ends`, `Show PDF`, `Show`

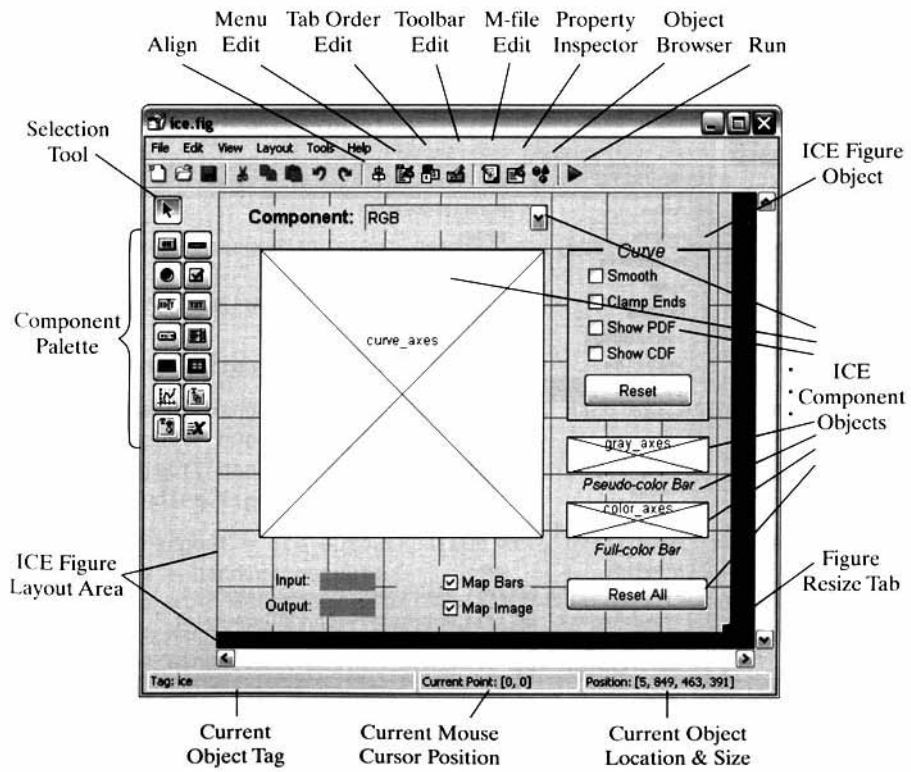
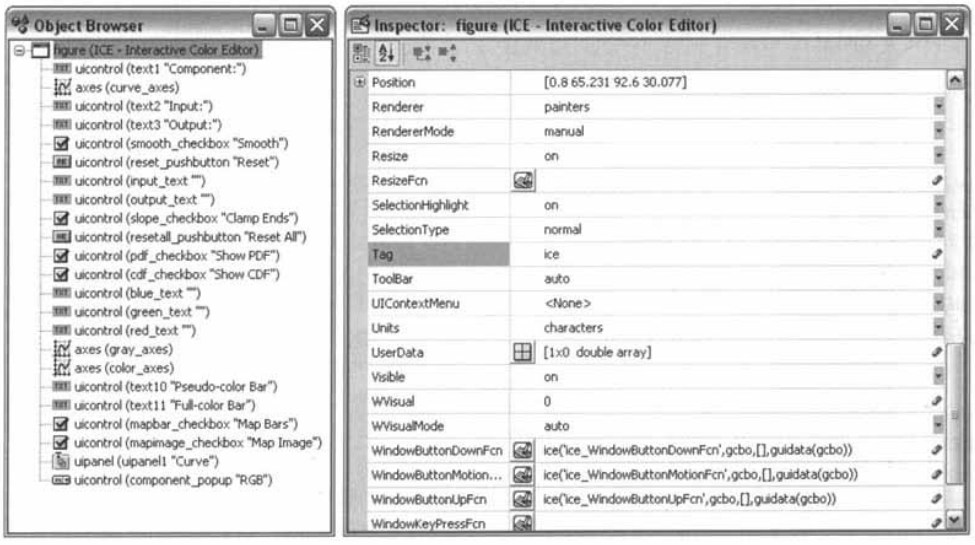


FIGURE B.1
The GUIDE Layout Editor mockup of the ICE GUI.

`CDF`, `Map Bars`, and `Map Image`), static text ("Component:", "Input:", ...), a panel outlining the curve controls, two push buttons (`Reset` and `Reset All`), a popup menu for selecting a color transformation curve, and three `axes` objects for displaying the selected curve (with associated control points) and its effect on both a gray-scale wedge and hue wedge. A hierarchical list of the elements comprising ICE (obtained by clicking the *Object Browser* button in the task bar at the top of the Layout Editor) is shown in Fig. B.2(a). Note that each element has been given a unique name or tag. For example, the axes object for curve display (at the top of the list) is assigned the identifier `curve_axes` [the identifier is the first entry after the open parenthesis in Fig. B.2(a)].

Tags are one of several *properties* that are common to all GUI objects. A scrollable list of the properties characterizing a specific object can be obtained by selecting the object [in the Object Browser list of Fig. B.2(a) or layout area of Fig. B.1 using the *Selection Tool*] and clicking the *Property Inspector* button on the Layout Editor's task bar. Figure B.2(b) shows the list that is generated when the `figure` object of Fig. B.2(a) is selected. Note that the `figure` object's `Tag` property [highlighted in Fig. B.2(b)] is `ice`. This property is important because GUIDE uses it to automatically generate `figure` callback function names. Thus, for example, the `WindowButtonDownFcn` property at the bottom of the scrollable Property Inspector window, which is executed when a mouse button is pressed over the figure window, is assigned the name `ice_WindowButtonDownFcn`. Recall that callback functions are merely M-functions that are executed when a user interacts with a GUI object. Other notable (and common to all GUI objects) properties include the `Position` and `Units` properties, which define the size and location of an object.

The GUIDE generated `figure` object is a container for all other objects in the interface.



a b

FIGURE B.2 The (a) GUIDE Object Browser and (b) Property Inspector for the ICE "figure" object.

Finally, we note that some properties are unique to particular objects. A pushbutton object, for example, has a `Callback` property that defines the function that is executed when the button is pressed and a `String` property that determines the button's label. The `Callback` property of the ICE `Reset` button is `reset_pushbutton_Callback` [note the incorporation of its `Tag` property from Fig. B.2(a) in the callback function name]; its `String` property is "Reset". Note, however, that the `Reset` pushbutton does not have a `WindowButtonMotionFcn` property; it is specific to "figure" objects.

B.2 Programming the ICE Interface

When the ICE FIG-file of the previous section is first saved or the GUI is first run (e.g., by clicking the *Run* button on the Layout Editor's task bar), GUIDE generates a starting *GUI M-file* called `ice.m`. This file, which can be modified using a standard text editor or MATLAB's M-file editor, determines how the interface responds to user actions. The automatically generated GUI M-file for ICE is as follows:

To enable M-file generation, select **Tools** and **GUI Options ...** and check the "Generate FIG-file and M-file" option.

`ice`

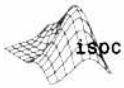
GUIDE generated starting M-file.

```
function varargout = ice(varargin)
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',        mfilename, ...
                   'gui_Singleton',   gui_Singleton, ...
                   'gui_OpeningFcn',  @ice_OpeningFcn, ...
                   'gui_OutputFcn',   @ice_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

function ice_OpeningFcn(hObject, eventdata, handles, varargin)
handles.output = hObject;
guidata(hObject, handles);
% uiwait(handles.figure1);

function varargout = ice_OutputFcn(hObject, eventdata, handles)
varargout{1} = handles.output;
function ice_WindowButtonDownFcn(hObject, eventdata, handles)
function ice_WindowButtonMotionFcn(hObject, eventdata, handles)
function ice_WindowButtonUpFcn(hObject, eventdata, handles)
function smooth_checkbox_Callback(hObject, eventdata, handles)
function reset_pushbutton_Callback(hObject, eventdata, handles)
function slope_checkbox_Callback(hObject, eventdata, handles)
```

```
function resetall_pushbutton_Callback(hObject, eventdata, handles)
function pdf_checkbox_Callback(hObject, eventdata, handles)
function cdf_checkbox_Callback(hObject, eventdata, handles)
function mapbar_checkbox_Callback(hObject, eventdata, handles)
function mapimage_checkbox_Callback(hObject, eventdata, handles)
function component_popup_Callback(hObject, eventdata, handles)
function component_popup_CreateFcn(hObject, eventdata, handles)
if ispc && isequal(get(hObject,'BackgroundColor'), ...
        get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```



Returns 1 for PC (Windows) versions of MATLAB and 0 otherwise.

This automatically generated file is a useful starting point or prototype for the development of the fully functional `ice` interface. (Note that we have stripped the file of many GUIDE-generated comments to save space.) In the sections that follow, we break this code into four basic sections: (1) the initialization code between the two "`DO NOT EDIT`" comment lines, (2) the figure opening and output functions (`ice_OpeningFcn` and `ice_OutputFcn`), (3) the figure callback functions (i.e., the `ice_WindowButtonDownFcn`, `ice_WindowButtonMotionFcn`, and `ice_WindowButtonUpFcn` functions), and (4) the object callback functions (e.g., `reset_pushbutton_Callback`). When considering each section, completely developed versions of the `ice` functions contained in the section are given, and the discussion is focused on features of general interest to most GUI M-file developers. The code introduced in each section will not be consolidated (for the sake of brevity) into a single comprehensive listing of `ice.m`. It is introduced in a piecemeal manner.

The operation of `ice` was described in Section 7.4. It is also summarized in the following Help text block from the fully developed `ice.m` M-function:

ice

Help text block of the final version.

```
%ICE Interactive Color Editor.
%
%   OUT = ICE('Property Name', 'Property Value', ...) transforms an
%   image's color components based on interactively specified mapping
%   functions. Inputs are Property Name/Property Value pairs:
%
%     Name            Value
%     ------------    ------------------------------------------------
%     'image'         An RGB or monochrome input image to be
%                     transformed by interactively specified
%                     mappings.
%     'space'         The color space of the components to be
%                     modified. Possible values are 'rgb', 'cmy',
%                     'hsi', 'hsv', 'ntsc' (or 'yiq'), 'ycbcr'. When
%                     omitted, the RGB color space is assumed.
%     'wait'          If 'on' (the default), OUT is the mapped input
%                     image and ICE returns to the calling function
%                     or workspace when closed. If 'off', OUT is the
%                     handle of the mapped input image and ICE
%                     returns immediately.
```

```
%
%   EXAMPLES:
%     ice OR ice('wait', 'off')           % Demo user interface
%     ice('image', f)                     % Map RGB or mono image
%     ice('image', f, 'space', 'hsv')     % Map HSV of RGB image
%     g = ice('image', f)                 % Return mapped image
%     g = ice('image', f, 'wait', 'off'); % Return its handle
%
%   ICE displays one popup menu selectable mapping function at a
%   time. Each image component is mapped by a dedicated curve (e.g.,
%   R, G, or B) and then by an all-component curve (e.g., RGB). Each
%   curve's control points are depicted as circles that can be moved,
%   added, or deleted with a two- or three-button mouse:
%
%     Mouse Button    Editing Operation
%     ------------    ---------------------------------------------
%     Left            Move control point by pressing and dragging.
%     Middle          Add and position a control point by pressing
%                     and dragging. (Optionally Shift-Left)
%     Right           Delete a control point. (Optionally
%                     Control-Left)
%
%   Checkboxes determine how mapping functions are computed, whether
%   the input image and reference pseudo- and full-color bars are
%   mapped, and the displayed reference curve information (e.g.,
%   PDF):
%
%     Checkbox        Function
%     ------------    ---------------------------------------------
%     Smooth          Checked for cubic spline (smooth curve)
%                     interpolation. If unchecked, piecewise linear.
%     Clamp Ends      Checked to force the starting and ending curve
%                     slopes in cubic spline interpolation to 0. No
%                     effect on piecewise linear.
%     Show PDF        Display probability density function(s) [i.e.,
%                     histogram(s)] of the image components affected
%                     by the mapping function.
%     Show CDF        Display cumulative distributions function(s)
%                     instead of PDFs.
%                     <Note: Show PDF/CDF are mutually exclusive.>
%     Map Image       If checked, image mapping is enabled; else
%                     not.
%     Map Bars        If checked, pseudo- and full-color bar mapping
%                     is enabled; else display the unmapped bars (a
%                     gray wedge and hue wedge, respectively).
%
%   Mapping functions can be initialized via pushbuttons:
%
%     Button          Function
%     ------------    ---------------------------------------------
```

```
%     Reset          Init the currently displayed mapping function
%                    and uncheck all curve parameters.
%     Reset All      Initialize all mapping functions.
```

B.2.1 Initialization Code

The opening section of code in the starting GUI M-file (at the beginning of Section B.2) is a standard GUIDE-generated block of initialization code. Its purpose is to build and display ICE's GUI using the M-file's companion FIG-file (see Section B.1) and control access to all internal M-file functions. As the enclosing "`DO NOT EDIT`" comment lines indicate, the initialization code should not be modified. Each time `ice` is called, the initialization block builds a structure called `gui_State`, which contains information for accessing `ice` functions. For instance, named field `gui_Name` (i.e., `gui_State.gui_Name`) contains the MATLAB function `mfilename`, which returns the name of the currently executing M-file. In a similar manner, fields `gui_OpeningFcn` and `gui_OutputFcn` are loaded with the GUIDE generated names of `ice`'s opening and output functions (discussed in the next section). If an ICE GUI object is activated by the user (e.g., a button is pressed), the name of the object's callback function is added as field `gui_Callback` [the callback's name would have been passed as a string in `varargin(1)`].

After structure `gui_State` is formed, it is passed as an input argument, along with `varargin(:)`, to function `gui_mainfcn`. This MATLAB function handles GUI creation, layout, and callback dispatch. For `ice`, it builds and displays the user interface and generates all necessary calls to its opening, output, and callback functions. Since older versions of MATLAB may not include this function, GUIDE is capable of generating a stand-alone version of the normal GUI M-file (i.e., one in which the FIG-file is replaced with a MAT-file) by selecting **Export ...** from the **File** menu. In the stand-alone version, function `gui_mainfcn` and several supporting routines, including `ice_LayoutFcn` and `local_openfig`, are appended to the normally FIG-file dependent M-file. The role of `ice_LayoutFcn` is to create the ICE GUI. In the stand-alone version of `ice`, it includes the statement

To choose a compatibility mode, select **File** and **Preferences ...**, followed by *General* and *MAT-Files*, and choose a MAT-file save format.

```
h1 = figure(...
'Units', 'characters',...
'Color', [0.87843137254902 0.874509803921569 0.890196078431373],...
'Colormap', [0 0 0.5625;0 0 0.625;0 0 0.6875;0 0 0.75;...
             0 0 0.8125;0 0 0.875;0 0 0.9375;0 0 1;0 0.0625 1;...
             0 0.125 1;0 0.1875 1;0 0.25 1;0 0.3125 1;0 0.375 1;...
             0 0.4375 1;0 0.5 1;0 0.5625 1;0 0.625 1;0 0.6875 1;...
             0 0.75 1;0 0.8125 1;0 0.875 1;0 0.9375 1;0 1 1;...
             0.0625 1 1;0.125 1 0.9375;0.1875 1 0.875;...
             0.25 1 0.8125;0.3125 1 0.75;0.375 1 0.6875;...
             0.4375 1 0.625;0.5 1 0.5625;0.5625 1 0.5;...
             0.625 1 0.4375;0.6875 1 0.375;0.75 1 0.3125;...
             0.8125 1 0.25;0.875 1 0.1875;0.9375 1 0.125;...
             1 1 0.0625;1 1 0;1 0.9375 0;1 0.875 0;1 0.8125 0;...
```

```
            1 0.75 0;1 0.6875 0;1 0.625 0;1 0.5625 0;1 0.5 0;...
            1 0.4375 0;1 0.375 0;1 0.3125 0;1 0.25 0;...
            1 0.1875 0;1 0.125 0;1 0.0625 0;1 0 0;0.9375 0 0;...
            0.875 0 0;0.8125 0 0;0.75 0 0;0.6875 0 0;0.625 0 0;...
            0.5625 0 0],...
'IntegerHandle', 'off',...
'InvertHardcopy', get(0, 'defaultfigureInvertHardcopy'),...
'MenuBar', 'none',...
'Name', 'ICE - Interactive Color Editor',...
'NumberTitle', 'off',...
'PaperPosition', get(0, 'defaultfigurePaperPosition'),...
'Position', [0.8 65.2307692307693 92.6 30.0769230769231],...
'Renderer', get(0, 'defaultfigureRenderer'),...
'RendererMode', 'manual',...
'WindowButtonDownFcn', 'ice(''ice_WindowButtonDownFcn'', gcbo, [],...
                        guidata(gcbo))',...
'WindowButtonMotionFcn', 'ice(''ice_WindowButtonMotionFcn'', gcbo,...
                        [], guidata(gcbo))',...
'WindowButtonUpFcn', 'ice(''ice_WindowButtonUpFcn'', gcbo, [],...
                        guidata(gcbo))',...
'HandleVisibility', 'callback',...
'Tag', 'ice',...
'UserData',[],...
'CreateFcn', {@local_CreateFcn, blanks(0), appdata} );
```

to create the main figure window. GUI objects are then added with statements like

```
h11 = uicontrol(...
'Parent',h1,...
'Units','normalized',...
'Callback',mat{5},...
'FontSize',10,...
'ListboxTop',0,...
'Position',[0.710583153347732 0.508951406649616 0.211663066954644
0.0767263427109974],...
'String','Reset',...
'Tag','reset_pushbutton',...
'CreateFcn', {@local_CreateFcn, blanks(0), appdata} );
```

Function `uicontrol ('PropertyName1', Value1, ...)` creates a user interface control in the current window with the specified properties and returns a handle to it.

which adds the `Reset` pushbutton to the figure. Note that these statements specify explicitly properties that were defined originally using the Property Inspector of the GUIDE Layout Editor. Finally, we note that the `figure` function was introduced in Section 2.3; `uicontrol` creates a user interface control (i.e., GUI object) in the current figure window based on property name/value pairs (e.g., `'Tag'` plus `'reset_pushbutton'`) and returns a handle to it.

B.2.2 The Opening and Output Functions

The first two functions following the initialization block in the starting GUI M-file at the beginning of Section B.2 are called *opening* and *output functions*,

respectively. They contain the code that is executed just before the GUI is made visible to the user and when the GUI returns its output to the command line or calling routine. Both functions are passed arguments `hObject`, `eventdata`, and `handles`. (These arguments are also inputs to the callback functions in the next two sections.) Input `hObject` is a graphics object handle, `eventdata` is reserved for future use, and `handles` is a structure that provides handles to interface objects and any application specific or user defined data. To implement the desired functionality of the ICE interface (see the Help text), both `ice_OpeningFcn` and `ice_OutputFcn` must be expanded beyond the "barebones" versions in the starting GUI M-file. The expanded code is as follows:

`ice_OpeningFcn`

From the final M-file.

```
%--------------------------------------------------------------------%
function ice_OpeningFcn(hObject, eventdata, handles, varargin)
%  When ICE is opened, perform basic initialization (e.g., setup
%  globals, ...) before it is made visible.

% Set ICE globals to defaults.
handles.updown = 'none';          % Mouse updown state
handles.plotbox = [0 0 1 1];      % Plot area parameters in pixels
handles.set1 = [0 0; 1 1];        % Curve 1 control points
handles.set2 = [0 0; 1 1];        % Curve 2 control points
handles.set3 = [0 0; 1 1];        % Curve 3 control points
handles.set4 = [0 0; 1 1];        % Curve 4 control points
handles.curve = 'set1';           % Structure name of selected curve
handles.cindex = 1;               % Index of selected curve
handles.node = 0;                 % Index of selected control point
handles.below = 1;                % Index of node below control point
handles.above = 2;                % Index of node above control point
handles.smooth = [0; 0; 0; 0];    % Curve smoothing states
handles.slope = [0; 0; 0; 0];     % Curve end slope control states
handles.cdf = [0; 0; 0; 0];       % Curve CDF states
handles.pdf = [0; 0; 0; 0];       % Curve PDF states
handles.output = [];              % Output image handle
handles.df = [];                  % Input PDFs and CDFs
handles.colortype = 'rgb';        % Input image color space
handles.input = [];               % Input image data
handles.imagemap = 1;             % Image map enable
handles.barmap = 1;               % Bar map enable
handles.graybar = [];             % Pseudo (gray) bar image
handles.colorbar = [];            % Color (hue) bar image

% Process Property Name/Property Value input argument pairs.
wait = 'on';
if (nargin > 3)
   for i = 1:2:(nargin - 3)
      if nargin - 3 == i
         break;
      end
      switch lower(varargin{i})
```

```
      case 'image'
         if ndims(varargin{i + 1}) == 3
            handles.input = varargin{i + 1};
         elseif ndims(varargin{i + 1}) == 2
            handles.input = cat(3, varargin{i + 1}, ...
                                varargin{i + 1}, varargin{i + 1});
         end
         handles.input = double(handles.input);
         inputmax = max(handles.input(:));
         if inputmax > 255
            handles.input = handles.input / 65535;
         elseif inputmax > 1
            handles.input = handles.input / 255;
         end

      case 'space'
         handles.colortype = lower(varargin{i + 1});
         switch handles.colortype
         case 'cmy'
            list = {'CMY' 'Cyan' 'Magenta' 'Yellow'};
         case {'ntsc', 'yiq'}
            list = {'YIQ' 'Luminance' 'Hue' 'Saturation'};
            handles.colortype = 'ntsc';
         case 'ycbcr'
            list = {'YCbCr' 'Luminance' 'Blue' ...
                    'Difference'  'Red Difference'};
         case 'hsv'
            list = {'HSV' 'Hue' 'Saturation' 'Value'};
         case 'hsi'
            list = {'HSI' 'Hue' 'Saturation' 'Intensity'};
         otherwise
            list = {'RGB' 'Red' 'Green' 'Blue'};
            handles.colortype = 'rgb';
         end
         set(handles.component_popup, 'String', list);

      case 'wait'
         wait = lower(varargin{i + 1});
      end
   end
end

% Create pseudo- and full-color mapping bars (grays and hues). Store
% a color space converted 1x128x3 line of each bar for mapping.
xi = 0:1/127:1;      x = 0:1/6:1;      x = x';
y = [1 1 0 0 0 1 1; 0 1 1 1 0 0 0; 0 0 0 1 1 1 0]';
gb = repmat(xi, [1 1 3]);        cb = interp1q(x, y, xi');
cb = reshape(cb, [1 128 3]);
if ~strcmp(handles.colortype, 'rgb')
   gb = eval(['rgb2' handles.colortype '(gb)']);
   cb = eval(['rgb2' handles.colortype '(cb)']);
end
```

```
gb = round(255 * gb);     gb = max(0, gb);      gb = min(255, gb);
cb = round(255 * cb);     cb = max(0, cb);      cb = min(255, cb);
handles.graybar = gb;     handles.colorbar = cb;

% Do color space transforms, clamp to [0, 255], compute histograms
% and cumulative distribution functions, and create output figure.
if size(handles.input, 1)
   if ~strcmp(handles.colortype, 'rgb')
      handles.input = eval(['rgb2' handles.colortype ...
                            '(handles.input)']);
   end
   handles.input = round(255 * handles.input);
   handles.input = max(0, handles.input);
   handles.input = min(255, handles.input);
   for i = 1:3
      color = handles.input(:, :, i);
      df = hist(color(:), 0:255);
      handles.df = [handles.df; df / max(df(:))];
      df = df / sum(df(:));   df = cumsum(df);
      handles.df = [handles.df; df];
   end
   figure;     handles.output = gcf;
end

% Compute ICE's screen position and display image/graph.
set(0, 'Units', 'pixels');      ssz = get(0, 'Screensize');
set(handles.ice, 'Units', 'pixels');
uisz = get(handles.ice, 'Position');
if size(handles.input, 1)
   fsz = get(handles.output, 'Position');
   bc = (fsz(4) - uisz(4)) / 3;
   if bc > 0
      bc = bc + fsz(2);
   else
      bc = fsz(2) + fsz(4) - uisz(4) - 10;
   end
   lc = fsz(1) + (size(handles.input, 2) / 4) + (3 * fsz(3) / 4);
   lc = min(lc, ssz(3) - uisz(3) - 10);
   set(handles.ice, 'Position', [lc bc 463 391]);
else
   bc = round((ssz(4) - uisz(4)) / 2) - 10;
   lc = round((ssz(3) - uisz(3)) / 2) - 10;
   set(handles.ice, 'Position', [lc bc uisz(3) uisz(4)]);
end
set(handles.ice, 'Units', 'normalized');
graph(handles);     render(handles);

% Update handles and make ICE wait before exit if required.
guidata(hObject, handles);
if strcmpi(wait, 'on')
   uiwait(handles.ice);
```

```
end

%--------------------------------------------------------------------%
function varargout = ice_OutputFcn(hObject, eventdata, handles)
%  After ICE is closed, get the image data of the current figure
%  for the output. If 'handles' exists, ICE isn't closed (there was
%  no 'uiwait') so output figure handle.

if max(size(handles)) == 0
   figh = get(gcf);
   imageh = get(figh.Children);
   if max(size(imageh)) > 0
      image = get(imageh.Children);
      varargout{1} = image.CData;
   end
else
   varargout{1} = hObject;
end
```

ice_OutputFcn

From the final M-file.

Rather than examining the intricate details of these functions (see the code's comments and consult Appendix A or the index for help on specific functions), we note the following commonalities with most GUI opening and output functions:

1. The `handles` structure (as can be seen from its numerous references in the code) plays a central role in most GUI M-files. It serves two crucial functions. Since it provides handles for all the graphic objects in the interface, it can be used to access and modify object properties. For instance, the `ice` opening function uses

   ```
   set(handles.ice, 'Units', 'pixels');
   uisz = get(handles.ice, 'Position');
   ```

   to access the size and location of the ICE GUI (in pixels). This is accomplished by setting the `Units` property of the `ice` figure, whose handle is available in `handles.ice`, to `'pixels'` and then reading the `Position` property of the figure (using the `get` function). The `get` function, which returns the value of a property associated with a graphics object, is also used to obtain the computer's display area via the `ssz = get(0, 'Screensize')` statement near the end of the opening function. Here, `0` is the handle of the computer display (i.e., root figure) and `'Screensize'` is a property containing its extent.

   In addition to providing access to GUI objects, the `handles` structure is a powerful conduit for sharing application data. Note that it holds the default values for twenty-three global `ice` parameters (ranging from the mouse state in `handles.updown` to the entire input image in `handles.input`). They must survive every call to `ice` and are added to `handles` at the start of `ice_OpeningFcn`. For instance, the `handles.set1` global is created by the statement

```
handles.set1 = [0 0; 1 1]
```

where `set1` is a named field containing the control points of a color mapping function to be added to the `handles` structure and `[0 0; 1 1]` is its default value [curve endpoints (0, 0) and (1, 1)]. Before exiting a function in which `handles` is modified,

Function `guidata` (`H, DATA`) stores the specified data in the figure's application data. `H` is a handle that identifies the figure—it can be the figure itself, or any object contained in the figure.

```
guidata(hObject, handles)
```

must be called to store variable `handles` as the application data of the figure with handle `hObject`.

2. Like many built-in graphics functions, `ice_OpeningFcn` processes input arguments (except `hObject`, `eventdata`, and `handles`) in property name and value pairs. When there are more than three input arguments (i.e., if `nargin > 3`), a loop that skips through the input arguments in pairs [`for i = 1:2:(nargin - 3)`] is executed. For each pair of inputs, the first is used to drive the `switch` construct,

```
switch lower(varargin{i})
```

which processes the second parameter appropriately. For `case 'space'`, for instance, the statement

```
handles.colortype = lower(varargin{i + 1});
```

sets named field `colortype` to the value of the second argument of the input pair. This value is then used to setup ICE's color component popup options (i.e., the `String` property of object `component_popup`). Later, it is used to transform the components of the input image to the desired mapping space via

```
handles.input = eval(['rgb2' ...
    handles.colortype '(handles.input)']);
```

where built-in function `eval(s)` causes MATLAB to execute string `s` as an expression or statement (see Section 13.4.1 for more on function `eval`). If `handles.input` is `'hsv'`, for example, `eval` argument `['rgb2' 'hsv' '(handles.input)']` becomes the concatenated string `'rgb2hsv(handles.input)'`, which is executed as a standard MATLAB expression that transforms the RGB components of the input image to the HSV color space (see Section 7.2.3).

3. The statement

```
% uiwait(handles.figure1);
```

in the starting GUI M-file is converted into the conditional statement

```
if strcmpi(wait, 'on') uiwait(handles.ice); end
```

in the final version of `ice_OpeningFcn`. In general,


```
uiwait(fig)
```

blocks execution of a MATLAB code stream until either a `uiresume` is executed or figure `fig` is destroyed (i.e., closed). [With no input arguments, `uiwait` is the same as `uiwait(gcf)` where MATLAB function `gcf` returns the handle of the current figure]. When `ice` is not expected to return a mapped version of an input image, but return immediately (i.e., before the ICE GUI is closed), an input property name/value pair of `'wait'/'off'` must be included in the call. Otherwise, ICE will not return to the calling routine or command line until it is closed—that is, until the user is finished interacting with the interface (and color mapping functions). In this situation, function `ice_OutputFcn` can not obtain the mapped image data from the `handles` structure, because it does not exist after the GUI is closed. As can be seen in the final version of the function, ICE extracts the image data from the `CData` property of the surviving mapped image output figure. If a mapped output image is not to be returned by `ice`, the `uiwait` statement in `ice_OpeningFcn` is not executed, `ice_OutputFcn` is called immediately after the opening function (long before the GUI is closed), and the handle of the mapped image output figure is returned to the calling routine or command line.

Finally, we note that several internal functions are invoked by `ice_OpeningFcn`. These—and all other `ice` internal functions—are listed next. Note that they provide additional examples of the usefulness of the `handles` structure in MATLAB GUIs. For instance, the

```
nodes = getfield(handles, handles.curve)
```

and

```
nodes = getfield(handles, ['set' num2str(i)])
```

statements in internal functions `graph` and `render`, respectively, are used to access the interactively defined control points of ICE's various color mapping curves. In its standard form,


```
F = getfield(S,'field')
```

returns to `F` the contents of named field `'field'` from structure `S`.

```
%-------------------------------------------------------------------%
function graph(handles)
%  Interpolate and plot mapping functions and optional reference
%  PDF(s) or CDF(s).

nodes = getfield(handles, handles.curve);
```

```
c = handles.cindex;      dfx = 0:1/255:1;
colors = ['k' 'r' 'g' 'b'];

% For piecewise linear interpolation, plot a map, map + PDF/CDF, or
% map + 3 PDFs/CDFs.
if ~handles.smooth(handles.cindex)
   if (~handles.pdf(c) && ~handles.cdf(c)) || ...
         (size(handles.df, 2) == 0)
      plot(nodes(:, 1), nodes(:, 2), 'b-', ...
           nodes(:, 1),  nodes(:, 2), 'ko', ...
           'Parent', handles.curve_axes);
   elseif c > 1
      i = 2 * c - 2 - handles.pdf(c);
      plot(dfx, handles.df(i, :), [colors(c) '-'], ...
           nodes(:, 1), nodes(:, 2), 'k-', ...
           nodes(:, 1), nodes(:, 2), 'ko', ...
           'Parent', handles.curve_axes);
   elseif c == 1
      i = handles.cdf(c);
      plot(dfx, handles.df(i + 1, :), 'r-', ...
           dfx, handles.df(i + 3, :), 'g-',  ...
           dfx, handles.df(i + 5, :), 'b-', ...
           nodes(:, 1), nodes(:, 2), 'k-',  ...
           nodes(:, 1), nodes(:, 2), 'ko', ...
           'Parent', handles.curve_axes);
   end

% Do the same for smooth (cubic spline) interpolations.
else
   x = 0:0.01:1;
   if ~handles.slope(handles.cindex)
      y = spline(nodes(:, 1), nodes(:, 2), x);
   else
      y = spline(nodes(:, 1), [0; nodes(:, 2); 0], x);
   end
   i = find(y > 1);        y(i) = 1;
   i = find(y < 0);        y(i) = 0;

   if (~handles.pdf(c) && ~handles.cdf(c)) || ...
         (size(handles.df, 2) == 0)
      plot(nodes(:, 1), nodes(:, 2), 'ko',  x, y, 'b-', ...
           'Parent', handles.curve_axes);
   elseif c > 1
      i = 2 * c - 2 - handles.pdf(c);
      plot(dfx, handles.df(i, :), [colors(c) '-'], ...
           nodes(:, 1), nodes(:, 2), 'ko', x, y, 'k-', ...
           'Parent', handles.curve_axes);
   elseif c == 1
      i = handles.cdf(c);
      plot(dfx, handles.df(i + 1, :), 'r-', ...
           dfx, handles.df(i + 3, :), 'g-', ...
           dfx, handles.df(i + 5, :), 'b-', ...
```

```
            nodes(:, 1), nodes(:, 2), 'ko', x, y, 'k-', ...
            'Parent', handles.curve_axes);
   end
end

% Put legend if more than two curves are shown.
s = handles.colortype;
if strcmp(s, 'ntsc')
    s = 'yiq';
end
if (c == 1) && (handles.pdf(c) || handles.cdf(c))
   s1 = ['-- ' upper(s(1))];
   if length(s) == 3
      s2 = ['-- ' upper(s(2))];       s3 = ['-- ' upper(s(3))];
   else
      s2 = ['-- ' upper(s(2)) s(3)];  s3 = ['-- ' upper(s(4)) s(5)];
   end
else
   s1 = '';    s2 = '';    s3 = '';
end
set(handles.red_text, 'String', s1);
set(handles.green_text, 'String', s2);
set(handles.blue_text, 'String', s3);

%--------------------------------------------------------------------%
function [inplot, x, y] = cursor(h, handles)
%  Translate the mouse position to a coordinate with respect to
%  the current plot area, check for the mouse in the area and if so
%  save the location and write the coordinates below the plot.

set(h, 'Units', 'pixels');
p = get(h, 'CurrentPoint');
x = (p(1, 1) - handles.plotbox(1)) / handles.plotbox(3);
y = (p(1, 2) - handles.plotbox(2)) / handles.plotbox(4);
if x > 1.05 || x < -0.05 || y > 1.05 || y < -0.05
   inplot = 0;
else
   x = min(x, 1);      x = max(x, 0);
   y = min(y, 1);      y = max(y, 0);
   nodes = getfield(handles, handles.curve);
   x = round(256 * x) / 256;
   inplot = 1;
   set(handles.input_text, 'String', num2str(x, 3));
   set(handles.output_text, 'String', num2str(y, 3));
end
set(h, 'Units', 'normalized');

%--------------------------------------------------------------------%
function y = render(handles)
%  Map the input image and bar components and convert them to RGB
%  (if needed) and display.

set(handles.ice, 'Interruptible', 'off');
```

```
set(handles.ice, 'Pointer', 'watch');
ygb = handles.graybar;      ycb = handles.colorbar;
yi = handles.input;         mapon = handles.barmap;
imageon = handles.imagemap & size(handles.input, 1);

for i = 2:4
   nodes = getfield(handles, ['set' num2str(i)]);
   t = lut(nodes, handles.smooth(i), handles.slope(i));
   if imageon
      yi(:, :, i - 1) = t(yi(:, :, i - 1) + 1);
   end
   if mapon
      ygb(:, :, i - 1) = t(ygb(:, :, i - 1) + 1);
      ycb(:, :, i - 1) = t(ycb(:, :, i - 1) + 1);
   end
end
t = lut(handles.set1, handles.smooth(1), handles.slope(1));
if imageon
   yi = t(yi + 1);
end
if mapon
   ygb = t(ygb + 1);     ycb = t(ycb + 1);
end

if ~strcmp(handles.colortype, 'rgb')
   if size(handles.input, 1)
      yi = yi / 255;
      yi = eval([handles.colortype '2rgb(yi)']);
      yi = uint8(255 * yi);
   end
   ygb = ygb / 255;        ycb = ycb / 255;
   ygb = eval([handles.colortype '2rgb(ygb)']);
   ycb = eval([handles.colortype '2rgb(ycb)']);
   ygb = uint8(255 * ygb);     ycb = uint8(255 * ycb);
else
   yi = uint8(yi);   ygb = uint8(ygb);     ycb = uint8(ycb);
end

if size(handles.input, 1)
   figure(handles.output);    imshow(yi);
end
ygb = repmat(ygb, [32 1 1]);    ycb = repmat(ycb, [32 1 1]);
axes(handles.gray_axes);        imshow(ygb);
axes(handles.color_axes);       imshow(ycb);
figure(handles.ice);
set(handles.ice, 'Pointer', 'arrow');
set(handles.ice, 'Interruptible', 'on');

%-------------------------------------------------------------------%
function t = lut(nodes, smooth, slope)
%  Create a 256 element mapping function from a set of control
%  points. The output values are integers in the interval [0, 255].
```

```
%  Use piecewise linear or cubic spline with or without zero end
%  slope interpolation.

t = 255 * nodes;    i = 0:255;
if ~smooth
   t = [t; 256 256];   t = interp1q(t(:, 1), t(:, 2), i');
else
   if ~slope
      t = spline(t(:, 1), t(:, 2), i);
   else
      t = spline(t(:, 1), [0; t(:, 2); 0], i);
   end
end
t = round(t);       t = max(0, t);          t = min(255, t);

%--------------------------------------------------------------------%
function out = spreadout(in)
%  Make all x values unique.

% Scan forward for non-unique x's and bump the higher indexed x--
% but don't exceed 1. Scan the entire range.
nudge = 1 / 256;
for i = 2:size(in, 1) - 1
   if in(i, 1) <= in(i - 1, 1)
      in(i, 1) = min(in(i - 1, 1) + nudge, 1);
   end
end

% Scan in reverse for non-unique x's and decrease the lower indexed
% x -- but don't go below 0. Stop on the first non-unique pair.
if in(end, 1) == in(end - 1, 1)
   for i = size(in, 1):-1:2
      if in(i, 1) <= in(i - 1, 1)
         in(i - 1, 1) = max(in(i, 1) - nudge, 0);
      else
         break;
      end
   end
end

% If the first two x's are now the same, init the curve.
if in(1, 1) == in(2, 1)
   in = [0 0; 1 1];
end
out = in;

%--------------------------------------------------------------------%
function g = rgb2cmy(f)
%   Convert RGB to CMY using IPT's imcomplement.

g = imcomplement(f);

%--------------------------------------------------------------------%
function g = cmy2rgb(f)
```

```
%   Convert CMY to RGB using IPT's imcomplement.
g = imcomplement(f);
```

B.2.3 Figure Callback Functions

The three functions immediately following the ICE opening and closing functions in the starting GUI M-file at the beginning of Section B.2 are *figure callbacks* `ice_WindowButtonDownFcn`, `ice_WindowButtonMotionFcn`, and `ice_WindowButtonUpFcn`. In the automatically generated M-file, they are *function stubs*—that is, MATLAB `function` definition statements without supporting code. Fully developed versions of the three functions, whose joint task is to process mouse events (clicks and drags of mapping function control points on ICE's `curve_axes` object), are as follows:

ice
Figure Callbacks

```
%-------------------------------------------------------------------%
function ice_WindowButtonDownFcn(hObject, eventdata, handles)
%  Start mapping function control point editing. Do move, add, or
%  delete for left, middle, and right button mouse clicks ('normal',
%  'extend', and 'alt' cases) over plot area.

set(handles.curve_axes, 'Units', 'pixels');
handles.plotbox = get(handles.curve_axes, 'Position');
set(handles.curve_axes, 'Units', 'normalized');
[inplot, x, y] = cursor(hObject, handles);
if inplot
   nodes = getfield(handles, handles.curve);
   i = find(x >= nodes(:, 1));      below = max(i);
   above = min(below + 1, size(nodes, 1));
   if (x - nodes(below, 1)) > (nodes(above, 1) - x)
      node = above;
   else
      node = below;
   end
   deletednode = 0;

   switch get(hObject, 'SelectionType')
   case 'normal'
      if node == above
         above = min(above + 1, size(nodes, 1));
      elseif node == below
         below = max(below - 1, 1);
      end
      if node == size(nodes, 1)
         below = above;
      elseif node == 1
         above = below;
      end
      if x > nodes(above, 1)
         x = nodes(above, 1);
      elseif x < nodes(below, 1)
```

```
            x = nodes(below, 1);
        end
        handles.node = node;      handles.updown = 'down';
        handles.below = below;    handles.above = above;
        nodes(node, :) = [x y];
    case 'extend'
        if ~any(nodes(:, 1) == x)
            nodes = [nodes(1:below, :); [x y]; nodes(above:end, :)];
            handles.node = above;    handles.updown = 'down';
            handles.below = below;   handles.above = above + 1;
        end
    case 'alt'
        if (node ~= 1) && (node ~= size(nodes, 1))
            nodes(node, :) = [];    deletednode = 1;
        end
        handles.node = 0;
        set(handles.input_text, 'String', '');
        set(handles.output_text, 'String', '');
    end
    handles = setfield(handles, handles.curve, nodes);
    guidata(hObject, handles);
    graph(handles);
    if deletednode
        render(handles);
    end
end

%--------------------------------------------------------------------%
function ice_WindowButtonMotionFcn(hObject, eventdata, handles)
%  Do nothing unless a mouse 'down' event has occurred. If it has,
%  modify control point and make new mapping function.

if ~strcmpi(handles.updown, 'down')
    return;
end
[inplot, x, y] = cursor(hObject, handles);
if inplot
    nodes = getfield(handles, handles.curve);
    nudge = handles.smooth(handles.cindex) / 256;
    if (handles.node ~= 1) && (handles.node ~= size(nodes, 1))
        if x >= nodes(handles.above, 1)
            x = nodes(handles.above, 1) - nudge;
        elseif x <= nodes(handles.below, 1)
            x = nodes(handles.below, 1) + nudge;
        end
    else
        if x > nodes(handles.above, 1)
            x = nodes(handles.above, 1);
        elseif x < nodes(handles.below, 1)
            x = nodes(handles.below, 1);
        end
    end
```

Functions `S = setfield(S, 'field', V)` sets the contents of the specified field to value `V`. The changed structure is returned.

```
   nodes(handles.node, :) = [x y];
   handles = setfield(handles, handles.curve, nodes);
   guidata(hObject, handles);
   graph(handles);
end

%--------------------------------------------------------------------%
function ice_WindowButtonUpFcn(hObject, eventdata, handles)
%  Terminate ongoing control point move or add operation. Clear
%  coordinate text below plot and update display.

update = strcmpi(handles.updown, 'down');
handles.updown = 'up';      handles.node = 0;
guidata(hObject, handles);
if update
   set(handles.input_text, 'String', '');
   set(handles.output_text, 'String', '');
   render(handles);
end
```

In general, figure callbacks are launched in response to interactions with a figure object or window—not an active `uicontrol` object. More specifically,

- The `WindowButtonDownFcn` is executed when a user clicks a mouse button with the cursor in a figure but not over an enabled `uicontrol` (e.g., a pushbutton or popup menu).
- The `WindowButtonMotionFcn` is executed when a user moves a depressed mouse button within a figure window.
- The `WindowButtonUpFcn` is executed when a user releases a mouse button, after having pressed the mouse button within a figure but not over an enabled `uicontrol`.

The purpose and behavior of `ice`'s figure callbacks are documented (via comments) in the code. We make the following general observations about the final implementations:

1. Because the `ice_WindowButtonDownFcn` is called on all mouse button clicks in the `ice` figure (except over an active graphic object), the first job of the callback function is to see if the cursor is within `ice`'s plot area (i.e., the extent of the `curve_axes` object). If the cursor is outside this area, the mouse should be ignored. The test for this is performed by internal function `cursor`, whose listing was provided in the previous section. In cursor, the statement

   ```
   p = get(h, 'CurrentPoint');
   ```

   returns the current cursor coordinates. Variable `h` is passed from `ice_WindowButtonDownFcn` and originates as input argument `hObject`. In all figure callbacks, `hObject` is the handle of the figure requesting service.

Property `'CurrentPoint'` contains the position of the cursor relative to the figure as a two-element row vector [*x y*].

2. Since `ice` is designed to work with two- and three-button mice, `ice_WindowButtonDownFcn` must determine which mouse button causes each callback. As can be seen in the code, this is done with a `switch` construct using the figure's `'SelectionType'` property. Cases `'normal'`, `'extent'`, and `'alt'` correspond to the left, middle, and right button clicks on three-button mice (or the left, shift-left, and control-left clicks of two-button mice), respectively, and are used to trigger the add control point, move control point, and delete control point operations.

3. The displayed ICE mapping function is updated (via internal function `graph`) each time a control point is modified, but the output figure, whose handle is stored in `handles.output`, is updated on *mouse button releases only*. This is because the computation of the output image, which is performed by internal function `render`, can be time-consuming. It involves mapping separately the input image's three color components, remapping each by the "all-component" curve, and converting the mapped components to the RGB color space for display. Note that without adequate precautions, the mapping function's control points could be modified inadvertently during this lengthy output mapping process.

   To prevent this, `ice` controls the interruptibility of its various callbacks. All MATLAB graphics objects have an `Interruptible` property that determines whether their callbacks can be interrupted. The default value of every object's `'Interruptible'` property is `'on'`, which means that object callbacks can be interrupted. If switched to `'off'`, callbacks that occur during the execution of the *now* noninterruptible callback are either ignored (i.e., cancelled) or placed in an *event queue* for later processing. The disposition of the interrupting callback is determined by the `'BusyAction'` property of the object being interrupted. If `'BusyAction'` is `'cancel'`, the callback is discarded; if `'queue'`, the callback is processed after the noninterruptible callback finishes.

   The `ice_WindowButtonUpFcn` function uses the mechanism just described to suspend temporarily (i.e., during output image computations) the user's ability to manipulate mapping function control points. The sequence

```
set(handles.ice, 'Interruptible', 'off');
set(handles.ice, 'Pointer', 'watch');

set(handles.ice, 'Pointer', 'arrow');
set(handles.ice, 'Interruptible', 'on');
```

   in internal function `render` sets the `ice` figure window's `'Interruptible'` property to `'off'` during the mapping of the output image and pseudo- and full-color bars. This prevents users from modifying mapping function control points while a mapping is being performed. Note also that the figure's

'Pointer' property is set to 'watch' to indicate visually that ice is busy and reset to 'arrow' when the output computation is completed.

B.2.4 Object Callback Functions

The final fourteen lines (i.e., ten functions) of the starting GUI M-file at the beginning of Section B.2 are *object callback* function stubs. Like the automatically generated figure callbacks of the previous section, they are initially void of code. Fully developed versions of the functions follow. Note that each function processes user interaction with a different ice uicontrol object (pushbutton, etc.) and is named by concatenating its Tag property with string '_Callback'. For example, the callback function responsible for handling the selection of the displayed mapping function is named the component_popup_Callback. It is called when the user activates (i.e., clicks on) the popup selector. Note also that input argument hObject is the handle of the popup graphics object—not the handle of the ice figure (as in the figure callbacks of the previous section). ICE's object callbacks involve minimal code and are self-documenting. Because ice does not use context-sensitive (i.e., right-click initiated) menus, function stub component_popup_CreateFcn is left in its intially void state. It is a callback routine that is executed during object creation.

ice
Object Callbacks

```
%--------------------------------------------------------------------%
function smooth_checkbox_Callback(hObject, eventdata, handles)
%  Accept smoothing parameter for currently selected color
%  component and redraw mapping function.

if get(hObject, 'Value')
   handles.smooth(handles.cindex) = 1;
   nodes = getfield(handles, handles.curve);
   nodes = spreadout(nodes);
   handles = setfield(handles, handles.curve, nodes);
else
   handles.smooth(handles.cindex) = 0;
end
guidata(hObject, handles);
set(handles.ice, 'Pointer', 'watch');
graph(handles);      render(handles);
set(handles.ice, 'Pointer', 'arrow');

%--------------------------------------------------------------------%
function reset_pushbutton_Callback(hObject, eventdata, handles)
%  Init all display parameters for currently selected color
%  component, make map 1:1, and redraw it.

handles = setfield(handles, handles.curve, [0 0; 1 1]);
c = handles.cindex;
handles.smooth(c) = 0;  set(handles.smooth_checkbox, 'Value', 0);
handles.slope(c) = 0;   set(handles.slope_checkbox, 'Value', 0);
handles.pdf(c) = 0;     set(handles.pdf_checkbox, 'Value', 0);
```

```
handles.cdf(c) = 0;     set(handles.cdf_checkbox, 'Value', 0);
guidata(hObject, handles);
set(handles.ice, 'Pointer', 'watch');
graph(handles);      render(handles);
set(handles.ice, 'Pointer', 'arrow');

%--------------------------------------------------------------------%
function slope_checkbox_Callback(hObject, eventdata, handles)
%  Accept slope clamp for currently selected color component and
%  draw function if smoothing is on.

if get(hObject, 'Value')
   handles.slope(handles.cindex) = 1;
else
   handles.slope(handles.cindex) = 0;
end
guidata(hObject, handles);
if handles.smooth(handles.cindex)
   set(handles.ice, 'Pointer', 'watch');
   graph(handles);      render(handles);
   set(handles.ice, 'Pointer', 'arrow');
end

%--------------------------------------------------------------------%
function resetall_pushbutton_Callback(hObject, eventdata, handles)
%  Init display parameters for color components, make all maps 1:1,
%  and redraw display.

for c = 1:4
   handles.smooth(c) = 0;      handles.slope(c) = 0;
   handles.pdf(c) = 0;         handles.cdf(c) = 0;
   handles = setfield(handles, ['set' num2str(c)], [0 0; 1 1]);
end
set(handles.smooth_checkbox, 'Value', 0);
set(handles.slope_checkbox, 'Value', 0);
set(handles.pdf_checkbox, 'Value', 0);
set(handles.cdf_checkbox, 'Value', 0);
guidata(hObject, handles);
set(handles.ice, 'Pointer', 'watch');
graph(handles);      render(handles);
set(handles.ice, 'Pointer', 'arrow');

%--------------------------------------------------------------------%
function pdf_checkbox_Callback(hObject, eventdata, handles)
%  Accept PDF (probability density function or histogram) display
%  parameter for currently selected color component and redraw
%  mapping function if smoothing is on. If set, clear CDF display.

if get(hObject, 'Value')
   handles.pdf(handles.cindex) = 1;
   set(handles.cdf_checkbox, 'Value', 0);
   handles.cdf(handles.cindex) = 0;
else
```

```
   handles.pdf(handles.cindex) = 0;
end
guidata(hObject, handles);      graph(handles);

%--------------------------------------------------------------------%
function cdf_checkbox_Callback(hObject, eventdata, handles)
%  Accept CDF (cumulative distribution function) display parameter
%  for selected color component and redraw mapping function if
%  smoothing is on. If set, clear CDF display.

if get(hObject, 'Value')
   handles.cdf(handles.cindex) = 1;
   set(handles.pdf_checkbox, 'Value', 0);
   handles.pdf(handles.cindex) = 0;
else
   handles.cdf(handles.cindex) = 0;
end
guidata(hObject, handles);      graph(handles);

%--------------------------------------------------------------------%
function mapbar_checkbox_Callback(hObject, eventdata, handles)
%  Accept changes to bar map enable state and redraw bars.

handles.barmap = get(hObject, 'Value');
guidata(hObject, handles);      render(handles);

%--------------------------------------------------------------------%
function mapimage_checkbox_Callback(hObject, eventdata, handles)
%  Accept changes to the image map state and redraw image.

handles.imagemap = get(hObject, 'Value');
guidata(hObject, handles);      render(handles);

%--------------------------------------------------------------------%
function component_popup_Callback(hObject, eventdata, handles)
%  Accept color component selection, update component specific
%  parameters on GUI, and draw the selected mapping function.

c = get(hObject, 'Value');
handles.cindex = c;
handles.curve = strcat('set', num2str(c));
guidata(hObject, handles);
set(handles.smooth_checkbox, 'Value', handles.smooth(c));
set(handles.slope_checkbox, 'Value', handles.slope(c));
set(handles.pdf_checkbox, 'Value', handles.pdf(c));
set(handles.cdf_checkbox, 'Value', handles.cdf(c));
graph(handles);

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

```
% Hint: popupmenu 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
```