Part IB Paper 8
Information Engineering Option

A – Photo Editing Course

Nick Kingsbury – www-sigproc.eng.cam.ac.uk/∼ngk
Given by Joan Lasenby in 2016: www-sigproc.eng.cam.ac.uk/∼jl

April 9, 2016

Lecture 1: Sections 1 & 2.
Lecture 2: Sections 3 & 4.
Lecture 3: Sections 5 & 6.
Lecture 4: Sections 7 & 8.
Lecture 5: Section 9.
Lecture 6: Examples Class

Contents

B – Image Features and Matching Course

Roberto Cipolla

C – Image Searching and Modelling using Machine Learning Methods

Roberto Cipolla

Recommended Textbooks

- A K Jain, **Fundamentals of Digital Image Processing**, Prentice-Hall, 1989.
- R C Gonzalez and R E Woods, **Digital Image Processing**, Addison Wesley, 1992.
- M Petrou and P Bosdogianni, **Image Processing: The Fundamentals**, John Wiley, 1999.

Matlab code for the Photo Editor is downloadable from NGK's website:
www-sigproc.eng.cam.ac.uk/∼ngk in the section **Downloadable teaching material**.

1 Program Framework for the Photo Editor

1.1 Introduction

On this course we shall be developing a photo-editor in Matlab, which has a number of the functionalities of current commercial packages such as Adobe Photo-Shop and Microsoft Digital Image Suite. Before we can get on to consider detailed algorithms for any of the functions, we must produce an image display framework and graphical user interface (GUI) to allow the functions to be called and controlled, and to show their results.

We have chosen to use Matlab as the basis for our system because it allows quite complicated image processing operations to be implemented with relatively small amounts of code, and it has the basic functions for providing a suitable GUI for the user to interact with the program.

We shall first describe the main script file for our editor.

1.2 The main script file: `ph_edit`

The code for the main script file `ph_edit.m` is shown in fig. 1.2. The editor is started by typing `ph_edit` in the command window of Matlab. Script files are different from function files in Matlab, in that no variables are passed as arguments when script files are called. The scripts just use the main workspace that is visible to the user (i.e. they are equivalent to typing commands at the command prompt). For GUIs with buttons and other controls it is often simplest if the controls create calls to script files rather than to functions.

The first 4 lines of code in `ph_edit.m` (excluding comment lines) clear the workspace and open a window for figure(1) (see fig. 1.1), of size 670×940 pixels (pels) with lower left corner at pel (80,30), which is suitable for a 1024×768 screen size with task bar on the left edge. This can be adjusted if required for other screen sizes or task bar locations – adjustment will be required for high resolution screens.

The next 2 lines of code use the Matlab function `uicontrol` to produce a prompt box of style **text**, and a control button of style **pushbutton** (the default), in the top left corner of the figure window. Both are initially blank. `promptbox` and `contrbutton` are **handles** to these two GUI controls which allow us to modify their modes of use later.

The next line uses another Matlab GUI function

```
uim = uimenu('label','Photo-edit');
```

to set up an extra pull-down menu, **Photo-edit**, to the right of **Help** on the menu-bar of the figure window. The handle for this menu is `uim`.

We can now add items to the new menu `uim`, using lines of the form

```
uimenu(uim,'label','Open ''Before''','call','ph_openfile');
```

which adds an item labelled **Open 'Before'** and defines its callback string (the command which is executed when this menu item is selected) to be `ph_openfile`. (We will discuss the meaning of this label and function shortly.) Note that in Matlab, double quotes are used to produce single quotes inside a string that is already within single quotes.

The first 5 items of the menu are concerned with opening and saving image files and moving them between two displayed images, labelled **Before** and **After**. We have chosen to display two images at any given time, so that the 'before' and 'after' versions of the current image processing operation can be seen simultaneously. This will be discussed below when we consider the script file `showimages.m`.

The last 9 menu items, of the form

```
uimenu(uim,'label','Crop','call','mode=''Init''; ph_crop','separator','on');
uimenu(uim,'label','Resize','call','mode=''Init''; ph_resize');
  :
```

provide all the main functions that we will be describing on this course, and each one calls a different script file, labelled `ph_...` . All of these script files are controlled by a string variable `mode`, which is set to **Init** when the function is called from this menu, but which may take other values when it is called by other buttons, set up during its initialisation phase. A separator bar is introduced above the first item in this part of the menu by setting its parameter **separator** to be **on**.

Finally if the **Before** image `xui` does not already exist, there is an automatic call to the script `ph_openfile` which prompts the user to select an input image file. Otherwise the existing images are displayed by a call to `showimages`. Both of these scripts are described below.

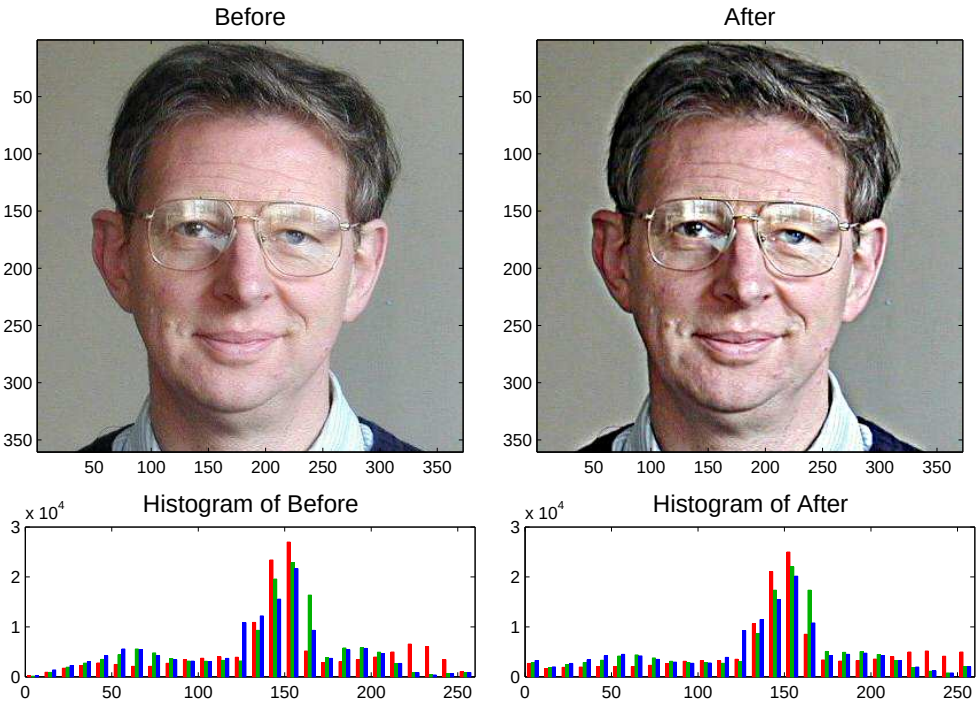


Fig. 1.1: Main window, figure(1), of the editor `ph_edit`.

```
% phedit.m - Photo Editing main program,
% for the Part IB Paper 8 Image Processing Option - Photo Editing.
%
% This script file opens the main figure window, sets up the GUI menu
% 'Image edit' and gets an input image file.
% Script files are used for this main program and also for the next level
% of functionality (files 'ph_...') in order to share the main workspace
% across all top-level functions.
%
% Nick Kingsbury, Cambridge University, 2006.

% Set up the main figure window to use most of a 1024x768 screen.
close all; % Close any open figures.
figure(1);
set(gcf,'position',[80  30   940   670]); % Change for other screen sizes.
set(gcf,'numbertitle','off','name','IB Paper 8 - Photo Editor'); % Title

% Set up a blank prompt box and a control button in the main window.
promptbox = uicontrol(gcf,'style','text','pos',[10 640 160 30]);
contrbutton = uicontrol(gcf,'pos',[30 620 120 20]);

% Set up 'Photo-edit' pull-down menu for fig 1.
uim = uimenu('label','Photo-edit');

% Add individual items to the menu.
uimenu(uim,'label','Open ''Before''','call','ph_openfile');
uimenu(uim,'label','Reopen ''Before''',...
    'call','xui = imread(infile); showimages');
uimenu(uim,'label','Save ''After''','call','ph_savefile');
uimenu(uim,'label','Copy ''Before'' to ''After''',...
    'call','yui=xui; newbefore=0; showimages');
uimenu(uim,'label','Copy ''After'' to ''Before''',...
    'call','xui=yui; newbefore=1; showimages');
% Put a separator line above the next item.
uimenu(uim,'label','Crop','call','mode=''Init''; ph_crop','separator','on');
uimenu(uim,'label','Resize','call','mode=''Init''; ph_resize');
uimenu(uim,'label','Rotate','call','mode=''Init''; ph_rotate');
uimenu(uim,'label','Morph','call','mode=''Init''; ph_morph');
uimenu(uim,'label','Colour shift','call','mode=''Init''; ph_colourshift');
uimenu(uim,'label','Lighting shift','call','mode=''Init''; ph_lightshift');
uimenu(uim,'label','Add noise','call','mode=''Init''; ph_addnoise');
uimenu(uim,'label','Filter','call','mode=''Init''; ph_filter');
uimenu(uim,'label','Enhance','call','mode=''Init''; ph_enhance');

% If xui does not exist, get an input image file; else show existing images.
if exist('xui')~=1, ph_openfile;
else newbefore=1; showimages;
end
```

Fig. 1.2: Photo Editor, main script file: `ph_edit.m`

1.3 Opening an input file: `ph_openfile`

The script to open an input file interactively is shown in fig. 1.3. It is called by the Photo-edit menu item **Open 'Before'** and also at the end of `ph_edit`.

The first two lines of code use the Matlab function `uigetfile` to open a GUI window so that the user can select an input image file of type .tif or .jpg . The directory path and filename are then concatenated into `infile`.

The Matlab function `imread` then reads the image file and saves it in `xui`, which is an $(m \times n \times 3)$ unsigned 8-bit integer (uint8) array. The 3 layers of the array store the red, green and blue components of the $(m \times n)$-pel colour image, using integers from 0 to 255 to represent the intensity of each colour component.

If the output image `yui` does not exist, `xui` is copied to it.

Finally the two images `xui` and `yui` are displayed as **Before** and **After** by a call to `showimages`. The variable `newbefore` is set to 1 so that `showimages` updates **Before** as well as **After**.

```
% ph_openfile.m
%
% Script to open a file for editing and display it.
%
% Nick Kingsbury, Cambridge University, 2006.

% Prompt user for a tif or jpeg filename.
[Filename,PathName] = uigetfile({'*.tif; *.jpg','Image files (*.tif, *.jpg)'},...
    'Open Image File');
infile = [PathName Filename];

% Read the file into (m x n x 3) unsigned integer array xui (the 'Before' image)
xui = imread(infile);

% Copy xui to yui ('After') if yui does not already exist.
if exist('yui')~=1, yui = xui; end

% Display xui and yui.
newbefore=1; showimages
```

Fig. 1.3: Script to open an input file: `ph_openfile.m`

1.4 Saving an output file: `ph_savefile`

The script to save an output file interactively is shown in fig. 1.4. It is called by the Photo-edit menu item **Save 'After'** and also when the **Close editor** button is pressed.

The first 5 lines of code produce a candidate output filename `outfile`, based on the input filename `infile`. They add `_a` to the input filename, unless it already has an ending like this, in which case the final character is incremented to the next letter of the alphabet.

Then the Matlab function `uiputfile` is called to allow the user to confirm or change this output filename or to cancel the save process. Finally, as long as `FileName` is non-zero, the Matlab function `imwrite` saves the **After** image `yui` in the format defined by the file extension (.tif or .jpg).

```
% ph_savefile.m
%
% Script to save the 'After' image to a file.
%
% Nick Kingsbury, Cambridge University, 2006.

% Suggest an output filename which has '_a' appended to the input name.
% If '_a' is already appended, change this to '_b' etc.
[pathstr,name,ext,versn] = fileparts(infile);
if name(end-1)~='_', name = [name '_a'];
else name(end) = name(end) + 1;
end
outfile = [pathstr  '\'  name  ext];

% Prompt user for an output filename.
[FileName,PathName] = uiputfile(outfile,'Save Image File');

% Write yui to the file if Filename is valid.
if FileName == 0, disp('''After'' image has not been saved.');
else imwrite(yui,[PathName FileName]);
end
```

Fig. 1.4: Script to save an output file: `ph_savefile.m`

1.5 Displaying the images: `showimages`

The script to display the images **Before** and **After**, together with their colour histograms, in the main figure window is shown in fig. 1.5. Colour histograms are a useful way of seeing whether a given image is using the full range of intensities and colours that are available, and of quantifying the effects of colour and lighting shifts that can be used to correct for poor exposure etc.

The code is divided into 4 main sections, to display the two image arrays, `xui` and `yui`, and their two histograms. If the variable `newbefore` is zero, then only `yui` and its histogram are updated, since it is assumed that `xui` has not changed. This results in a faster update, especially for larger images (> 0.5 Mpel).

The first section of code (executed if `newbefore` is true/non-zero) uses 2 Matlab functions: `subplot` to select axes in the upper left of the window; and `image` to plot the RGB-format uint8 array `xui` to create the **Before** image. The following three lines force the pels to be square, set the image title and force an immediate update of the display. The final two lines of this section of code set the default prompt message and control-button function.

The second section of code is always executed and creates the **After** image in much the same way as the first, in the upper right part of the window.

The third section of code calculates the histogram of `xui`, and plots it in the lower left of the window. First, the array `barcolour` defines the colours used for the three histogram bars in each bin, which represent the counts of red, green and blue pels within each bin range. The red and green bar intensities are reduced from unity, so that they appear with a similar subjective brightness to the blue bars.

To calculate separate histograms for RGB, the 3D array `xui` is reshaped into an $(mn \times 3)$ matrix `xh`, in which each column contains all the pels of a given colour. The Matlab function `hist` calculates the histograms of the 3 columns of `xh` (after conversion to data type double from uint8), and then `bar` plots the histograms and provides a vector `ha` of handles to them. The following `for` loop uses the handles to set the desired face and edge colour for each set of bars. We have chosen the histogram bins to be 10 units wide, centred on $4.5, 14.5 \cdots 254.5$. Finally the title is defined and the end limits for the x-axis are set to 0 and 260 (otherwise they would default to 0 and 300).

The fourth section of code calculates the histogram for `yui`, and plots it in the lower right of the window, similarly to the code for the histogram of `xui`.

Lastly, the zoom function is enabled and, if there is a second figure window enabled (e.g. for command buttons), it is brought to the foreground. Any additional figure windows are closed.

```
% showimages.m
%
% Script to display the 'Before' and 'After' images and their histograms.
%
% Nick Kingsbury, Cambridge University, 2006.

figure(1)

% Display the 'Before' image, if it has changed.
if newbefore,
    h1 = subplot('Position',[0.03 0.35 0.45 0.55]);
    xhandle = image(xui);
    axis image;
    title('Before','FontSize',14);
    drawnow;
    % Set the default prompt text and control button function.
    set(promptbox,'str','Choose ''Photo-edit'' menu-function');
    set(contrbutton,'str','Close editor','call','ph_savefile; close all');
end

% Display the 'After' image.
h2 = subplot('Position',[0.53 0.35 0.45 0.55]);
yhandle = image(yui);
axis image;
title('After','FontSize',14);
drawnow;

% Define bar colours for the histograms (red, dark green, blue).
barcolour = [0.9 0 0; 0 0.7 0; 0 0 1];

% Display the 'Before' histogram, if it has changed.
if newbefore,
    h3 = subplot('Position',[0.03 0.05 0.45 0.2]);
    sx = size(xui);
    % Reshape image to 3 columns of RGB values.
    xh = reshape(xui,prod(sx(1:2)),3);
    [nb,hout] = hist(double(xh),4.5:10:254.5); % Calculate histogram.
    ha = bar(hout,nb); % Plot histogram.
    % Set bar colours and make edge lines the same colour.
    for k=1:3,
      set(ha(k),'FaceColor',barcolour(k,:),'EdgeColor',barcolour(k,:));
    end
    title('Histogram of Before','FontSize',14)
    set(gca,'XLim',[0 260]); % Set x-axis limits.
    drawnow;
end
```

Fig. 1.5 continued overleaf.

```
% Display the 'After' histogram.
h4 = subplot('Position',[0.53 0.05 0.45 0.2]);
sy = size(yui);
yh = reshape(yui,prod(sy(1:2)),3);
[nb,hout] = hist(double(yh),4.5:10:254.5);
hb = bar(hout,nb);
for k=1:3,
    set(hb(k),'FaceColor',barcolour(k,:),'EdgeColor',barcolour(k,:));
end
title('Histogram of After','FontSize',14);
set(gca,'XLim',[0 260]);
drawnow;

% Enable zooming of all the images.
zoom on;

% Display command figure (2), if one exists.
figs = findobj('Type','figure'); % Find current figures.
if any(figs==2), figure(2); end % Display figure 2 as command figure.
if any(figs>2), close(figs(find(figs>2))); end % Close any extra figs.
```

Fig. 1.5: Script to show images: `showimages.m`

1.6 Other simple operations within `ph_edit`

Before pursuing the more sophisticated processes in the lower part of the Photo-edit menu, we briefly reconsider the top five menu options (above the separator bar). Referring back to the middle part of fig. 1.2, we may summarise them as follows:

Open 'Before': achieved by a call to `ph_openfile`.

Reopen 'Before': reads again from the input file, and is achieved by the callback string

```
xui = imread(infile); showimages
```

Save 'After': achieved by a call to `ph_savefile`.

Copy 'Before' to 'After': achieved by the callback string

```
yui=xui; newbefore=0; showimages
```

Copy 'After' to 'Before': achieved by the callback string

```
xui=yui; newbefore=1; showimages
```

2 Cropping the image: `ph_crop`

The script `ph_crop` is one of 9 script files which implement the main image processing functions within the Photo editor. They all have a similar format, in which the operation that is carried out when the script is called is controlled by `mode`, a string variable in the main workspace. For clarity, all the available operations, relevant to a particular processing function, are coded in the same script file and are selected by `mode` using the **switch ... case** syntax of the Matlab language. Matlab does not allow multiple scripts within a single file, so this is probably the most straightforward way to achieve this effect.

Those familiar with the switch function in C/C++ should note the following comment from the Matlab help page on switch:

> *Unlike the C language switch construct, the MATLAB switch does not fall through. That is, switch executes only the first matching case, subsequent matching cases do not execute. Therefore, break statements are not used.*

Fig. 2.1 shows the script `ph_crop`. Its purpose is to crop the image to a particular size by selecting a rectangle from within the **Before** image and keeping just the pels within the crop limits. It creates an **After** image that is in general smaller than the **Before** image. We provide two ways to define the crop rectangle: 1. by entering the min and max X and Y coordinates of the rectangle; or 2. by selecting a rectangle with the cursor using the Matlab zoom function.

In this file, there are 5 valid cases for `mode`, which are selected by the **switch** statement at the start:

Init initialises the **Crop** function. It defines the control button in figure 1 to call the **Get zoom** case (see below) and defines the prompt box accordingly. It opens a small command figure (2) near the top of the screen and, in the for-loop, sets up 4 edit boxes (with label boxes) for entering / displaying the crop limits numerically, and two buttons to **Crop now** and **Close** the function.

Set limits is called whenever an edit box is used, in order to convert the text in the four edit boxes to the values of the crop limits {Xmin Xmax Ymin Ymax}. The first for-loop converts the text to numerical values. Then these values are bounded so that {Xmin Ymin} lie between 1 and the size of the image minus 1, and {Xmax Ymax} lie between {Xmin+1 Ymin+1} and the size of the image. The second for-loop writes the bounded crop limits back into the edit boxes.

Get zoom is called when the control button in figure 1 is pressed. It gets the values of the **Before** image axis limits, using the axis handle `h1`. It is expected that these would have been set to the desired crop limits by the user operating the Matlab zoom mode within the figure window with his mouse. Then the values are bounded and written to the edit boxes as above.

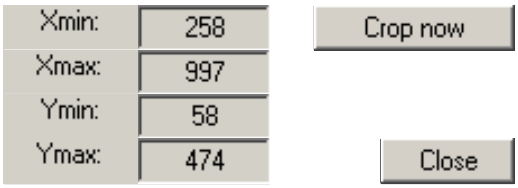
Crop now is called when this button in the figure 2 window is pressed. It first checks the edit boxes (as in **Set Limits**) to see if the crop limits have changed, and then performs the crop with

```
yui = xui(croplim(3):croplim(4),croplim(1):croplim(2),:);
```

which selects the relevant part of each layer of `xui` and saves it in `yui`. Then the **After** image is updated with `showimages`.

Close is called when this button in the figure 2 window is pressed. It closes figure(2) and redisplays **Before** and **After** in figure(1) to show the uncropped and cropped images respectively.

Command window, figure(2):




```
% ph_crop.m
%
% Script called by 'Crop' menu item in photo editor.
%
% Nick Kingsbury, Cambridge University, 2006.

switch mode

    case 'Init'
        % Set up control button in the main window to get zoom coordinates.
        figure(1);
        set(promptbox,'str',...
         'Zoom in to desired crop window in ''Before'' and click this button:');
        set(contrbutton,'str','Get zoom coordinates',...
            'call','mode=''Get zoom''; ph_crop');

        % Open new fig window near top of screen for entering params.
        figure(2);
        set(gcf,'position',[400  624  260  100]);
        set(gcf,'numbertitle','off','name','Crop Image');

        % Set default crop limits, [xmin xmax ymin ymax].
        if exist('croplim') ~= 1, croplim = round(axis(h1)); end

        label = {'Xmin:','Xmax:','Ymin:','Ymax:'};
        for k = 1:4
            edbox(k) = uicontrol(gcf,'style','edit',...
                'string',sprintf('%d',croplim(k)),'pos',[70 90-20*k 60 20],...
                'call','mode=''Set limits''; ph_crop');
            uicontrol(gcf,'style','text','str',label{k},...
                'pos',[10 90-20*k 60 20]);
        end
        cropnow = uicontrol(gcf,'str','Crop now','pos',[150 70 90 20],...
            'call','mode=''Crop now''; ph_crop');
        closebtn = uicontrol(gcf,'str','Close','pos',[180 10 60 20],...
            'call','mode=''Close''; ph_crop');
```

Fig. 2.1 continued overleaf.

```
case 'Set limits'
    % Get crop limits from edbox(1:4).
    for k=1:4,
        croplim(k) = round(sscanf(get(edbox(k),'string'),'%f'));
    end
    % Apply bounds and rewrite them into edbox(1:4).
    sx = size(xui);
    croplim([1 3]) = min(max(croplim([1 3]),1),sx([2 1])-1);
    croplim([2 4]) = max(min(croplim([2 4]),sx([2 1])),croplim([1 3])+1);
    for k=1:4,
        set(edbox(k),'string',sprintf('%d',croplim(k)));
    end

case 'Get zoom'
    % Get crop limits from current axis limits of 'Before' image.
    croplim = round(axis(h1));
    figure(2);
    % Apply bounds and write them into edbox(1:4).
    sx = size(xui);
    croplim([1 3]) = min(max(croplim([1 3]),1),sx([2 1])-1);
    croplim([2 4]) = max(min(croplim([2 4]),sx([2 1])),croplim([1 3])+1);
    for k=1:4,
        set(edbox(k),'string',sprintf('%d',croplim(k)));
    end

case 'Crop now'
    % Update limits in case edbox(1:4) have changed.
    for k=1:4,
        croplim(k) = round(sscanf(get(edbox(k),'string'),'%f'));
    end
    sx = size(xui);
    croplim([1 3]) = min(max(croplim([1 3]),1),sx([2 1])-1);
    croplim([2 4]) = max(min(croplim([2 4]),sx([2 1])),croplim([1 3])+1);
    for k=1:4,
        set(edbox(k),'string',sprintf('%d',croplim(k)));
    end
    % Crop 'Before' image and place result in 'After'.
    yui = xui(croplim(3):croplim(4),croplim(1):croplim(2),:);
    newbefore=0;
    showimages;

case 'Close'
    close(2);
    newbefore=1;
    showimages;

end
```

Fig. 2.1: Script to crop image: `ph_crop.m`

3 Resizing the image: `ph_resize`

The script `ph_resize` is concerned with resampling the image so that it becomes a different size (i.e. uses more or fewer pixels). This can be useful if multiple images are to be combined – perhaps in some sort of collage. In order to achieve resizing, we have written a function `im_resize()` which, at its heart, uses the 2-D interpolation function of Matlab `interp2()`. This is described in section 3.2. First we look at the script which implements the user interface.

3.1 The script `ph_resize`

This script adopts the same general form as `ph_crop`. It again comprises five cases selected by `mode`, as follows:

Init initialises the **Resize** function. It opens a command figure (2) near the top of the screen and initialises the new size for the **After** image to be the same as the current size (that of **Before**). The third element of `newsize` is a scale factor which relates the new size to the current size, and this is intialised to unity. The first for-loop sets up 3 edit boxes (with label boxes) for entering / displaying the scale factor and/or new X and Y sizes numerically, and the second loop inserts the sizes and scale factor in the boxes. Then a text box is set up to remind the user of the current X and Y sizes. Finally the buttons **Resize now** and **Close** are created.

Set size is called when the **X size** or **Y size** box is modified. The values from these two boxes are saved in `newsize(1:2)`. The values of `newsize(1:2)` are rewritten to these boxes and the scale box is blanked (as it is not meaningful if the two sizes are set independently).

Set scale is called when the **Scale** box is modified. Its value is saved in `newsize(3)`, and `newsize(1:2)` is updated with the new scale factor times the current size. These new values are then rewritten to the relevant boxes.

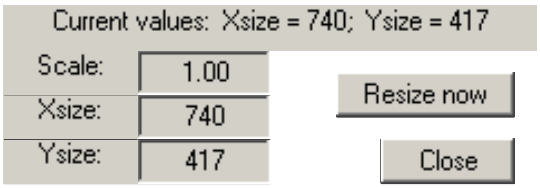
Resize now is called when this button is pressed. The values of **Y size** and **X size** are saved in `newsize(1:2)`, and the image is resized by

```
yui = im_resize(xui,newsize(1:2));
```

Then `showimages` displays the resized image `yui`.

Close is called when this button is pressed. It closes figure(2) and redisplays **Before** and **After** in figure(1) to show the current and resized images respectively.

Command window, figure(2):




```
% ph_resize.m
% Routine called by 'Resize' menu item in photo editor.
%
% Nick Kingsbury, Cambridge University, March 2006.

switch mode

    case 'Init'
        % Open new fig window for entering params
        figure(2);
        set(gcf,'position',[400  624  260  100]);
        set(gcf,'numbertitle','off','name','Resize Image');
        sx = size(xui);

        % Set default size to that of xui and default scale to 1.
        if exist('newsize') ~= 1, newsize = [sx(1:2) 1]; end

        label = {'Ysize:','Xsize:','Scale:'};
        for k = 1:3
          edbox(k) = uicontrol(gcf,'style','edit','pos',[70 20*k-10 60 20]);
          uicontrol(gcf,'style','text','str',label{k},'pos',[10 20*k-10 60 20]);
        end
        for k=1:2,
            set(edbox(k),'string',sprintf('%d',newsize(k)),...
                'call','mode=''Set size''; ph_resize');
        end;
        set(edbox(3),'string',sprintf('%.2f',newsize(3)),...
            'call','mode=''Set scale''; ph_resize');

        curvals=sprintf('Current values: Xsize = %d; Ysize = %d',sx(2),sx(1));
        uicontrol(gcf,'style','text','str',curvals,'pos',[10 70 240 20]);

        resizenow = uicontrol(gcf,'str','Resize now','pos',[160 40 80 20],...
            'call','mode=''Resize now''; ph_resize');
        closebtn = uicontrol(gcf,'str','Close','pos',[180 10 60 20],...
            'call','mode=''Close''; ph_resize');
```

Fig. 3.1 continued overleaf.

```
case 'Set size'
    % Update size directly from Xsize or Ysize boxes.
    for k=1:2,
        newsize(k)=round(sscanf(get(edbox(k),'string'),'%f'));
        set(edbox(k),'string',sprintf('%d',newsize(k)));
    end;
    set(edbox(3),'string',' '); % Scale is meaningless in this case.

case 'Set scale'
    % Update from Scale box.
    newsize(3) = sscanf(get(edbox(3),'string'),'%f');
    newsize(1:2) = round(newsize(3) * sx(1:2));
    for k=1:2,
        set(edbox(k),'string',sprintf('%d',newsize(k)));
    end;
    set(edbox(3),'string',sprintf('%.2f',newsize(3)));

case 'Resize now'
    % Get size from Xsize and Ysize boxes.
    for k=1:2,
        newsize(k)=round(sscanf(get(edbox(k),'string'),'%f'));
    end;
    yui = im_resize(xui,newsize(1:2));
    newbefore = 0;
    showimages;

case 'Close'
    close(2);
    newbefore = 1;
    showimages;

end
```

Fig. 3.1: Script to resize image: `ph_resize.m`

3.2 The function `im_resize()`

This function is similar to one called `imresize` in the Matlab Image Processing toolbox, but we have tried to avoid using this toolbox as it can be expensive to purchase and the code for standard Matlab functions can be rather complicated and difficult to understand.

Our function `im_resize` is shown in fig. 3.2. `sx` and `sy` are vectors representing the sizes of the input and output 3-D image arrays `xui` and `yui`.

`ri` and `ci` are vectors which define the row and column indices of the sampling points for the output image from within the input image. In other words, if the columns of a 4-pixel wide input image contained pixels of unit width centred at

[0.5 1.5 2.5 3.5] (assuming the image edges are at 0.0 and 4.0)

and we wished to double the size of the image, then `ci` would need to be the following vector:

$$[0.25 \;\; 0.75 \;\; 1.25 \;\; 1.75 \;\; 2.25 \;\; 2.75 \;\; 3.25 \;\; 3.75]$$

so that the 8 new locations are still uniformly spaced between the image edges at 0.0 and 4.0 and the pixels are now 0.5 units wide.

```
function yui = im_resize(xui,newsize)

% function yui = im_resize(xui,newsize)
% Resize xui to newsize(1:2).
%
% Nick Kingsbury, Cambridge University, 2006.

wh = waitbar(0,'Resizing image');

% Calculate new image size
sx = size(xui);
if length(sx)==2, sx(3) = 1; end % Allow for xui being 2D.
sy = [newsize(1:2) sx(3)];

% ri and ci are the vectors of the row and column indices of the
% locations in xui from which each interpolated point in yui is to come.
% Original locations in xui are [0.5:(sx-0.5)].
% New locations are [0.5:(sy-0.5)] * (sx/sy).
ri = [0.5:sy(1)-0.5]' * (sx(1)/sy(1));
ci = [0.5:sy(2)-0.5] * (sx(2)/sy(2));

% Correct for Matlab indexing from [1:sx] instead of [0.5:(sx-0.5)],
% and ensure that values in rj and cj lie within the range 1 to sx.
rj = max(min(ri+0.5,sx(1)),1);
cj = max(min(ci+0.5,sx(2)),1);

% Perform the resizing of each colour slice using interp2().
yui = uint8(zeros(sy));
for k = 1:sx(3),
    yui(:,:,k) = uint8(interp2(double(xui(:,:,k)),cj,rj,'linear') + 0.5);
    waitbar(k/3); % Increment waitbar after each colour is done.
end

close(wh) % Close wait bar

return
```

Fig. 3.2: Function to perform resizing of an image: `im_resize.m`

Similarly to halve the size of the image, the pixels would be 2 units wide and `ci` would be

$$[1.0 \quad 3.0]$$

These calculations are performed for arbitrary size changes by the 2 lines of code

```
ri = [0.5:sy(1)-0.5]' * (sx(1)/sy(1));
ci = [0.5:sy(2)-0.5] * (sx(2)/sy(2));
```

3.3 Bi-linear interpolation

To calculate pixels at locations that are different from the sampling points in the input image, we normally use linear interpolation. In 1-D, if we have pixels x_a and x_b, sampled at locations a and b, then the linearly interpolated pixel at location p (assumed to lie somewhere between a and b) is given by:

$$x_p = x_a + \frac{p-a}{b-a}(x_b - x_a) = \frac{(b-p)x_a + (p-a)x_b}{b-a} \tag{3.1}$$

A simple example of this is shown in fig. 3.3(a). The input data points are shown as circles. The crosses show the interpolated points that would be produced if the number of points were being doubled, using the above code for `ri` and `ci`.

In 2-D, the equivalent operation is bi-linear interpolation, in which the interpolated pixel at $\{p, q\}$ is given in terms of the four surrounding pixels

$$\begin{matrix} x_{a,c} & x_{a,d} \\ x_{b,c} & x_{b,d} \end{matrix}$$

by

$$x_{p,q} = \frac{(d-q)\ [(b-p)x_{a,c} + (p-a)x_{b,c}] + (q-c)\ [(b-p)x_{a,d} + (p-a)x_{b,d}]}{(b-a)(d-c)} \tag{3.2}$$

This is equivalent to performing 1-D linear interpolation twice, first down the columns and then across the rows (or vice-versa).

In our code, the bi-linear interpolation is performed in 2-D in the for-loop for each colour slice `k` of `xui`, by the standard Matlab function `interp2`, using the statement

```
yui(:,:,k) = uint8(interp2(double(xui(:,:,k)),cj,rj,'linear'));
```

Prior to the for-loop, the vectors `ri` and `ci` are modified to become `rj` and `cj` by adding 0.5 and constraining their end points so that they do not lie outside of the ranges (1 to `sx(1)`) or (1 to `sx(2)`) respectively. We add 0.5 because `interp2` assumes that the elements of `xui` are centred at locations 1 ... `sx(1)` and 1 ... `sx(2)`, rather than at the odd multiples of 0.5 assumed previously. The end points are constrained because `interp2` only works correctly if all of the interpolated points lie within the array of input samples (it produces

NaN, Not-a-Number, for any points that do not). This constraining of the end points is equivalent to assuming that the rows and columns of edge pixels in the input image are repeated (mirrored) just outside of the image boundaries so that the curves are flat here.

In the above statement, the calls to `uint8()` and `double()` are needed because `interp2()` works only with real (double) matrices whereas `xui` and `yui` are both unsigned integer 3-D arrays in order to save memory space (only 3 bytes per pel instead of 24 bytes per pel). We add 0.5 to the interpolated colour slices before converting back to integers. Hence the `uint8()` conversion rounds the results to the nearest integers, and it also limits them to the range 0 to 255.

Since the calculations in `interp2` can take several seconds for large images, we use a wait-bar to indicate progress. It is intialised at the start of the function, incremented as each colour slice is interpolated, and closed at the end.

When interpolation is used in this way, it can have a smoothing effect on the data. For example, if the image size is halved, then each new pixel is the average of the four pixels in the input image that it replaces. Conversely if the image size is doubled, then each new pixel is offset by 0.25 of the input pixel spacing from its nearest neighbour in the input image and the four weights used in equation (3.2) become some permutation of $\{\frac{9}{16} \ \frac{3}{16} \ \frac{3}{16} \ \frac{1}{16}\}$. This produces a smoothing effect that can be demonstrated by zooming in on an image that has been resized to twice its original size.

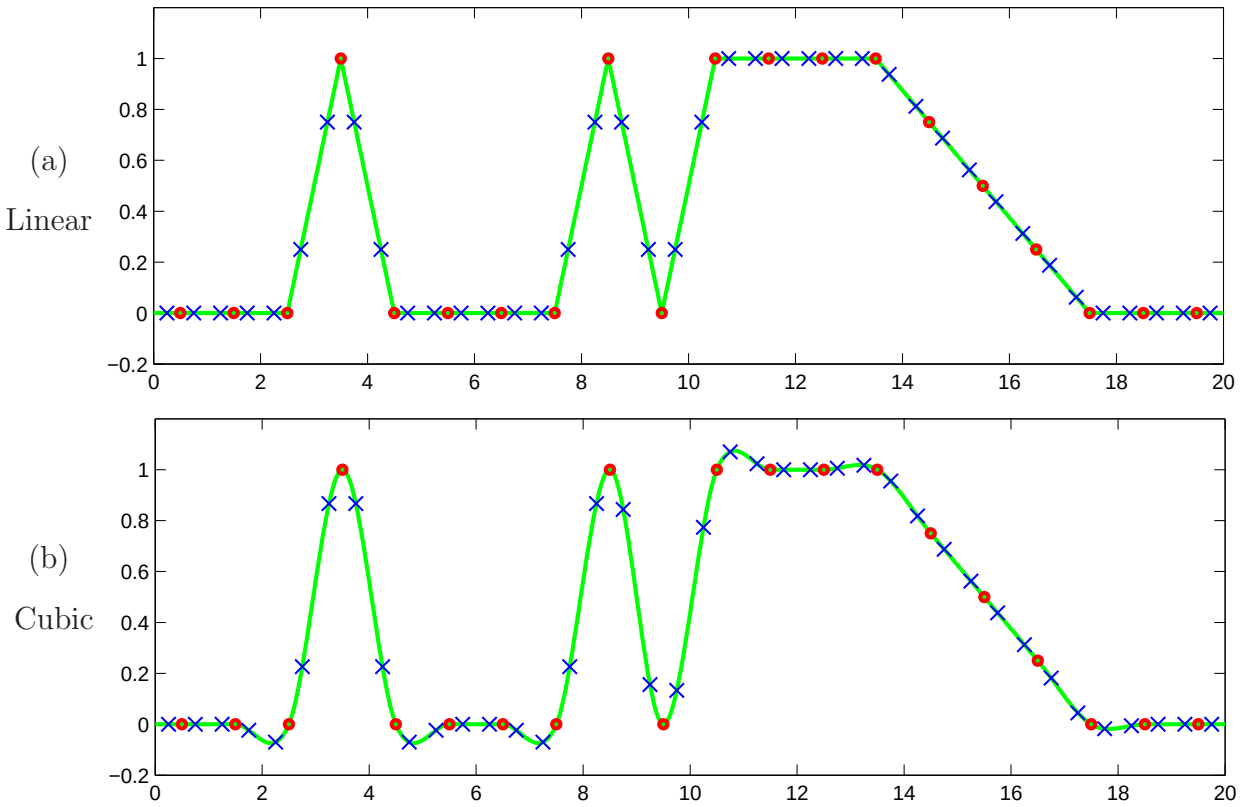


Fig. 3.3: (a) Linear and (b) Cubic interpolation on a 20-element row vector (red circles). The solid lines (green) show the continuously interpolated function, and the crosses (blue) show the values obtained when the vector size is doubled.

3.4 Bi-cubic interpolation

There is a more complicated option than bi-linear interpolation. It is bi-cubic interpolation, and it tends to produce a little less smoothing of the pixels when the image size is doubled. However it requires about four times the computational cost, and so is seldom used for large images. In 1-D, cubic interpolation to a point p uses 4 consecutive samples (instead of 2) located at $\{a, b, c, d\}$, such that $b \leq p \leq c$, to produce an interpolated point

$$\begin{aligned} x_p &= \tfrac{1}{2}\left[-u(1-u)^2 x_a + (1-u)(2+2u-3u^2)x_b + u(1+4u-3u^2)x_c - u^2(1-u)x_d\right] \\ &\text{where} \quad u = \frac{p-b}{c-b} \end{aligned} \tag{3.3}$$

so that $u = 0$ when p is at b, and $u = 1$ when p is at c. The sampling points $\{a, b, c, d\}$ should be uniformly spaced. The results of this method are shown in fig. 3.3(b).

Comparing this with linear interpolation, if we used the same notation equation (3.1) would become

$$x_p = (1-u)x_b + ux_c \tag{3.4}$$

The cubic expression in equation (3.3) is designed to pass through the points x_b when $u = 0$ and x_c when $u = 1$ (as linear interpolation does), but it also includes the constraints that the gradient, dx_p/du, of the cubic curve is

$$\frac{x_c - x_a}{2} \text{ when } u = 0 \qquad \text{and} \qquad \frac{x_d - x_b}{2} \text{ when } u = 1$$

These values ensure continuity of gradient, as well as value, through each sample point (or knot point) on the interpolated curve, and give maximum smoothness when the input points lie on a straight line. Note how, in fig. 3.3(b), the new samples (crosses) in the high-frequency part of the waveform (samples 8 to 11) are spaced further apart and represent a significantly larger amplitude of oscillation than the equivalent samples do in fig. 3.3(a) with linear interpolation. This demonstrates that higher frequencies in the signal are attenuated less by cubic than by linear interpolation. The frequency responses of the two types of interpolation may be obtained by taking the Fourier transform of their impulse responses, shown in fig. 3.3 by the portions of the solid line between samples 1 and 6 in each case. These are shown in fig. 3.4, and we can see that the frequency response of cubic interpolation gives greater flatness at low frequencies and provides higher gain than linear interpolation up to about 0.65 of the sampling frequency of the input data. The improved smoothness of cubic interpolation is the reason for the much lower amplitude of its frequency response sidelobes above the sampling frequency.

Bi-cubic interpolation in 2-D is obtained by applying equation (3.3) in both the vertical and horizontal directions to the 4×4 neighbourhood of pixels around $x_{p,q}$, as was done for the bi-linear case using a 2×2 neighbourhood above. This option may be selected by changing the final argument of `interp2` to 'cubic'. The algebraic expression for $x_{p,q}$ gets very complicated so we do not give it here.

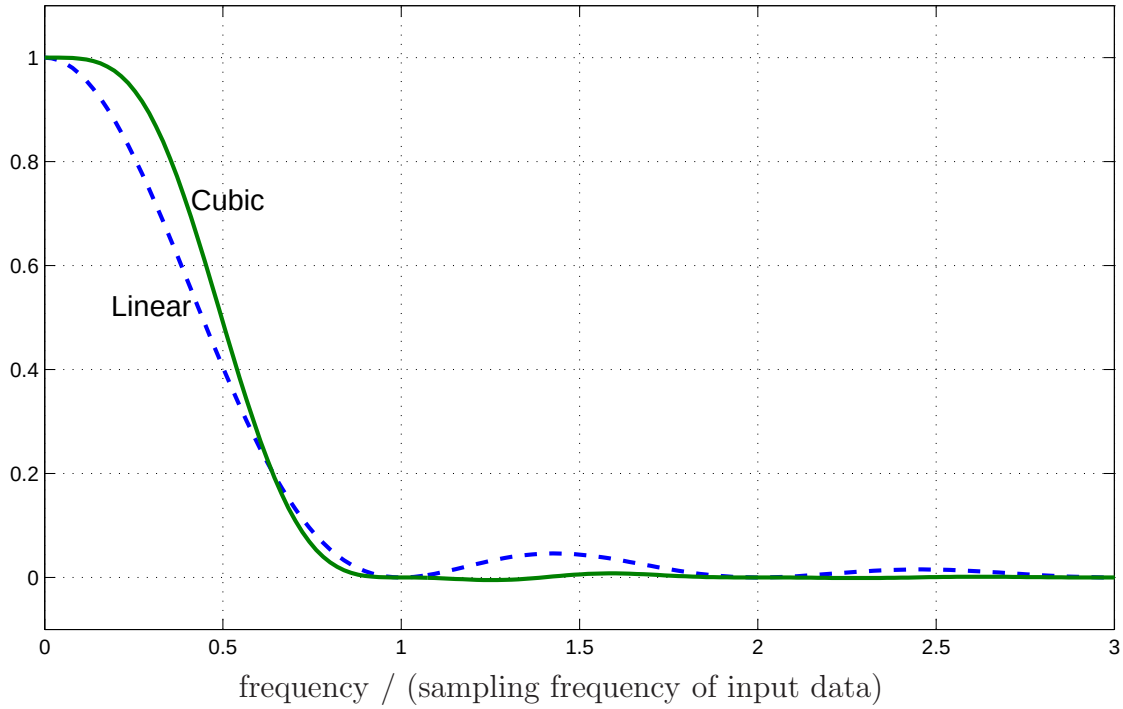


Fig. 3.4: Frequency responses of linear and cubic interpolation processes, obtained by taking the Fourier transforms of their impulse responses.

4 Rotating the image: `ph_rotate`

The script `ph_rotate` is concerned with rotating the image. This can either be used to correct for gross rotations of the camera (e.g. through $\pm 90°$) or for small errors which result in sloping horizons or non-true verticals etc. In order to achieve this, we have written a function `im_rotate()` which, like `im_resize()`, uses the 2-D interpolation function `interp2()`. First we look at the script which implements the user interface.

4.1 The script `ph_rotate`

This script adopts the same general form as `ph_crop` and `ph_resize`. It again comprises five cases selected by `mode`, although one case, **Rotate**, uses a separate **switch** statement at the end so that it can be executed after one of the other cases has been executed before it. The cases are:

Init initialises the **Rotate** function. It opens a command window, figure(2), near the top of the screen and initialises the rotation variable θ to zero. It sets up 4 control buttons for selecting rotations of $-90°, 0°, 90°$ or $180°$ and a text bar above them defining the direction of positive rotation. Then it defines a slider bar and an edit box (with their labels) for setting other rotation angles, and a **Close** button as usual. There is no 'Rotate now' button, because the **Rotate** case is executed whenever any of the other buttons are used.

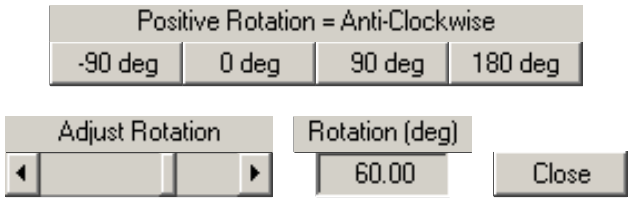
Slider is called when the slider is activated to select θ. It reads the slider value and converts it to θ in the range $-180° \ldots 180°$. It then sets `mode` equal to **Rotate** so that the **Before** image is rotated by θ.

Edit box is called when the edit box is used to define θ numerically. It sets θ to the value in the edit box and sets `mode` equal to **Rotate**, as above.

Close is called when this button is pressed. It closes figure(2) and redisplays **Before** and **After** in figure(1) to show the current and rotated images respectively.

Rotate is called whenever any of the other cases or buttons redefine `mode` to be **Rotate**. It updates the slider position and the edit box to show the current value of θ and then performs the required rotation of `xui` to give `yui`, by a call to `im_rotate()`. Finally `showimages` displays the rotated image `yui`.

Command window, figure(2):




```
% ph_rotate.m
% Routine called by 'Rotate' menu item in photo editor.
%
% Nick Kingsbury, Cambridge University, 2006.

switch mode

    case 'Init'
        % Set up figure window for Rotate commands
        figure(2);
        set(gcf,'position',[400  624  300  100]);
        set(gcf,'numbertitle','off','name','Rotate Image');

        % Initialise rotation angle theta if it does not exist.
        if exist('theta') ~= 1, theta = 0; end

        % Buttons for multiples of 90 deg.
        label = {'-90 deg',' 0 deg ',' 90 deg','180 deg'};
        for u = 1:4,
            btn(u) = uicontrol(gcf,'str',label{u},'pos',[60*u-30 60 60 20]);
        end
        set(btn(1),'call','theta=-90; mode=''Rotate''; ph_rotate');
        set(btn(2),'call','theta=0; mode=''Rotate''; ph_rotate');
        set(btn(3),'call','theta=90; mode=''Rotate''; ph_rotate');
        set(btn(4),'call','theta=180; mode=''Rotate''; ph_rotate');

        uicontrol(gcf,'style','text','str',...
            'Positive Rotation = Anti-Clockwise','pos',[30 80 240 16]);

        % Other buttons: slider, edit box, and close box.
        slide = uicontrol(gcf,'style','slider','value',theta/360+0.5,...
            'sliderstep',[1 10]/360,'pos',[10 10 120 20],...
            'call','mode=''Slider''; ph_rotate');
        uicontrol(gcf,'style','text','str','Adjust Rotation',...
            'pos',[10 30 120 16]);
        edbox = uicontrol(gcf,'style','edit','string',sprintf('%.2f',theta),...
            'pos',[150 10 60 20],'call','mode=''Edit box''; ph_rotate');
        uicontrol(gcf,'style','text','str','Rotation (deg)',...
            'pos',[140 30 80 16]);
        closebtn = uicontrol(gcf,'str','Close','pos',[230 10 60 20],...
            'call','mode=''Close''; ph_rotate');
```

Fig. 4.1 continued overleaf.

```
    case 'Slider'
        theta = (get(slide,'value')-0.5)*360;
        mode = 'Rotate';

    case 'Edit box'
        theta = sscanf(get(edbox,'string'),'%f');
        mode = 'Rotate';

    case 'Close'
        close(2);
        newbefore = 1;
        showimages;

end

switch mode

    case 'Rotate'
        % Update slider position and edit box value.
        set(slide,'value',theta/360+0.5);
        set(edbox,'string',sprintf('%.2f',theta));
        % Rotate the image and display it.
        yui = im_rotate(xui,theta);
        newbefore=0;
        showimages

end
```

Fig. 4.1: Script to rotate image: `ph_rotate.m`

4.2 The function `im_rotate()`

This function is similar to one called `imrotate` in the Matlab Image Processing toolbox. Our function `im_rotate` is shown in fig. 4.2.

The initial **if** statement checks whether θ is $90°, -90°, 180°$ or $0°$, and, if so, it uses a combination of the transpose operator and reversal of row or column indices to achieve the desired rotation. A for-loop is needed when transpose is used, because each colour slice must be transposed separately as a 2-D matrix.

For all other rotation angles, the more complex function, listed under **else**, is used. Again a wait-bar is employed here, as this operation can be a bit slow.

First a rotation matrix R is formed as

$$R = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \tag{4.1}$$

Then the required size of the output image `y` is calculated so that it will be big enough to accomodate the whole of the rotated image. If the input image is of size $U \times V$ pels, then the output image must be of size $P \times Q$ pels, where

$$P = U\,|\cos\theta\,| + V\,|\sin\theta\,| \qquad \text{and} \qquad Q = U\,|\sin\theta\,| + V\,|\cos\theta\,| \tag{4.2}$$

This is calculated and rounded up to the next larger integer by

```
sy = ceil(sx(1:2) * abs(R));
```

The rotation needs to be performed about the centre of the image `y`, so we first form `p` and `q`, two vectors that give the distance of each row / column of `y` from the image centre.

We now need to calculate the location $[u, v]$ in `xui` from which each pel at $[p, q]$ in `y` needs to come. If u, v, p, q are all measured from the centre (point of rotation) of their respective images, then

$$[u, v] = [p, q]\ R^T \tag{4.3}$$

(Note that u, p represent row indices, measured downwards, and v, q represent column indices, measured from left to right.)

The following code lines calculate u and v for every point in image `y`:

```
u = cs*p*ones(1,sy(2)) + sn*ones(sy(1),1)*q + (sx(1)+1)/2;
v = cs*ones(sy(1),1)*q - sn*p*ones(1,sy(2)) + (sx(2)+1)/2;
```

The coordinates of the centre of image `xui` are added at the end of these two statements so that `u` and `v` now are measured from the top left corner of `xui` in the normal way for matrices, and are in the correct form to be used by `interp2()`.

Next, the for-loop calculates the rotated colour slices using `interp2()` in the same way as in `im_resize`. We need to use interpolation because each location $[u, v]$ will in general not be located precisely at a pixel location of `xui`. Again, for speed, we use bi-linear interpolation, rather than bi-cubic. The statement

```
y(find(isnan(y))) = 128;
```

finds all pels outside the rotated image (which are returned as NaN by `interp2()`) and colours them mid-grey (R=G=B=128). Then `y` is converted back to uint8 format and saved in `yui`.

```
function yui = im_rotate(xui,theta)

% function yui = im_rotate(xui,theta)
% Rotate xui by theta degrees.
%
% Nick Kingsbury, Cambridge University, 2006.

% Use quick rotations if theta is a multiple of 90 deg.
if theta == 90,
    for k = 1:3, yui(:,:,k) = xui(:,end:-1:1,k)'; end
elseif theta == -90,
    for k = 1:3, yui(:,:,k) = xui(end:-1:1,:,k)'; end
elseif abs(theta) == 180,
    yui = xui(end:-1:1,end:-1:1,:);
elseif theta == 0,
    yui = xui;
else  % Slower function for any angle.
    wh = waitbar(0,'Rotating image');
    thrad = theta * pi / 180; % theta in radians
    cs = cos(thrad);
    sn = sin(thrad);
    R = [cs sn; -sn cs];
    sx = size(xui);
    sy = ceil(sx(1:2) * abs(R)); % Size for rotated image y

    % p and q are vectors of distances of each row/column of y
    % from its centre.
    p = [1:sy(1)]' - (sy(1)+1)/2;
    q = [1:sy(2)] - (sy(2)+1)/2;

    % u and v are the matrices of the row and column indices of the
    % locations in xui from which each interpolated point in y is to come.
    u = cs*p*ones(1,sy(2)) + sn*ones(sy(1),1)*q + (sx(1)+1)/2;
    v = cs*ones(sy(1),1)*q - sn*p*ones(1,sy(2)) + (sx(2)+1)/2;

    % Perform the rotation using interp2().
    yui = uint8(zeros(sy(1),sy(2),3));
    for k = 1:3,
        y = interp2(double(xui(:,:,k)),v,u,'linear');
        % Replace all points coming from outside of xui by mid grey.
        y(find(isnan(y))) = 128;
        yui(:,:,k) = uint8(y); % Convert output image back to uint8.
        waitbar(k/3);
    end
    close(wh) % Close wait bar
end
return
```

Fig. 4.2: Function to perform rotating of an image: `im_rotate.m`

5 Morphing the image: `ph_morph`

The script `ph_morph` is concerned with morphing the image, which means distorting parts of it spatially. Full morphing often means gradually converting one image into another (e.g. one face to another) by a series of local translations and mixings of the two images. This is rather complex to do (and beyond the scope of this course unfortunately), so we have implemented just the local translation part and apply it to a single image. We can still get some interesting / amusing effects however.

Again we use the 2-D interpolation function `interp2()` to achieve smooth translations by non-integer numbers of pels. It has been quite difficult to devise a simple user interface for this function, but below is my attempt at this!

5.1 The script `ph_morph`

This script adopts the same general form as previous ones but is more complicated. It comprises 11 cases selected by `mode`.

The morphing is defined by sets of **Control Points** in a matrix `cp`. Each row of `cp` comprises 5 elements, which define the 2 coordinates of the morph start point, the 2 coordinates of the morph end point, and the morph radius. These are normally selected by 3 mouse clicks on the start point, the end point and a point at the desired radius from the start point (which can often conveniently also be the end point). The morphing is designed to shift pels near the start point $\mathbf{c}_0 = [u, v]$ to being near the end point $\mathbf{c}_1 = [p, q]$ instead. It is necessary to define how large a region of the image around the start point will be shifted with it, and this is the function of the radius parameter r. We use a Gaussian 'blob' $g(s,t)$, centred on $\mathbf{c}_1$ with standard deviation r, to define a smoothly varying field of pixel shift vectors as follows:

$$\mathbf{x}_{shift}(s,t) = g(s,t)[u-p,\ v-q] \quad \text{where} \quad g(s,t) = \exp\left(-\frac{(s-p)^2+(t-q)^2}{2r^2}\right) \tag{5.1}$$

Hence the pixel at a given location $[s,t]$ in the output image will be interpolated from location $[s,t] + \mathbf{x}_{shift}(s,t)$ in the input image. Fig. 5.1 shows the shape of $g(s,t)$.

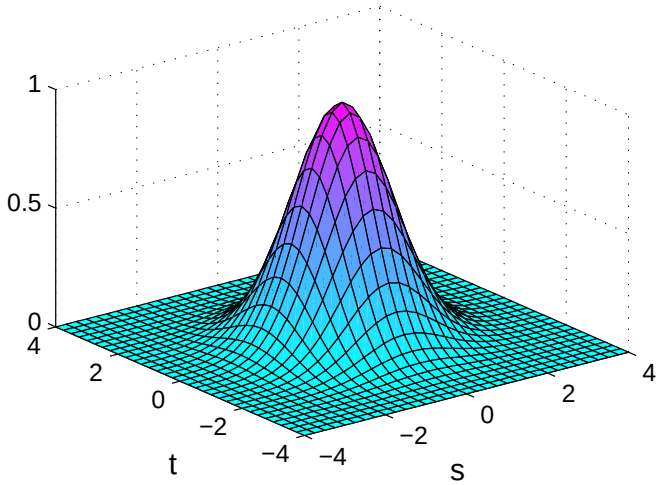


Fig. 5.1: 2-D Gaussian function ('blob'), $g(s,t)$, centred on [0,0] with $r = 1$.

In order that the ordering of pixels in the input image is retained in the output image, it is necessary for the gradient of $\mathbf{x}_{shift}(s,t)$ to be greater than -1 everywhere (because it is added to a vector with gradient $= +1$, and the overall gradient must remain positive to preserve the pixel ordering). Now, for Gaussian functions:

$$\frac{d}{dx}\, e^{-x^2/2r^2} = -\frac{x}{r^2}\, e^{-x^2/2r^2} \quad \text{and} \quad \frac{d^2}{dx^2}\, e^{-x^2/2r^2} = -\frac{1}{r^2}\left(1 - \frac{x^2}{r^2}\right) e^{-x^2/2r^2} \tag{5.2}$$

Hence the steepest gradient occurs when $x = \pm r$ and is $e^{-0.5}/r = 0.6065/r$ in magnitude. This gets scaled by the displacement vector $[u-p,\ v-q]$, so in theory r must be greater than 0.6065 times the magnitude of this vector in order to avoid a negative overall gradient. In practice for acceptable-looking output images, we find that r must be at least 0.8 times the magnitude of the displacement vector; i.e. $r \geq 0.8\sqrt{(u-p)^2 + (v-q)^2}$. This limit is applied by the program whenever the selected r is less than 0.8 of the distance from $\mathbf{c}_0$ to $\mathbf{c}_1$.

The user interface is designed so that new control points can be added to those already selected, and as each set of control points is completed, the **After** image is updated to show their effect. Sets of control points may be deleted from the list in reverse order of entry. When a suitable morph has been generated, the user can request a movie sequence of 10 frames which show a gradual transition from **Before** to **After**, by pressing the **Play** button. Once generated, the sequence can be played alternately backwards and forwards using the **Play again** control button in figure(1). The sets of control points stored in `cp` may be saved in a .mat file, using the **Save** button, and may be reloaded from this file using the **Load** button. The buttons, created when `ph_morh` is intialised, are shown at the top of fig. 5.2.

The 11 cases in the code for `ph_morh` are:

Init initialises the **Morph** function. It opens a command figure (2) near the top of the screen. First it sets up the 6 control buttons on the top row, it initialises the control point matrix `cp` to be empty, and it sets up a status display above the 6 buttons to show how many sets of control points are in `cp`. Then it sets up a slider and edit-box to allow selection of individual movie frames, and a **Close** button as usual.

The variable `newsequence` is used to indicate when `cp` has been changed, and hence when the **Play** button should cause a new sequence of frames to be calculated if it is pressed.

New is called when this button is pressed. It sets `cp` to empty, the matrix of shift vectors `xshift = 0` and `newsequence = 1`. It then sets `mode` to **Get points** so that new control points can be entered using the mouse.

Add is called when this button is pressed. It is similar to **New** except that `cp` and `xshift` are not reset.

Delete is called when this button is pressed. It removes the final row of control points from `cp` (until it is empty) and recalculates the morphed image in `yui` using the remaining entries from `cp`. It then updates **After** in the display.

Play is called when this button is pressed. If `newsequence` is true (non-zero), we calculate a new sequence of 10 frames, using a scale-factor of $\frac{k-1}{9}$ on the $\mathbf{x}_{shift}$ vector field before

each morphing operation, so that the field gradually increases from zero to its full value as $k = 1$ to 10. The frames are stored in the 4-D array `yu`. Then the prompt and control-button in figure(1) are redefined to do **Play again**, and the sequence from `yu` is displayed smoothly frame-by-frame by setting the **Cdata** property of `yhandle` (the handle to **After**). The direction of **Play** is controlled by `dk`, which is initialised to 1 when the sequence is first calculated and then negated after each play.

Load is called when this button is pressed. It loads the control points, `cp`, from a user-selected file, and then displays the resulting morphed version of **Before** as **After**.

Save is called when this button is pressed. It saves `cp` in a user-selected file.

Select frame is called when the slider is moved. It displays the frame from `yu` that corresponds to the slider position $k = 1$ to 10. It also updates the edit box with the value of k.

Enter frame is called when the edit box is used to define the frame number k numerically. It sets k to the value in the edit box and displays frame k from `yu`. It also updates the slider position to the new k.

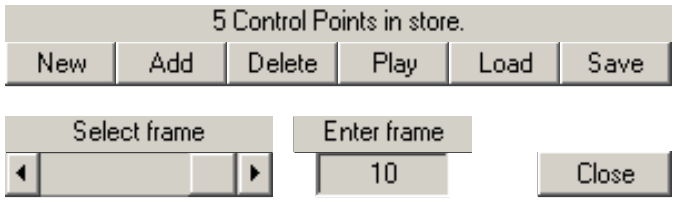
Get points is called whenever any of the other cases or buttons redefine `mode` to be **Get points**. It sets the figure(1) control button to the **Terminate morph** function and then enters a while-loop to allow enrty of each new set of control points. The loop can only be exited by a call to **break**. I have modified the standard Matlab graphical input function `ginput()` to allow the cursor to be defined by the second argument, and called this version `ginputnk()`. Hence we use **crosshair**, **cross** and **circle** as cursors when entering the coordinates for $\mathbf{c}_0, \mathbf{c}_1$ and $\mathbf{c}_2$ respectively. The Gaussian radius r is then obtained as $r = |\mathbf{c}_2 - \mathbf{c}_0|$. After each point is entered, it is plotted and the prompt is updated to request the next point. When all three have been entered, the new shift field and morphed image are calculated by

```
[yui,xshift] = im_morph(xui,xshift,cp(end,:));
```

Then the **After** image and the control-point status message are updated. The loop is terminated by pressing the figure(1) control button or the Return key on the keyboard, which then updates the full display and brings figure(2) into view.

Close is called when this button is pressed. It closes figure(2) and updates figure(1).

Command window, figure(2):




```
%% ph_morph.m
%
% Script called by 'Morph' menu item in photo editor.
% mode defines which block of code is executed. Valid modes are:
% 'init','new','add','delete','play','load','save','close'
%
% Nick Kingsbury, Cambridge University, March 2006.

switch mode

    case 'Init'
        % Set up figure window for Morph commands
        figure(2);
        set(gcf,'position',[400  624  320  100]);
        set(gcf,'numbertitle','off','name','Morph Image');
        % Define buttons
        label = {'New','Add','Delete','Play','Load','Save',...
                'Select frame','Enter frame','Close'};
        for u = 1:6,
            btn(u) = uicontrol(gcf,'str',label{u},'pos',[50*u-40 60 50 20],...
                'call',sprintf('mode=label{%d}; ph_morph',u));
        end
        if exist('cp')~=1, cp = []; end % Initialise control-point matrix.
        cpstatus = uicontrol(gcf,'style','text','pos',[10 80 300 16],...
           'str',sprintf('%d Control Points in store.',size(cp,1)));

        % Other buttons: slider, edit box, and close.
        nframes = 10;
        slide = uicontrol(gcf,'style','slider','value',1,'pos',[10 10 120 20],...
            'sliderstep',[1 3]/(nframes-1),'call','mode=label{7}; ph_morph');
        uicontrol(gcf,'style','text','str',label{7},'pos',[10 30 120 16]);
        % set(slide,'call',['theta=(get(slide,''value'')-0.5)*360;']);
        edbox = uicontrol(gcf,'style','edit','string',nframes,...
            'pos',[150 10 60 20],'call','mode=label{8}; ph_morph');
        uicontrol(gcf,'style','text','str',label{8},'pos',[140 30 80 16]);

        closebtn = uicontrol(gcf,'str','Close','pos',[250 10 60 20],...
            'call','mode=''Close''; ph_morph');

        newsequence = 1;
```

Fig. 5.2 continued overleaf.

```
case 'New'
    cp = [];
    xshift = 0;
    newsequence = 1;
    yui = xui;
    update_y;
    mode='Get points';

case 'Add'
    newsequence = 1;
    update_y;
    mode='Get points';

case 'Delete'
    newsequence = 1;
    % Remove most recent row of control points from cp,
    % recalculate result of morph, and display it.
    if size(cp,1) > 1,
        cp = cp(1:(end-1),:);
        [yui,xshift] = im_morph(xui,0,cp);
    else
        cp = [];
        xshift = 0;
        yui = xui;
    end
    set(cpstatus,'str',sprintf('%d Control Points in store.',size(cp,1)));
    update_y;
    figure(2);

case 'Play'
    figure(1);
    set(yhandle,'Erasemode','none');
    % If required, calculate new sequence of frames for movie.
    if newsequence,
        yu = uint8(zeros([size(yui) nframes]));
        for k=1:nframes,
            waitstr = sprintf('Morphing frame %d of %d',k,nframes);
            yui = im_morph(xui,((k-1)/(nframes-1))*xshift,[],waitstr);
            yu(:,:,:,k) = yui;
            set(yhandle,'Cdata',yui);
            drawnow;
        end;
        newsequence = 0;
        dk = 1; % direction = 'forward'
    end
    % Set up prompt and control button.
    set(promptbox,'str','Click button to morph back and forth');
    set(contrbutton,'str','Play again','call','mode=''Play''; ph_morph');
```

Fig. 5.2 continued overleaf.

```
        % Play movie.
        for k=1:nframes,
            yui = yu(:,:,:,k*dk + (nframes+1)*(1-dk)/2);
            set(yhandle,'Cdata',yui);
            pause(0.05);
        end;
        dk = -dk; % swap direction for next 'play'.

    case 'Load'
        [Filename,PathName] = uigetfile({'*.mat','Morph cp files (*.mat)'},...
            'Open CP File');
        % Prompt user for a tif or jpeg filename.
        load([PathName Filename],'cp')
        [yui,xshift] = im_morph(xui,0,cp);
        update_y;
        newsequence = 1;
        figure(2);

    case 'Save'
        % Prompt user for an output filename.
        if ~exist('cpfile'), cpfile = 'morph_cp.mat'; end
        [FileName,PathName] = uiputfile(cpfile,'Save CP File');
        % Write yui to the file if Filename is valid.
        if FileName == 0,
            disp('Control-point image has not been saved.');
        else
            save([PathName FileName],'cp');
        end

    case 'Select frame'
        k = round(get(slide,'value')*(nframes-1)) + 1;
        set(edbox,'string',sprintf('%d',k));
        yui = yu(:,:,:,k);
        set(yhandle,'Cdata',yui,'Erasemode','none');

    case 'Enter frame'
        k = max(min(round(sscanf(get(edbox,'string'),'%f')),nframes),1);
        set(slide,'value',(k-1)/(nframes-1));
        yui = yu(:,:,:,k);
        set(yhandle,'Cdata',yui,'Erasemode','none');

end
```

Fig. 5.2 continued overleaf.

```
% The following modes can be called after the first group.
switch mode

    case 'Get points'
        set(contrbutton,'str','Terminate morph','call',' ');
        while 1,
           set(promptbox,'str','Place cross on morph start point');
           c0 = ginputnk(1,'crosshair');
           if isempty(c0) | any(c0<0), break; end;
           hold on; plot(c0(1),c0(2),'c+'); hold off;
           set(promptbox,'str','Place cross on morph end point');
           c1 = ginputnk(1,'cross');
           if isempty(c1) | any(c1<0), break; end;
           hold on; plot(c1(1),c1(2),'cx'); hold off;
           set(promptbox,'str','Place circle at radius from start point');
           c2 = ginputnk(1,'circle');
           if isempty(c2) | any(c2<0), break; end;
           rad = sqrt(sum((c2-c0).^2));
           cp = [cp; c0 c1 rad];
           [yui,xshift] = im_morph(xui,xshift,cp(end,:));
           update_y;
           set(cpstatus,'str',sprintf('%d Control Points in store.',size(cp,1)));
        end;
        newbefore=1;
        showimages;
        figure(2);

    case 'Close'
        close(2);
        newbefore = 1;
        showimages;

end
```

Fig. 5.2: Script to morph image: `ph_morph.m`

5.2 The function `im_morph()`

This function calculates the shift vector field `xshift` and the morphed output image `yui`, from the input image `xui` and the control-point matrix `cp`. To save computation time whenever an extra row is added to `cp`, the previous `xshift` may be supplied as an input together with only the new final row of `cp`. Alternatively if the whole of `cp` is supplied, the input `xshift` should be zero. The optional fourth input argument of the function is a string that is displayed on the wait-bar, so that this can show status information such as the frame number when 10 frames are being morphed for the **Play** option.

First we set up the wait-bar and some vectors which depend on the image size. Then, if `cp` is not empty, the for-loop in `nc` computes the Gaussian blob $g(s,t)$, corresponding to each set of control points `cp(nc,:)` as in equation (5.1), and adds $g(s,t)$ times $(\mathbf{c}_1 - \mathbf{c}_0)$ to the old value of `xshift` to get its new value. For convenience, we represent the two components of `xshift` as the real and imaginary parts of a complex matrix.

Once `xshift` has been accumulated across all rows of `cp`, matrices of row and column indices, `ri` and `ci`, are calculated from `xshift`. These are then used by the `interp2()` function in the final for-loop to achieve the desired shifts of pixels, in the same way as was done in `im_rotate()` with matrices `v` and `u`.

The wait-bar is incremented after each colour slice is morphed and is closed when the function is completed.

```
function [yui,xshift] = im_morph(xui,xshift,cp,waitstr)

% function [yui,xshift] = im_morph(xui,xshift,cp,waitstr)
%
% Morph xui, so that pixels near control points cp(:,1:2) are moved to cp(:,3:4).
% The region of influence is a set of Gaussian blobs whose std dev = cp(:,5).
% xshift is the shift matrix that accumulates the effects of each blob.
% Set xshift = 0 on the first call to im_morph.
% waitstr is the displayed title of the wait bar.
%
% Nick Kingsbury, Cambridge University, 2006.

if nargin < 4, waitstr = 'Morphing image'; end
wh = waitbar(0,waitstr);

sx = size(xui);
s = [1:sx(1)].';
t = [1:sx(2)];
o1 = ones(sx(1),1);
o2 = ones(1,sx(2));

% If extra morph points are given, add their effects to xshift.
if (nargin > 2) & (~isempty(cp)),
    sc = size(cp); % cp must be of width 5.
    for nc = 1:sc(1),  % Loop for each pair of points to be morphed.
      % Ensure that ri and ci have all elements still in monotonically increasing
      % order in each direction, by increasing rad(nc) if necessary.
      rad = max(cp(nc,5),0.8*sqrt(sum((cp(nc,3:4)-cp(nc,1:2)).^2)));
      % Generate the Gaussian blob, centred on c1(nc,1:2) and of radius rad(nc).
      gblob = exp(-(0.5/rad.^2)*((s-cp(nc,4)).^2*o2 + o1*(t-cp(nc,3)).^2));
      % Scale the blob by (c0(nc,:)-c1(nc,:)) and add it to xshift.
      xshift = xshift + gblob*(cp(nc,1) - cp(nc,3) + j*(cp(nc,2)-cp(nc,4)));
    end
end

% ri and ci are the matrices of the row and column indices of the
% locations in xui from which each interpolated point in yui is to come.
ri = max(min(s*o2 + imag(xshift),sx(1)),1);
ci = max(min(o1*t + real(xshift),sx(2)),1);

% Perform the morphing using interp2().
yui = uint8(zeros(sx(1),sx(2),3));
for k = 1:3,
    yui(:,:,k) = uint8(interp2(double(xui(:,:,k)),ci,ri,'linear') + 0.5);
    waitbar(k/3);
end
close(wh) % Close wait bar
return
```

Fig. 5.3: Function to perform morphing of an image: `im_morph.m`

6 Colour conversions and colour correction: `ph_colourshift`

The script `ph_colourshift` is concerned with adjusting colours within the image. To do this we have designed a rather general interface which allows each colour component of the complete image to be scaled (multiplied) by a factor or shifted in level using addition of an offset value. The range of options here can be increased by working in a variety of colour representations (i.e. not just RGB – red, green and blue components).

First we shall discuss three of these representations and then describe the script file which uses them.

6.1 Colour Representations: RGB, YUV and HSV

RGB means **Red Green Blue** and is the normal colour space used in most modern computers and their video display cards. The two other commonly-used colour spaces that we will consider are YUV (**Luminance, Chrominance**) and HSV (**Hue, Saturation, Value**).

Some common colours and their RGB, YUV and HSV values are listed below in table 6.1. We will consider each colour space in turn.

Colour	*R*	*G*	*B*	*Y*	*U*	*V*	*H*	*S*	*V*
Black	0	0	0	0	0	0	-	-	0
Mid-grey	0.5	0.5	0.5	0.5	0	0	-	0	0.5
White	1	1	1	1	0	0	-	0	1
Red	1	0	0	0.3	−0.15	0.4375	0	1	1
Yellow	1	1	0	0.9	−0.45	0.0625	0.167	1	1
Green	0	1	0	0.6	−0.3	−0.375	0.333	1	1
Cyan	0	1	1	0.7	0.15	−0.4375	0.5	1	1
Blue	0	0	1	0.1	0.45	−0.0625	0.667	1	1
Magenta	1	0	1	0.4	0.3	0.375	0.833	1	1
Pink	1	0.5	0.5	0.65	−0.0750	0.2188	0	0.5	1
Pale green	0.5	1	0.5	0.8	−0.15	−0.1875	0.333	0.5	1
Pale blue	0.5	0.5	1	0.55	0.225	−0.0313	0.667	0.5	1

Table 6.1: Common colours and their values in RGB, YUV and HSV spaces.

The RGB Colour Space

The R, G and B components of an RGB image, represent the amplitude of each colour component (red, green and blue) over a perceptually uniform range from 0 to 1, when the components are real values. If the components are unsigned single-byte integers (`uint8`), then they are scaled up by 255 to cover the range 0 to 255. Display screens are usually adjusted so that, for any shade of grey, R = G = B.

The main problem with the RGB space is that each colour affects the apparent brightness (luminance) by different amounts, and the human eye is much more sensitive to changes in luminance than it is to changes in colour (chrominance). Usually most of the information

about a scene is contained in its luminance rather than its colour (chrominance). This is why black-and-white (monochrome) reproduction was acceptable for photography and TV for many years until technology provided colour reproduction at a sufficiently cheap price to make its modest advantages worth having.

The YUV Colour Space

To allow us to take advantage of the different ways that the human visual system responds to luminance and chrominance components of an image, we can represent it in the YUV colour space. Here Y is the **luminance** (brightness) component, and U and V are two colour-difference or **chrominance** components, indicating how far away from grey the colour is, in the blue and red directions respectively.

The **luminance** (Y) of a pel may be obtained from its RGB components as:

$$Y = 0.3R + 0.6G + 0.1B \tag{6.1}$$

These coefficients are only approximate, and in other places values of 0.3, 0.59 and 0.11 are used. They have been selected according to the amount of each pure colour (R, G or B) which is needed to produce a given amount of apparent brightness to the human visual system. Hence green produces approximately twice as much apparent brightness as red, and six times as much as blue!

When $R = G = B$ the pel is always some shade of grey, and if $Y = R = G = B$ in these cases, the 3 coefficients in equation (6.1) must sum to unity (which they do).

When Y defines the luminance of a pel, its **chrominance** is defined by U and V such that:

$$\begin{aligned} U &= 0.5(B - Y) &&= -0.15R - 0.3G + 0.45B \\ V &= 0.625(R - Y) &&= 0.4375R - 0.375G - 0.0625B \end{aligned} \tag{6.2}$$

Note that grey pels will always have $U = V = 0$. The scale factors of 0.5 and 0.625 are the largest multiples of $\frac{1}{8}$ which ensure that U and V remain within the range ± 0.5 when R, G and B vary independently from 0 to 1.

The transformation between RGB and YUV colour spaces is linear and may be achieved by multiplication by a 3×3 matrix $\mathbf{C}$ or its inverse.

Hence

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \mathbf{C} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \qquad \text{where} \quad \mathbf{C} = \begin{bmatrix} 0.3 & 0.6 & 0.1 \\ -0.15 & -0.3 & 0.45 \\ 0.4375 & -0.375 & -0.0625 \end{bmatrix} \tag{6.3}$$

and

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \mathbf{C}^{-1} \begin{bmatrix} Y \\ U \\ V \end{bmatrix} \qquad \text{where} \quad \mathbf{C}^{-1} = \begin{bmatrix} 1 & 0 & 1.6 \\ 1 & -0.3333 & -0.8 \\ 1 & 2 & 0 \end{bmatrix} \tag{6.4}$$

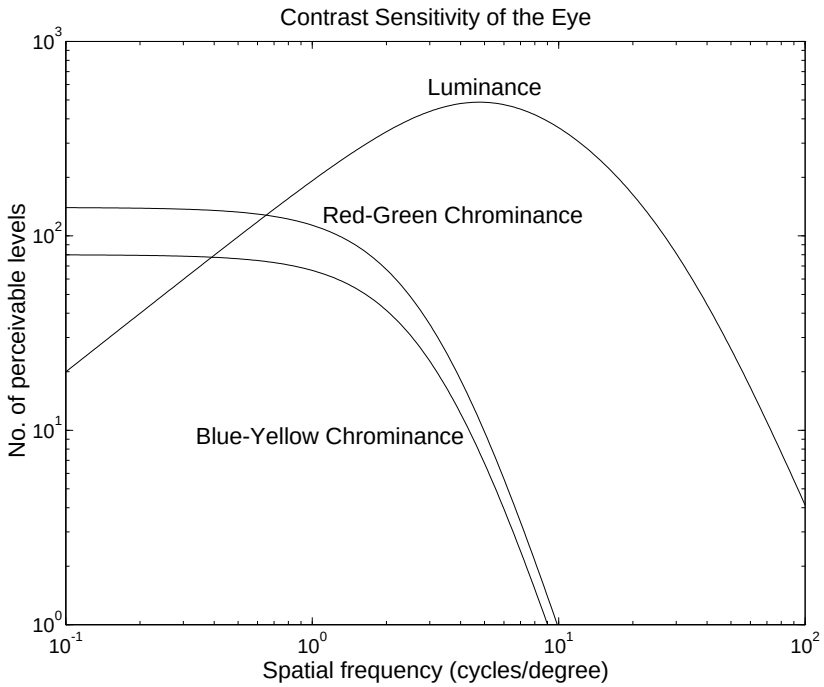


Fig 6.1: Sensitivity of the eye to luminance and chrominance intensity changes.

Fig 6.1 shows the sensitivity of the eye to luminance (Y) and chrominance (U,V) components of images. The horizontal scale is spatial frequency, and represents the frequency of an alternating pattern of parallel stripes with sinusoidally varying intensity. The vertical scale is the contrast sensitivity of human vision, which is the ratio of the maximum visible range of intensities to the minimum discernable peak-to-peak intensity variation at the specified frequency.

In fig 1.1 we see that:

- The maximum sensitivity to Y occurs for spatial frequencies around 5 cycles / degree, which corresponds to striped patterns with a half-period (stripe width) of 1.8 mm at a distance of 1 m ($\sim$ arm's length).
- The eye has very little response above 100 cycles / degree, which corresponds to a stripe width of about 0.1 mm at 1 m. A standard 19-inch computer display has a diagonal dimension of $19 \times 25.4 \simeq 480$mm. If it has an image size of 1280×1024 pixels, then the pixel edge size is $480/\sqrt{1280^2 + 1024^2} = 0.29$mm; so these displays still fall somewhat short of the ideal for people with good eyesight. Despite the fact that modern flat-panel displays have resolutions that are slightly coarser than is possible with the older CRT technology, in general they are more pleasing and less tiring to view because the pixel edges are so sharp and there is no flicker.
- The sensitivity to luminance drops off at low spatial frequencies, showing that we are not very good at estimating absolute luminance levels **as long as they do not change with time** – the luminance sensitivity to temporal fluctuations (flicker) does not fall off at low spatial frequencies.

- The maximum chrominance sensitivity is much lower than the maximum luminance sensitivity, with blue-yellow (U) sensitivity being about half of red-green (V) sensitivity and about $\frac{1}{6}$ of the maximum luminance sensitivity.
- The chrominance sensitivities fall off above 1 cycle / degree, requiring a much lower spatial bandwidth than luminance.

We can now see why it is better to convert to the YUV domain before attempting bandwidth critical tasks such as image compression and transmission of analogue colour TV signals. The U and V components may be sampled at a lower rate than Y (due to their narrower bandwidth) and may be quantised more coarsely (due to their lower contrast sensitivity). This is done in most JPEG-encoded images and MPEG-encoded video (e.g. digital TV and DVDs).

By adjusting the gain applied to the U and V components we can easily alter the amount (or saturation) of colour in an image, and by adding an offset to U and V we can adjust the colour balance (or tint). The brightness and contrast level of the image may be altered by adjusting the offset or gain applied to the Y component.

The HSV Colour Space

HSV stands for **Hue, Saturation, Value**. It is a non-linear version of some of the YUV ideas, except that it codes the chrominance information using polar coordinates (H, S) instead of cartesians (U, V), and is formed as follows:

The Value component, V, of HSV indicates the amplitude of the largest component out of R, G and B. It represents the *approximate* intensity (luminance) of the pixel.

The Saturation component, S, is the difference between the value V and the smallest of R, G and B. It represents how far the colour of the pixel is from some shade of grey.

Hence

$$V = \max(R, G, B) \qquad \text{and} \qquad S = \frac{V - \min(R, G, B)}{V} \tag{6.5}$$

The Hue component, H, is a function of the colour of the largest component, adjusted by the other components as follows:

$$H = \begin{cases} \frac{G-B}{6SV} & \text{if } V = R \text{ and } G \geq B \\ \frac{B-R}{6SV} + \frac{1}{3} & \text{if } V = G \\ \frac{R-G}{6SV} + \frac{2}{3} & \text{if } V = B \\ \frac{G-B}{6SV} + 1 & \text{if } V = R \text{ and } G < B \end{cases} \tag{6.6}$$

Hence the colours {red, yellow, green, cyan, blue, magenta, red} form a 'circle' of hues from 0 to 1 (see table 6.1). Note that pink, pale green and pale blue have the same hues as red, green and blue, but have a saturation of 0.5 instead of 1.0, because the pale colours are the averages of the saturated colours and white.

The hue and saturation components of an HSV colour are effectively a polar coordinate representation of the chrominance with H being angle/2π and S being the radius (normalised by the value V).

The advantage of an image being in HSV form, is that the apparent brightness can be controlled by V, the strength of colour by S, and the colour tint by H. This independent control of brightness, saturation and tint is not possible with either RGB or YUV, although YUV does allow control of the brightness with just Y. The disadvantage of HSV is that it is relatively costly to do the conversion back to RGB. YUV is much better from this viewpoint, as it is a linear inverse transformation that can be performed very efficiently in Matlab.

6.2 The script `ph_colourshift`

This script adopts the same general form as previous ones. In this case it comprises 7 cases selected by `mode`. The cases are:

Init initialises the **Colour shift** function. It opens a command figure (2) near the top of the screen which can take one of three forms, shown at the top of fig. 6.2, depending on which colour space is selected. It initialises the colour gains to unity and colour offsets to zero and defines labels for the three colour-space modes. Then it defines the 6 sliders and edit boxes for the 3 gains and 3 offsets, and their titles. It also defines the **Reset**, **Swap** and **Close** buttons. Finally, if the colour mode `cmode` is either YUV or HSV, it converts the **Before** image into YUV or HSV format as required. Then `mode` is set to **Colour** so that the sliders and boxes and their labels are all updated and a new colour-corrected image is generated.

Slider is called when any of the sliders are activated. It reads the slider values and converts them to `cgain(1:3)` in the range $0 \ldots 2$ and `cofst(1:3)` in the range $-0.5 \ldots 0.5$. It then sets `mode` equal to **Colour** so that the edit boxes are updated and colour correction is applied.

Edit box is called when any of the edit boxes are used to define `cgain` and `cofst` numerically. It sets these variables to the values in the edit boxes and sets `mode` equal to **Colour**, so that the sliders are updated and colour correction is applied.

Reset is called when this button is pressed. It resets all the `cgain` components to unity and all the `cofst` components to zero. It then sets `mode` equal to **Colour**, so that the sliders and edit boxes are updated and any previous colour correction is removed.

Swap is called when this button is pressed. This increments `cmode` modulo 3, so that it toggles between RGB, YUV and HSV colour modes, corresponding to values of 0, 1 and 2 respectively. It recalculates the YUV or HSV version of `xui` whenever one of these modes is entered. `cgain` and `cofst` are reset to their default values whenever `cmode` is changed, and again `mode` is set to **Colour**.

Close is called when this button is pressed. It closes figure 2 and redisplays **Before** and **After** in figure 1 to show the current and rotated images respectively.

Colour is the main colour-correction part of this script. It is called whenever any of the other cases or buttons redefine `mode` to be **Colour**. It updates the slider positions and the edit boxes to show the current values of `cgain` and `cofst` and then performs the required colour correction of `xui` to give `yui`. The correction is performed in the relevant colour mode, and if that is YUV or HSV, then the result is converted back to RGB in `yui`, which is displayed by `update_y`.

In RGB mode, the gain and offset corrections are applied directly to each colour slice of `xui` using:

```
yui(:,:,k) = uint8(cgain(k)*double(xui(:,:,k)) + cofst(k)*256);
```

Any out-of-range values are automatically limited to 0 or 255 by `uint8()`.

In YUV mode, the corrections are applied to `xyuv`, with different upper and lower limits for Y ($k = 1$) and for U, V:

```
if k==1, yyuv(:,:,k) = max(min(xyuv(:,:,k)*cgain(k)+cofst(k),1),0);
else yyuv(:,:,k) = max(min(xyuv(:,:,k)*cgain(k)+cofst(k),0.5),-0.5);
end
```

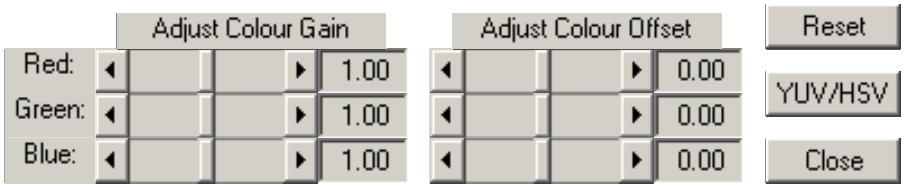
The same applies in HSV mode to `xhsv`, except that the H parameter ($k = 1$) is now evaluated modulo 1, rather than being limited by `max` and `min`, as the hue is a cyclic quantity over the range $0 \ldots 1$. The code is therefore:

```
if k==1, yhsv(:,:,k) = mod(xhsv(:,:,k)*cgain(k)+cofst(k),1);
else yhsv(:,:,k) = max(min(xhsv(:,:,k)*cgain(k)+cofst(k),1),0);
end
```

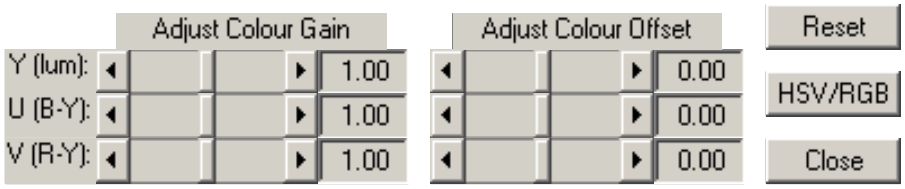
The functions for converting between RGB and YUV or HSV are given in figs. 6.3 to 6.6. From the discussions of colour conversions in section 6.1, their contents should be reasonably obvious so we will not discuss them further. The exception is `hsv2rgb()`, which is rather obscure; but it is not necessary to understand this function for this course – the function `rgb2hsv()` defines the mapping to HSV, which is all we need to be able to understand the HSV colour space.

The two HSV functions are in fact simplified versions of those supplied as standard Matlab functions. The many alternative options for different input and output data types have been stripped out, so they only work with 3-D colour image arrays of type `double`. This makes them a lot more human-readable. As long as our versions are in the current workspace, they will be used in preference to the standard Matlab ones with the same names.

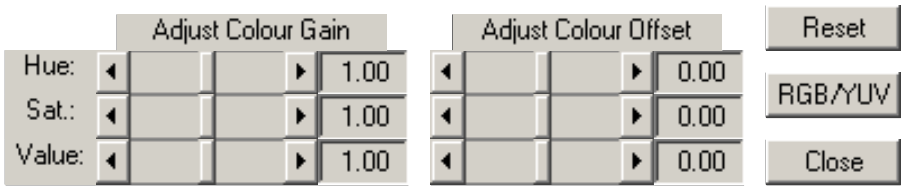
RGB Command window:



YUV Command window:



HSV Command window:




```
% ph_colourshift.m
% Routine called by 'Colour shift' menu item in photo editor.
%
% Nick Kingsbury, Cambridge University, March 2006.

switch mode

    case 'Init'
        % Set up figure window for Colour Shift commands
        figure(2);
        set(gcf,'position',[400  624  420  100]);
        set(gcf,'numbertitle','off','name','Colour Shift');

        % Initialise colour correction gains and cmode, if they have not
        % been used already.  cmode = 0 for RGB, 1 for YUV, 2 for HSV.
        if exist('cgain') ~= 1, cgain = [1 1 1]'; end
        if exist('cofst') ~= 1, cofst = [0 0 0]'; end
        if exist('cmode') ~= 1, cmode = 0; end

        % Buttons for colour selection.
        labelrgb = {'Red:','Green:','Blue:'};
        labelyuv = {'Y (lum):','U (B-Y):','V (R-Y):'};
        labelhsv = {'Hue:','Sat.:','Value:'};
```

Fig. 6.2 continued overleaf.

```
    for k = 1:3,
      slide(k) = uicontrol(gcf,'style','slider','sliderstep',[1 10]/100,...
       'pos',[50 70-20*k 100 20],'call','mode=''Slider''; ph_colourshift');
      lab(k) = uicontrol(gcf,'style','text','pos',[10 70-20*k 40 20]);
      edbox(k) = uicontrol(gcf,'style','edit','pos',[150 70-20*k 40 20],...
       'call','mode=''Edit box''; ph_colourshift');
      slide(k+3) = uicontrol(gcf,'style','slider','sliderstep',[1 10]/100,...
       'pos',[200 70-20*k 100 20],'call','mode=''Slider''; ph_colourshift');
      edbox(k+3) = uicontrol(gcf,'style','edit','pos',[300 70-20*k 40 20],...
       'call','mode=''Edit box''; ph_colourshift');
    end
    uicontrol(gcf,'style','text','str','Adjust Colour Gain',...
        'pos',[60 70 120 16]);
    uicontrol(gcf,'style','text','str','Adjust Colour Offset',...
        'pos',[210 70 120 16]);

    % Other buttons: reset, swap and close box.
    resetbal = uicontrol(gcf,'str','Reset','pos',[350 70 60 20],...
        'call','mode=''Reset''; ph_colourshift');
    swap = uicontrol(gcf,'str','YUV/HSV','pos',[350 40 60 20],...
        'call','mode=''Swap''; ph_colourshift');
    closebtn = uicontrol(gcf,'str','Close','pos',[350 10 60 20],...
        'call','mode=''Close''; ph_colourshift');

    % If YUV or HSV mode, pre-calculate xyuv or xhsv.
    if cmode == 1,
        xyuv=rgb2yuv(double(xui)/255);
        yyuv = xyuv;
    elseif cmode == 2,
        xhsv=rgb2hsv(double(xui)/255);
        yhsv = xhsv;
    end
    mode = 'Colour';

case 'Slider'
    for k=1:3,
        cgain(k) = 2*get(slide(k),'value');
        cofst(k) = get(slide(k+3),'value') - 0.5;
    end
    mode = 'Colour';

case 'Edit box'
    for k=1:3,
        cgain(k) = min(max(sscanf(get(edbox(k),'string'),'%f'),0),2);
        cofst(k) = min(max(sscanf(get(edbox(k+3),'string'),'%f'),-0.5),0.5);
    end
    mode = 'Colour';
```

Fig. 6.2 continued overleaf.

```
   case 'Reset'
        cgain = [1 1 1]';
        cofst = [0 0 0]';
        mode = 'Colour';

    case 'Swap'
        cmode = mod(cmode+1,3);
        if cmode == 1,
            xyuv=rgb2yuv(double(xui)/255);
            yyuv = xyuv;
        elseif cmode == 2,
            xhsv=rgb2hsv(double(xui)/255);
            yhsv = xhsv;
        end
        cgain = [1 1 1]';
        cofst = [0 0 0]';
        mode = 'Colour';

    case 'Close'
        close(2);
        newbefore = 1;
        showimages;

end

switch mode

    case 'Colour'
        for k=1:3,
            set(slide(k),'value',0.5*cgain(k));
            set(edbox(k),'string',sprintf('%.2f',cgain(k)));
            set(slide(k+3),'value',cofst(k)+0.5);
            set(edbox(k+3),'string',sprintf('%.2f',cofst(k)));
        end

        if cmode == 0, % RGB
            set(swap,'str','YUV/HSV');
            for k=1:3,
                set(lab(k),'str',labelrgb{k});
                yui(:,:,k) = uint8(cgain(k)*double(xui(:,:,k)) + cofst(k)*256);
            end
```

Fig. 6.2 continued overleaf.

```
        elseif cmode == 1, % YUV
            set(swap,'str','HSV/RGB');
            for k=1:3,
              set(lab(k),'str',labelyuv{k});
              if k==1, yyuv(:,:,k) = max(min(xyuv(:,:,k)*cgain(k)+cofst(k),1),0);
              else yyuv(:,:,k) = max(min(xyuv(:,:,k)*cgain(k)+cofst(k),0.5),-0.5);
              end
            end
            yui = uint8(255*yuv2rgb(yyuv));

        else % HSV
            set(swap,'str','RGB/YUV');
            for k=1:3,
              set(lab(k),'str',labelhsv{k});
              if k==1, yhsv(:,:,k) = mod(xhsv(:,:,k)*cgain(k)+cofst(k),1);
              else yhsv(:,:,k) = max(min(xhsv(:,:,k)*cgain(k)+cofst(k),1),0);
              end
            end
            yui = uint8(255*hsv2rgbnk(yhsv));
        end

        update_y;
        figure(2);

end
```

Fig. 6.2: Script to adjust colour: `ph_colourshift.m`

```
function y = rgb2yuv(x)
%
% function y = rgb2yuv(x)
%
% Convert a 3-layer RGB image x into a 3-layer YUV image y.
% The rules are:
%   Y = 0.3*R + 0.6*G + 0.1*B
%   U = 0.5*(B - Y) = 0.45*B - 0.15*R - 0.3*G
%   V = 0.625*(R - Y) = 0.4375*R - 0.375*G - 0.0625*B
%
% Nick Kingsbury, Cambridge University, 2006.

% Set up conversion matrix.
C = [0.3 0.6 0.1; -0.15 -0.3 0.45; 0.4375 -0.375 -0.0625];

% Convert each image layer into a column vector and multiply the 3 vectors by C'.
sx = size(x);
y = reshape(x,sx(1)*sx(2),sx(3)) * C';
y = reshape(y,sx); % Convert y back to the shape of x.

return
```

Fig. 6.3: Function to convert RGB to YUV: `rgb2yuv()`

```
function y = yuv2rgb(x)
%
% function y = yuv2rgb(x)
%
% Convert a 3-layer YUV image x into a 3-layer RGB image y.
% The rules are:
%   Y = 0.3*R + 0.6*G + 0.1*B
%   U = 0.5*(B - Y) = 0.45*B - 0.15*R - 0.3*G
%   V = 0.625*(R - Y) = 0.4375*R - 0.375*G - 0.0625*B
%
% Nick Kingsbury, Cambridge University, 2006.

% Set up conversion matrix.
C = [0.3 0.6 0.1; -0.15 -0.3 0.45; 0.4375 -0.375 -0.0625];
Ci = inv(C); % Invert the matrix.

% Convert each image layer into a column vector and multiply the 3 vectors by C'.
sx = size(x);
y = reshape(x,sx(1)*sx(2),sx(3)) * Ci';
y = reshape(y,sx); % Convert y back to the shape of x.

return
```

Fig. 6.4: Function to convert YUV to RGB: `yuv2rgb()`

```
function y = rgb2hsv(x)

%RGB2HSV Convert red-green-blue colors to hue-saturation-value.
% x and y must be 3-D colour image arrays, covering the range 0 to 1.
%
% Simplified from the standard matlab function by
% Nick Kingsbury, Cambridge University, 2006.

% Get colour slices
r = x(:,:,1); g = x(:,:,2); b = x(:,:,3);
siz = size(r);
r = r(:); g = g(:); b = b(:);

% Compute v and unnormalised s.
v = max(max(r,g),b);
s = zeros(size(v));
h = zeros(size(v));
s = (v - min(min(r,g),b));

% Compute h.
z = ~s; % z is used to set h=0 when s=0.
s = s + z;
k = find(r == v);
h(k) = (g(k) - b(k))./s(k);
k = find(g == v);
h(k) = 2 + (b(k) - r(k))./s(k);
k = find(b == v);
h(k) = 4 + (r(k) - g(k))./s(k);
h = h/6;
k = find(h < 0);
h(k) = h(k) + 1; % Adjust negative h to be between 0 and 1.
h=(~z).*h; % Set h=0 if s=0.

% Normalise s with v, unless v=0, when we make s=0 also.
k = find(v);
s(k) = (~z(k)).*s(k)./v(k);
k = find(~v);
s(k) = 0;

% Assemble h in correct 3D format.
h = reshape(h,siz);
s = reshape(s,siz);
v = reshape(v,siz);
y = cat(3,h,s,v);
return
```

Fig. 6.5: Function to convert RGB to HSV: `rgb2hsv()`

```
function y = hsv2rgb(x)

%HSV2RGB Convert hue-saturation-value colors to red-green-blue.
% x and y must be 3-D colour image arrays, covering the range 0 to 1.
%
% Simplified from the standard matlab function by
% Nick Kingsbury, Cambridge University, 2006.

% Get HSV slices.
h = x(:,:,1); s = x(:,:,2); v = x(:,:,3);
siz = size(h);
s = s(:); v = v(:);
h = 6*h(:); % Scale up hue to cover the range 0 to 6.

k = fix(h-6*eps); % Measure integer and fractional parts of hue.
f = h-k;
t = 1-s;
n = 1-s.*f;
p = 1-(s.*(1-f));
e = ones(size(h));
k0 = double(k==0); % These 6 lines increase speed over the Matlab hsv2rgb().
k1 = double(k==1);
k2 = double(k==2);
k3 = double(k==3);
k4 = double(k==4);
k5 = double(k==5);
% Compute r,g,b from h and s only.
r = k0.*e + k1.*n + k2.*t + k3.*t + k4.*p + k5.*e;
g = k0.*p + k1.*e + k2.*e + k3.*n + k4.*t + k5.*t;
b = k0.*t + k1.*t + k2.*p + k3.*1 + k4.*1 + k5.*n;

% Correct r,g,b for v component using f, and assemble y.
f = v./max([r(:);g(:);b(:)]);
y = zeros([siz,3]);
y(:,:,1) = reshape(f.*r,siz);
y(:,:,2) = reshape(f.*g,siz);
y(:,:,3) = reshape(f.*b,siz);
return
```

Fig. 6.6: Function to convert HSV to RGB: `hsv2rgb()`

7 Histograms and Lighting correction: `ph_lightshift`

The script `ph_lightshift` allows us to correct for lighting and exposure problems in images. It operates by adjusting the gain applied to all three components (R, G and B) of each pixel. This avoids any change of colour (hue) without the complexity of working in a non-RGB colour space. We generate a gain map, equal in size to the image, such that if a pixel intensity needs to be increased then a gain of greater than unity is used, and if it needs to be reduced we use a gain of less than unity.

7.1 Lighting correction methods

If we examine the histogram of a poorly lit image, we will usually find that some pixel intensities are used much more frequently than others. The lighting problems may often be significantly reduced if an intensity mapping process is employed that tends to make more uniform use of all intensity levels. One way of achieving this is **histogram equalisation**.

It may be shown that, for a monochrome imaging system with intensity levels from 0 to 255, that the histogram will be approximately equalised if the following mapping of intensity levels is used:

$$y_k = \sum_{i=0}^{k} \frac{255\ n_i}{N} \qquad \text{where} \quad N = \sum_{i=0}^{255} n_i \tag{7.1}$$

and each pixel of level k is replaced by one of level y_k, and n_i is the number of pixels in the image with level i (i.e. the set of $n_i, i = 0 \ldots 255$, forms the histogram of the image). Note that in order to apply equalisation to **colour** images, we need to calculate the histogram of the luminance image (using equation (6.1)) to obtain the values for n_i and then scale the R, G and B components of any given pixel by y_k/k where k is the luminance value for that pixel. This preserves the hue and saturation of the pixel while changing its luminance.

In our editor, equalisation is achieved by pressing the button **Hist equal** in the Lighting Shift window.

While this technique can be very effective at bringing out detail in regions of the image where there are many pixels of similar luminance levels (since n_i is relatively large at these luminance levels), it also tends to lose resolution at other levels where there are few pixels and n_i is small. In our Genoa image, this can be seen as a loss of contrast in areas such as around the buildings near the sea. Hence we need something more natural looking.

Suppose we have an image with areas that are too dark. To enhance these regions we must apply a gain greater than unity to them, but we must also avoid saturating the pixels in bright areas. **Gamma correction** is one method of achieving this. It uses a non-linear mapping of the form:

$$y_k = 255\ (k/255)^{1/\gamma} \tag{7.2}$$

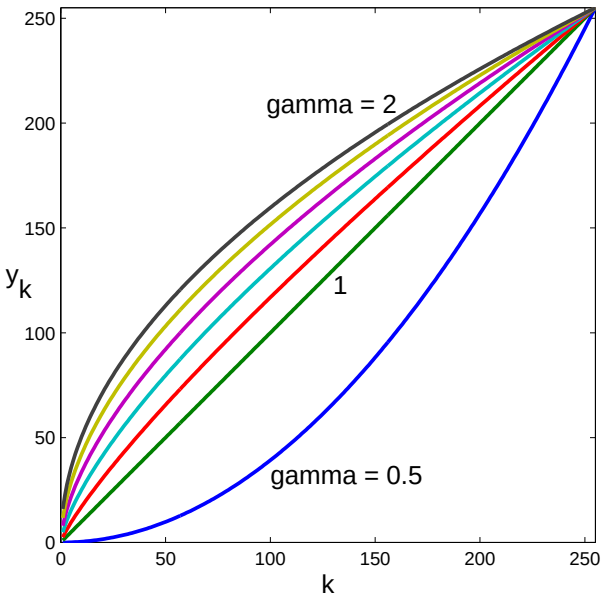
If $\gamma = 1$, this gives $y_k = k$ (i.e. no correction), but if $\gamma > 1$ then low luminance pixels are scaled up while those with higher intensities are left almost unchanged (since $y_{255} = 255$ for any value of γ). Fig. 7.1a shows plots of y_k against luminance k for $\gamma = 0.5, 1, 1.2, 1.4, 1.6, 1.8$ and 2. As before, in practice R, G and B are scaled by y_k/k to preserve colours correctly. The value of γ can be selected by the user (with the Gamma slider in figure(2)) so that

approximate histogram equalisation is achieved without the severe loss of contrast achieved by the true histogram equalisation curve (shown as a thin line in fig. 7.1b).

Finally we consider a more flexible technique for lighting correction, which allows shadows, mid-tones and highlights to be treated in different ways. The concept is similar to gamma correction, except that a more flexible shape is achieved by specifying the locations of knot-points on a piece-wise linear curve. Fig. 7.1b shows such a curve, with circles showing the knot-points. The elements of the variable `light` define how far (as a proportion of 256) each of the three knot-points at 64, 128 and 192 are raised or lowered. We see how the true histogram equalisation curve can be approximated by the piece-wise linear one, while avoiding the extreme reductions in gradient that caused the loss of contrast for mid-range luminance levels.

One further trick to improve the appearance of the corrected image, is to base the correction gain on values of k which correspond to the luminance of a **blurred** version of the input image. This ensures that the gain only fluctuates smoothly over the image, and therefore tends to preserve any local intensity variations of the input image, even when the correction curve is operating in a fairly low-gradient region. The one disadvantage of this is that it can produce a (slight) bright halo around dark objects. We find that a 2-D Gaussian blurring filter (see section 8 of this course) with breadth parameter b (also known as standard deviation or half-width) of between 2 and 5 pels gives good results. We have chosen a value of $b = 4$ pels for this parameter.

(a) Gamma Correction



(b) Piecewise Linear Correction

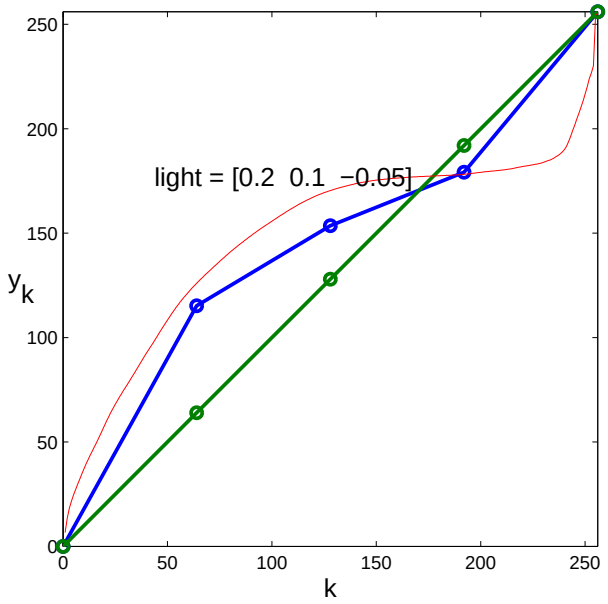


Fig. 7.1: Gamma and piecewise linear correction methods, based on adjusting pixels with luminance k to have luminance y_k. The thin line in (b) is the histogram equalisation curve for the Genoa image.

7.2 The script `ph_lightshift`

This script adopts the same general form as the others. It comprises 7 cases selected by `mode`, one of which, **Lighting**, uses a separate **switch** statement at the end so that it can be executed after other cases have been executed before it. The cases are:

Init initialises the **Lighting shift** function. It opens a command figure (2) near the top of the screen which takes the form shown at the top of fig. 7.2. It initialises the correction offsets `light` to zero and `cgamma` to unity. It then sets up the four sliders and their associated edit boxes and labels, followed by the buttons for **Reset**, **Hist equal** and **Close**. Then `mode` is set to **Lighting** so that the sliders and boxes and their labels are all updated and a new lighting-corrected image is generated.

Slider is called when any of the sliders are activated. It reads the slider values and converts them to `light(1:3)` in the range $-0.25 \cdots 0.25$ and `cgamma` in the range $0.1 \cdots 2$. It then sets `mode` equal to **Lighting** so that the edit boxes are updated and lighting correction is applied.

Edit box is called when any of the edit boxes are used to define `light` and `cgamma` numerically. It sets these variables to the values in the edit boxes and sets `mode` equal to **Lighting**.

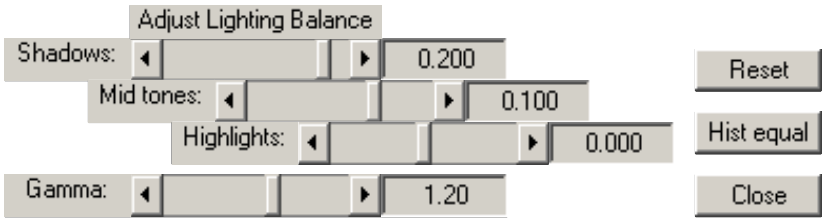
Reset is called when this button is pressed. It resets `light(1:3)` to zero and `cgamma` to unity. It then sets `mode` equal to **Lighting** so that the edit boxes are updated and `yui` equals `xui`.

Hist equal is called when this button is pressed. It performs `yui = im_histeq(xui);` and displays the result using `update_y`.

Close is called when this button is pressed. It closes figure 2 and redisplays **Before** and **After** in figure 1 to show the current and corrected images respectively.

Lighting is called whenever any of the other cases or buttons redefine `mode` to be **Lighting**. It updates the slider positions and the edit boxes to show the current values of `light` and `cgamma`. It then performs the required lighting correction of `xui` to give `yui`, by a call to `im_lighting()` and displays the result.

Command window figure(2):




```
% ph_lightshift.m
% Routine called by 'Lighting shift' menu item in photo editor.
%
% Nick Kingsbury, Cambridge University, 2006.

switch mode

    case 'Init'
        % Set up figure window for Rotate commands
        figure;
        set(gcf,'position',[400  610  410  120]);
        set(gcf,'numbertitle','off','name','Lighting Shift');

        % Initialise colour correction gains
        if exist('light') ~= 1, light = [0;0;0]; end
        if exist('cgamma') ~= 1, cgamma = 1; end

        % Buttons for different lighting ranges.
        label = {'Shadows:','Mid tones:','Highlights:','Gamma:'};
        for k = 1:4,
          if k < 4, ypos = 95 - 20*k; xpos = 40*k - 30;
          else ypos = 10; xpos = 10;
          end
          slide(k) = uicontrol(gcf,'style','slider','sliderstep',[1 10]/100,...
            'pos',[xpos+60 ypos 120 20],'call','mode=''Slider''; ph_lightshift');
          uicontrol(gcf,'style','text','str',label{k},'pos',[xpos ypos 60 20]);
          edbox(k) = uicontrol(gcf,'style','edit','pos',[xpos+180 ypos 60 20],...
            'call','mode=''Edit box''; ph_lightshift');
        end
        uicontrol(gcf,'style','text','str','Adjust Lighting Balance',...
            'pos',[70 95 120 16]);

        % Other buttons: reset and close box.
        resetbtn = uicontrol(gcf,'str','Reset','pos',[340 70 60 20],...
            'call','mode=''Reset''; ph_lightshift');
        histequ = uicontrol(gcf,'str','Hist equal','pos',[340 40 60 20],...
            'call','mode=''Hist equal''; ph_lightshift');
        closebtn = uicontrol(gcf,'str','Close','pos',[340 10 60 20],...
            'call','mode=''Close''; ph_lightshift');
        mode = 'Lighting';
```

Fig. 7.2 continued overleaf.

```
    case 'Slider'
        for k = 1:3,
            light(k) = 0.5*(get(slide(k),'value') - 0.5);
            set(edbox(k),'string',sprintf('%.3f',light(k)));
        end
        cgamma = max(2*get(slide(4),'value'),0.1);
        set(edbox(4),'string',sprintf('%.2f',cgamma));
        mode = 'Lighting';

    case 'Edit box'
        for k = 1:3,
            light(k) = sscanf(get(edbox(k),'string'),'%f');
            set(slide(k),'value',2*light(k) + 0.5);
        end
        cgamma = max(sscanf(get(edbox(4),'string'),'%f'),0.1);
        set(slide(4),'value',0.5*cgamma);
        mode = 'Lighting';

    case 'Reset'
        light = [0;0;0];
        cgamma = 1;
        mode='Lighting';

    case 'Hist equal'
        yui = im_histeq(xui);
        update_y;
        figure(2);

    case 'Close'
        close(2);
        newbefore = 1;
        showimages;
end

switch mode
    case 'Lighting'
        % Perform lighting correction.
        if all(light == 0) & cgamma == 1, yui = xui;
        else [yui,light] = im_lighting(xui,light,cgamma);
        end
        % Update sliders and edit boxes from light and replot yui data.
        for k=1:3,
            set(slide(k),'value',2*light(k) + 0.5);
            set(edbox(k),'string',sprintf('%.3f',light(k)));
        end;
        set(slide(4),'value',0.5*cgamma);
        set(edbox(4),'string',sprintf('%.2f',cgamma));
        update_y;
        figure(2);
end
```

Fig. 7.2: Script to perform lighting correction: `ph_lightshift.m`

7.3 The function `im_histeq()`

This function performs histogram equalisation of a colour image and is shown in fig. 7.3. It is similar in function to the Matlab function `histeq()`, which is part of the Image Processing toolbox. However our function only deals with 3-D colour input and output data of type uint8, and so is much simpler.

It calculates `histL`, the histogram of the luminance `L` of `xui`, which it then converts to `yk` using equation (7.1). Then `gain` is calculated from y_k/k at each pel and is applied to each colour slice of `xui` to produce `yui`.

```
function yui = im_histeq(xui)

% function yui = im_histeq(xui)
%
% Perform histogram equalisation on xui.
%
% Nick Kingsbury, Cambridge University, 2006.

wh = waitbar(1/5,'Equalising histogram');

% Calculate intensity image and its 256-bin histogam.
L = 0.3*double(xui(:,:,1)) + 0.6*double(xui(:,:,2)) + 0.1*double(xui(:,:,3)) + 1;
histL = hist(L(:),[1:256]);
waitbar(2/5);

% Uncomment the next line to reduce the severity of the equalisation
% by adding a constant to all histogram bins.
% histL = histL + size(xui,1)*size(xui,2)/256;

yk = cumsum(255*histL/sum(histL));
% save histeq yk histL

gain = yk(round(L)) ./ L;
yui = uint8(zeros(size(xui)));
for k = 1:3, % Apply gain matrix to each colour slice.
    waitbar((k+2)/5);
    yui(:,:,k) = uint8(gain .* double(xui(:,:,k)) + 0.5);
end
close(wh) % Close wait bar
return
```

Fig. 7.3: Function to perform histogram equalisation of an image: `im_histeq.m`

7.4 The function `im_lighting()`

This function, shown in fig. 7.4, performs both gamma correction and piece-wise linear correction of pixel intensities. The final gain applied to each pixel is the product of the gain from gamma correction and the gain from piece-wise linear correction.

First the luminance image `L1` is calculated, and then blurred by a 2-D Gaussian filter to produce `L`. The blurring is achieved by applying a 1-D filter twice, first down the columns of `L1`, and then down the columns of the transposed result. The column filtering function `colfilter` is used for this and is discussed in the next section. The x and y coordinates of the knot-points, `tx` and `ty`, are calculated from the 3 elements of the input parameter `light`. They are adjusted so that the gradient of the correction curve is always positive. The three sliders for shadows, mid-tones and highlights define `light(1:3)` and the adjustment for positive gradient always ensures that the sliders (which are staggered) do not 'overtake' each other.

In order to calculate the gain functions from piece-wise linear law, we first calculate `grad(1:4)`, the gradient of the four segments of the piece-wise linear function. Then we find which pels belong to each segment (sets `s0` to `s3`), and calculate the values of `gain` for all pels within each set using a linear interpolation based on the gradient and start knot-point of that segment. The blurred luminance image `L` is used for this.

After this the gamma correction gain is combined with the previous gain matrix. For gamma correction, we use the unblurred luminance image `L1`, because this gives less visible noise in the enhanced dark regions of the image.

Finally in the for-loop, the gain matrix is applied to the 3 colour slices of `xui` to produce the lighting-corrected image `yui`.

```
function [yui,light] = im_lighting(xui,light,cgamma)

% Adjust lighting of xui using light.
% The 3 elements of light define deviations of a piecewise-linear
% lighting correction function from unit gradient as follows:
% tx = [64 128 192]'; % Intensity points in xui.
% ty = tx + 256 * light; % Equivalent intensities in yui.
% cgamma defines gamma correction, and should be 1 for no effect.
%
% Nick Kingsbury, Cambridge University, 2006.

wh = waitbar(0,'Relighting image');
sx = size(xui);

% Calculate luminance image.
L1 = 0.3*double(xui(:,:,1)) + 0.6*double(xui(:,:,2)) + 0.1*double(xui(:,:,3));
```

Fig. 7.4 continued overleaf.

```
% Blur the luminance image, so as to give a smooth segmentation of xui
% into shadows, mid-tones and highlights.
h = gaussian(4);
waitbar(1/5);
L = colfilter(colfilter(L1,h)',h)';
waitbar(2/5);
% figure(3);image(uint8(L));colormap(gray(256));pause(0.1)

% Define the correction function to give equivalent intensity points
% in xui and yui.
tx = [64 128 192]'; % Intensity points in xui.
ty = tx + 256 * light; % Equivalent intensities in yui.
ty(2) = max(ty(2),ty(1)+1); % Ensure gradient is always positive.
ty(3) = max(ty(3),ty(2)+1);
if ty(3)>254, error('Inconsistent lighting vector'); end
light = (ty - tx)/256; % Recalculate light to allow for limiting.

% Calculate gradients of the 4 segments of the correction function.
grad = ([ty;255] - [0;ty])./([tx;255] - [0;tx]);

% Find pixels in each segment of the correction function.
s0 = find(L < tx(1));
s1 = find(L >= tx(1) & L < tx(2));
s2 = find(L >= tx(2) & L < tx(3));
s3 = find(L >= tx(3));

% Calculate gain to be applied to each pixel.
gain = ones(sx(1:2));
gain(s0) = grad(1);
gain(s1) = (ty(1) + (L(s1)-tx(1))*grad(2))./L(s1);
gain(s2) = (ty(2) + (L(s2)-tx(2))*grad(3))./L(s2);
gain(s3) = (ty(3) + (L(s3)-tx(3))*grad(4))./L(s3);

% Apply gamma correction to the gains, based on the luminance image L1.
gain = gain .* ((max(L1,1)/255).^(1/cgamma - 1));
% figure(4);image(uint8(gain*64));colormap(gray(256));pause(0.1)

% Perform the relighting using the matrix, gain.
yui = xui;
for k = 1:3,
    waitbar((k+2)/5);
    yui(:,:,k) = uint8(double(xui(:,:,k)) .* gain + 0.5);
end
close(wh) % Close wait bar
return
```

Fig. 7.4: Function to perform lighting correction of an image: `im_lighting.m`

8 Filtering the image: `ph_filter`

The script `ph_filter` is concerned with filtering the image, using lowpass or highpass filters or some combination of the two.

Before looking at the code of `ph_filter`, we shall look at some of the characteristics of the two types of filter and some of the maths that governs this.

8.1 Lowpass Filtering with Gaussian filters

Lowpass filters tend to pass the lower frequency components of the image (smooth regions) and reject the higher frequency components (fine details). Hence they **smooth** the image. This has the beneficial effect of reducing any visible noise in the image, but unfortunately it also tends to blur any sharp edges and fine textures.

We shall consider one particular type of lowpass filter in this application. It is the Gaussian filter, which, in its unit-variance form in the time domain, has an impulse response:

$$g(t) = \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-t^2}{2}\right) \tag{8.1}$$

The Gaussian impulse response is unique, in that its frequency response is also a Gaussian-shaped function, given by:

$$G(\omega) = \int_{-\infty}^{\infty} g(t) \exp(-j\omega t)\, dt = \exp\left(\frac{-\omega^2}{2}\right) \tag{8.2}$$

It is the only function which has the same shape in both the time and frequency domains. Its other key property is that the area under $g(t)$ integrates to unity, as long as the $1/\sqrt{2\pi}$ scaling factor is present on the RHS of equ. (8.1). This can be shown by setting $\omega = 0$ in equ. (8.2). It gives the filter unit gain at lower frequencies, which is usually a necessary pre-requisite for successful image display.

The above Fourier transform relation is a bit complicated to derive, so we have included its full derivation in section 8.4 (effectively an appendix) for those who are interested.

The Gaussian filter can be seen to be a **Lowpass** filter, because $G(\omega) \to 1$ when $\omega \to 0$, and $G(\omega) \to 0$ when $|\omega| \to \infty$. Hence low frequency components of the signal are passed with almost unit gain, while high frequency components are rejected with approximately zero gain.

2-D filtering

To create a lowpass filter for a 2-D signal such as an image, it is usual to apply a 1-D filter first to the columns of the image and then to the rows (or vice-versa). If the image intensity $I(x, y)$ is a function of position (x, y), then the result of convolving the filter with the columns of the image is a new image:

$$I_c(x, y) = \int I(x, y - v)\, g_c(v)\, dv \tag{8.3}$$

where $g_c(y)$ is the impulse response of the column filter and, from now on, all integrals are assumed to be over infinite limits unless stated otherwise.

If we now convolve the rows of $I_c(x, y)$ with the row filter, whose impulse response is $g_r(x)$, we get:

$$\begin{aligned} I_{r,c}(x,y) &= \int I_c(x-u,y)\; g_r(u)\; du \\ &= \int \left[\int I(x-u,y-v)\; g_c(v)\; dv \right] \; g_r(u)\; du \\ &= \int \int I(x-u,y-v)\; g_r(u)\; g_c(v)\; du\; dv \\ &= \int \int I(x-u,y-v)\; g_{r,c}(u,v)\; du\; dv \\ &\quad \text{where} \;\; g_{r,c}(u,v) \;=\; g_r(u)\; g_c(v) \end{aligned} \tag{8.4}$$

This is the expression for convolution in 2-D, using a filter with a 2-D impulse response (also known as a *point-spread function*) given by $g_{r,c}(x, y) = g_r(x)\; g_c(y)$.

We shall use the Gaussian filter $g(t)$ for $g_r(x)$ and $g_c(y)$ with a constant breadth scale factor b, such that t is replaced by x/b and y/b. Hence:

$$g_r(x) = \frac{1}{b}\; g\left(\frac{x}{b}\right) \qquad \text{and} \qquad g_c(y) = \frac{1}{b}\; g\left(\frac{y}{b}\right) \tag{8.5}$$

The extra factors of $1/b$ are needed above, in order to keep the important property of unit gain at low frequencies, which ensures that the filtered image will occupy approximately the same range of intensity levels as before for larger patches of similar colour. This can be seen if we calculate the frequency response $G_r(\omega)$ for the filter $g_r(x)$:

$$\begin{aligned} G_r(\omega) &= \int g_r(x) \exp(-j\omega x)\, dx \\ &= \int \frac{1}{b}\; g\left(\frac{x}{b}\right) \exp(-j\omega x)\, dx \\ &= \int g(t) \exp(-j\omega b t)\, dt \qquad \text{where} \;\; t = \frac{x}{b} \;\; \text{and} \;\; dt = \frac{dx}{b} \\ &= G(\omega b) \;=\; \exp\left(\frac{-\omega^2 b^2}{2}\right) \end{aligned} \tag{8.6}$$

(a) Impulse response $g_r(x)$ (b) Frequency response $G_r(\omega)$

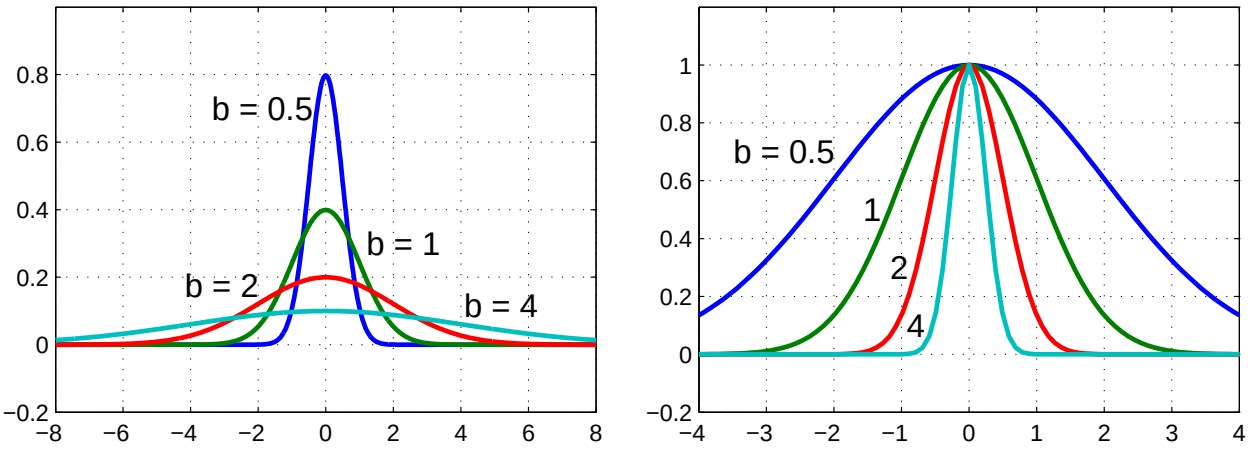


Fig. 8.1: Impulse responses and frequency responses of 1-D Gaussian lowpass filters, $g_r(x)$, when the breadth parameter $b = 0.5, 1, 2, 4$.

Hence $G_r(0) = G(0) = 1$, as required. The frequency response expression also shows us that, as b is increased, the bandwidth of the filter reduces, and the filter will produce more blurring of the image. For example, when $\omega b = 1$, the gain of the filter will have reduced from unity to $\exp(-0.5) = 1/\sqrt{e} = 0.6065$, as shown for several values of b in fig. 8.1.

One of the main reasons why Gaussian filters are attractive for image processing is that they are the only filter shape which produces a truly isotropic (omni-directional) 2-D impulse response from separate row and column filtering stages. This can be seen if we calculate the 2-D impulse response $g_{r,c}(x, y)$ as follows:

$$\begin{aligned} g_{r,c}(x,y) &= g_r(x)\; g_c(y) \;=\; \frac{1}{b}\, g\!\left(\frac{x}{b}\right)\; \frac{1}{b}\, g\!\left(\frac{y}{b}\right) \\ &= \frac{1}{\sqrt{2\pi}\; b}\; \exp\left(\frac{-x^2}{2b^2}\right) \frac{1}{\sqrt{2\pi}\; b}\; \exp\left(\frac{-y^2}{2b^2}\right) \\ &= \frac{1}{2\pi\, b^2}\; \exp\left(\frac{-(x^2+y^2)}{2b^2}\right) \end{aligned} \tag{8.7}$$

If we convert (x, y) to polar coordinates (r, θ), then we find that

$$g_{r,c}(r,\theta) \;=\; \frac{1}{2\pi\, b^2}\; \exp\left(\frac{-r^2}{2b^2}\right) \tag{8.8}$$

This is independent of θ, and so the 2-D impulse response is circularly symmetric and the filter will have the same effect on the image in all directions – i.e. it will be **isotropic**. It is shown in fig. 8.2, for the case when $b = 1$ (i.e. when the breadth of the response, between opposite points where the amplitude is $1/\sqrt{e}$ of its peak value, is $2b = 2$). The contour plot is included to show the circular symmetry of the response, and the contour at 0.1 is approximately the level at which the breadth is $2b$ (actually it is at $1/(2\pi\sqrt{e}) = 0.0965$).

The frequency response of this filter is also isotropic – but we shall not prove this here.

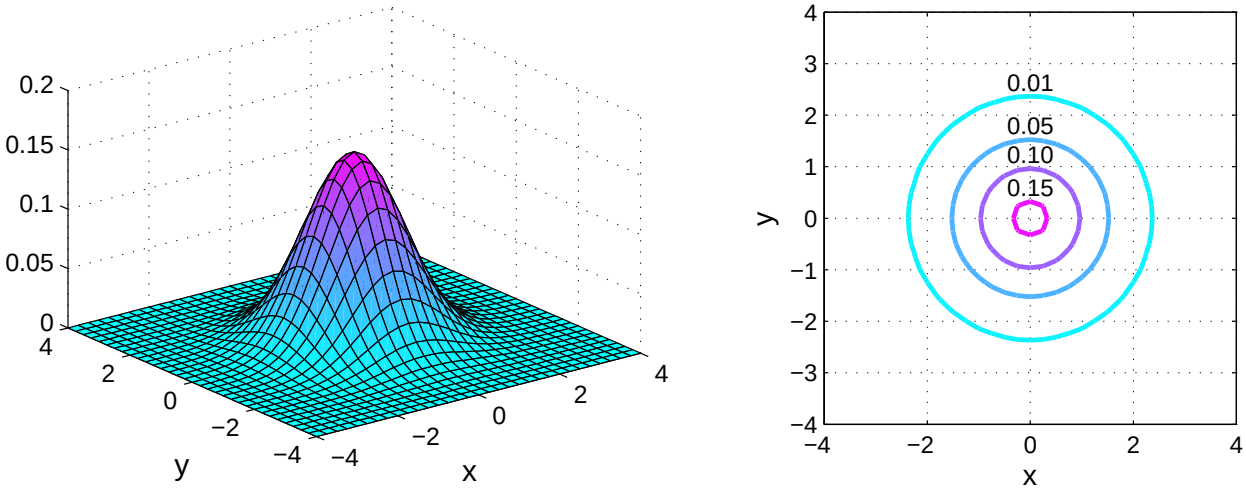


Fig. 8.2: Impulse response of a 2-D Gaussian lowpass filter, $g_{r,c}(x, y)$, when the breadth parameter $b = 1$ in equ. (8.7). A mesh plot is on the left, and a contour plot is on the right with the contour levels shown.

Computational issues of separable filters

It is worth noting here that isotropic (circularly symmetric) filters with *arbitrary* (non-Gaussian) shapes may be created by a suitable choice of function for $g_{r,c}(x, y)$, but that such functions cannot be factorised into the **separable** form $g_r(x)g_c(y)$, unless they are Gaussian. The main problem with this is that the convolution of an image directly with a 2-D function is usually much more computationally expensive than two separate convolutions with 1-D functions along the rows and the columns in turn.

For example, consider a 1-D Gaussian filter with breadth parameter $b = 5$. The length N of the sampled impulse response of this filter needs to be 31 samples if all terms greater than 1% of the peak amplitude are retained. To convolve this with the columns of an image, requires N multiplications and additions per pel of the image. The same amount of computation is required for convolving this filter with the rows of the image. Hence the total computation needed is $2N$ multiplications and additions per pel. If instead the response was non-separable and we had to convolve the $N \times N$ equivalent 2-D impulse response with each pel of the image, this would require N^2 multiplications and additions per pel of the image. If $N = 31$, this would be $N^2/2N = N/2 = 15.5$ times as much computation – a substantial increase if the image size is large!

The Matlab function `conv2(x,g)`, which is used in our function `colfilter()`, has a computation time which is proportional to the product of the areas of the image `x` and the filter `g`. Hence it exhibits exactly the behaviour described above, and it is much more efficient to do two convolutions of `x` with 1-D filters of area $N \times 1$ samples, than it is to do a single convolution with a 2-D filter of area $N \times N$ samples.

This is therefore a key reason why separable Gaussian filters are a preferred way of obtaining isotropic 2-D filter responses.

Functions to implement the filters

Our function `im_lowpass.m` to perform 2-D lowpass filtering is shown in fig. 8.3. We pass it the image to be filtered `x` and the standard deviation `sigma` of the Gaussian filter (this is our breadth parameter b above). It calls the function `gaussian()` to calculate the impulse response `g` and then performs the filtering on each colour slice by two calls to the column filtering function `colfilter()`, using the transpose operator to swap rows and columns between the two calls. These functions are listed in figs. 8.4 and 8.5.

`gaussian(sigma,thresh)` takes the value of `sigma` and creates a vector of samples of a Gaussian pulse. The length of the vector is proportional to `sigma`, and is long enough to include all samples of the Gaussian that are at least `thresh` times the peak value. The length of the output vector is always odd, and its centre sample is the peak. The default value for `thresh` is 0.01. The sum of the vector is normalised to unity (for unit gain of the filter at zero frequency).

```
function y = im_lowpass(x,sigma);
% function y = im_lowpass(x,sigma);
%
% Perform 2-D Gaussian lowpass filtering on colour image x.
%
% Nick Kingsbury, Cambridge University, 2006.

wh = waitbar(0,'Filtering image');

% Generate 1-D Gaussian impulse response.
g = gaussian(sigma);

% Filter each colour slice of x.
y = x;
for k=1:3,
    waitbar(k/4);
    y(:,:,k) = colfilter(colfilter(y(:,:,k),g).',g).';
end
waitbar(1);
y = max(min(y,255),0);
close(wh) % Close wait bar
return;
```

Fig. 8.3: Function to perform lowpass filtering of an image: `im_lowpass()`

```
function h = gaussian(sigma,thresh)
% function h = gaussian(sigma,thresh)
%
% Generate a gaussian vector / impulse response, h, with
% a standard deviation of sigma samples and a total value of unity.
% The length of h is odd and is truncated at the points where
% exp(-x^2/2) < thresh. By default, thresh = 0.01.
%
% Nick Kingsbury, Cambridge University, Nov 2005.

if nargin < 2, thresh = 0.01; end

% Solve for when exp(-x^2 / 2*sigma^2) = thresh
xmax = sigma * sqrt(max(-2*log(thresh),1e-6));

% Calculate h over the range when exp(-x^2/2) >= thresh.
n = floor(xmax);
x = [-n:n]';
h = exp(x.*x/(-2*sigma*sigma));
h = h / sum(h); % Normalise h so it sums to unity.
return;
```

Fig. 8.4: Function to calculate the coefficients of a Gaussian lowpass filter: `gaussian()`

```
function Y = colfilter(X, h)
% function Y = colfilter(X, h)
% Filter the columns of image X using filter vector h, without decimation.
% If length(h) is odd, each output sample is aligned with each input sample
% and Y is the same size as X.
% If length(h) is even, each output sample is aligned with the mid point
% of each pair of input samples, and size(Y) = size(X) + [1 0];
%
% Cian Shaffrey, Nick Kingsbury, Cambridge University, August 2000

[r,c] = size(X);
m = length(h);
m2 = fix(m/2);
if any(X(:))  % Test for any non-zero samples in X.
   % Symmetrically extend row indices with repeat of end samples.
   xe = reflect([(1-m2):(r+m2)], 0.5, r+0.5);

   % Perform filtering on the columns of the extended matrix X(xe,:), keeping
   % only the 'valid' output samples, so Y is the same size as X if m is odd.
   Y = conv2(X(xe,:),h(:),'valid');
else
   Y = zeros(r+1-rem(m,2),c); % Short cut if X is all zeros.
end
return;
```

Fig. 8.5: Function to perform filtering down the columns of an image matrix: `colfilter()`

`colfilter(X,h)` filters the columns of matrix `X` using the efficient built-in Matlab function `conv2`. The main complication concerns how we deal with filtering at the image boundaries. For images, it is usually best if we assume that the image is *mirrored* at its boundaries. Hence we create extended columns for `X` using an index vector `xe` which is of the form:

$$[m2, m2-1 \quad \ldots \quad 1, 1, 2 \quad \ldots \quad r-1, r, r \quad \ldots \quad r-m2+1]$$

where r is the height of the columns of `X` and $m2$ is the half-length of the filter `h`. The matrix `X(xe,:)` then has the symmetrically extended columns needed for correct filtering by `conv2` in its **valid** mode, so as to generate a filtered image the same size as `X`.

```
function y = reflect(x,minx,maxx)
% function y = reflect(x,minx,maxx)
% Reflect the values in matrix x about the scalar values minx and maxx.
% Hence a vector x containing a long linearly increasing series
% is converted into a waveform which ramps linearly up and down
% between minx and maxx.
% If x contains integers and minx and maxx are (integers + 0.5),
% the ramps will have repeated max and min samples.
%
% Nick Kingsbury, Cambridge University, January 1999.

y = x;

% Reflect y in maxx.
t = find(y > maxx);
y(t) = 2*maxx - y(t);

% Reflect y in minx.
t1 = find(y < minx);
t2 = 0;
while ~isempty(t1) | ~isempty(t2), % Repeat until no more values out of range.
   if ~isempty(t1), y(t1) = 2*minx - y(t1); end
   t2 = find(y > maxx);
   if ~isempty(t2), y(t2) = 2*maxx - y(t2); end
   t1 = find(y < minx);
end
return;
```

Fig. 8.6: Function to reflect members of a sequence about max and min boundary values: `reflect()`

The function `y = reflect(x,minx,maxx)`, shown in fig. 8.6, is responsible for converting a monotonically increasing vector

$$x = [-m2+1, -m2+2 \quad \ldots \quad 0, 1, 2 \quad \ldots \quad r-1, r, r+1 \quad \ldots \quad r+m2-1, r+m2]$$

into

$$y = [m2, m2-1 \quad \ldots \quad 1, 1, 2 \quad \ldots \quad r-1, r, r \quad \ldots \quad r-m2+1]$$

which is the symmetrically reflected vector given on the earlier page. In this case, `minx = 0.5` and `maxx = r + 0.5`. The code is a bit complicated because it is designed to correctly handle the case when $r < m2$ and several reflections are needed at each boundary.

8.2 Highpass Filtering with Gaussian filters

Highpass filters are filters with greater gain at high frequencies than at low frequencies. They are usually used for enhancing edges of objects, or for detecting them so as to control other processing operations.

They may be generated by subtracting a lowpass filtered signal from the unfiltered signal, with suitably chosen scale factors α and β. This can be illustrated for 1-D signals in the frequency domain as follows:

$$Y(\omega) \;=\; \alpha\, X(\omega) - \beta\, G(\omega)\, X(\omega) \;=\; X(\omega)[\alpha - \beta\, G(\omega)] \tag{8.9}$$

Hence the filter frequency response will be

$$H(\omega) \;=\; \frac{Y(\omega)}{X(\omega)} \;=\; \alpha - \beta\, G(\omega) \tag{8.10}$$

If $G(\omega)$ is lowpass with unit gain at low frequencies and zero gain at high frequencies, then the gain of $H(\omega)$ will tend to $(\alpha - \beta)$ at low frequencies and α at high frequencies. Hence we may create a 1-D highpass filter, with unit gain at low frequencies and a gain of say 2 at high frequencies, by choosing $(\alpha - \beta) = 1$ and $\alpha = 2$ (hence $\beta = 1$).

The effect of this type of filter on a step edge is illustrated in fig. 8.7, by waveform (c), which is α times waveform (a) minus β times waveform (b). This type of filter is appropriate for enhancing the sharpness of edges in an image without losing the important low-frequency content of the image.

If instead we choose $(\alpha - \beta) = 0$, then we get zero gain at low frequencies and the effect of this type of highpass filter on a step edge is shown in fig. 8.7(d). This type of filter is suitable for detecting edges (using the absolute magnitude of the filter output) and ignoring the smoothly varying parts of the signal.

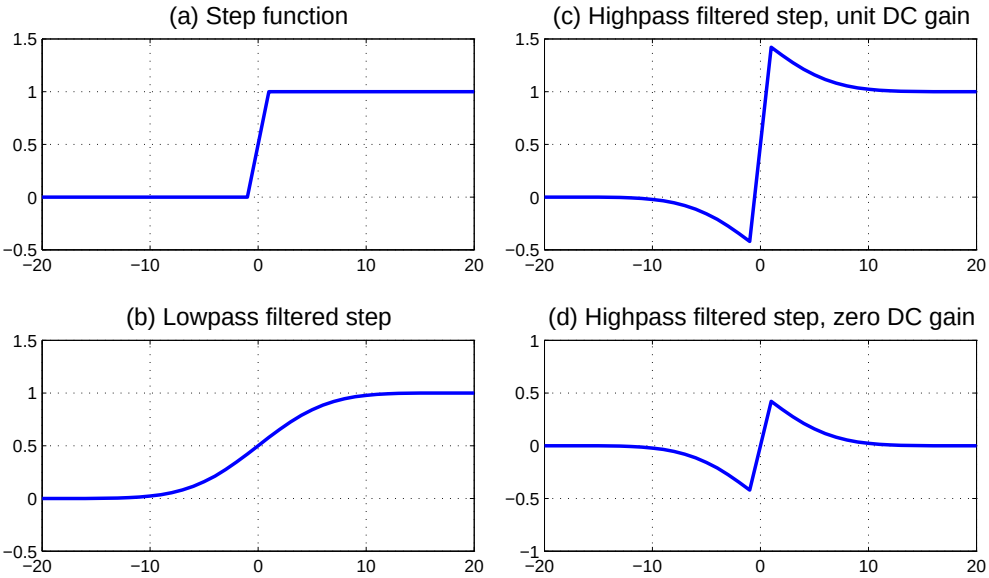


Fig. 8.7: (a) An input step function; (b) the step, filtered by a Gaussian lowpass filter (breadth $b = 5$); (c) the result of subtracting (b) from twice (a) to obtain a highpass filter with unit gain at low frequencies; (d) the result of subtracting (b) from (a) to obtain a highpass filter with zero gain at low frequencies.

To obtain a 2-D highpass filter, we could simply perform a pair of 1-D highpass filtering operations on the columns and then the rows of our image, as we did for the lowpass case. However it is not possible to get an isotropic impulse response this way.

To get an **isotropic** highpass response, using separate 1-D row and column filtering for efficiency, we must first create a Gaussian lowpass filtered version of our image and then subtract it from the original image with suitable scaling factors α and β, as explained above for 1-D filters. Since the lowpass filter is Gaussian and therefore isotropic, the resulting highpass filter will also be isotropic.

Continuing the 2-D notation of equ. (8.4), we can now express the intensities of the pixels in our highpass filtered image $I_h(x, y)$ as follows:

$$I_h(x, y) \;=\; \alpha I(x, y) - \beta I_{r,c}(x, y) \tag{8.11}$$

where $I_{r,c}(x, y)$ is the Gaussian lowpass filtered version of the input image $I(x, y)$.

We have chosen to give the user two parameters to control this highpass filter: the breadth b of the lowpass Gaussian filters g_x and g_y; and the low-frequency (DC) gain G_0 of the highpass filter. We then fix the high-frequency gain of the filter at $(1 + G_0)$. Hence:

$$\alpha = 1 + G_0 \quad \text{and} \quad \alpha - \beta = G_0 \;, \qquad \text{so} \quad \beta = 1 \tag{8.12}$$

If G_0 is near zero we find that the highpass filter removes most of the low-frequency components of I, so that I_h no longer has pixel intensities that approximately cover the range 0 to 255. Instead they tend to be centred around zero and this leads to the displayed image looking very dark. Ideally we want the pixel levels to remain centred approximately on 128. We achieve this by adding an offset term to equ. (8.11), which is proportional to $(1 - G_0)$ and equals 128 when $G_0 = 0$.

Therefore the new expression for I_h, including this offset and the results of equ. (8.12), becomes:

$$I_h(x, y) \;=\; (1 + G_0)I(x, y) - I_{r,c}(x, y) + 128(1 - G_0) \tag{8.13}$$

This is what we have implemented in our code.

Experimenting with the highpass filter on real images, shows that when we choose $G_0 = 1$, edges are enhanced, and further, that repeated use of the highpass filter can remove the worst effects of blurring with a lowpass filter.

If we select $G_0 = 0$ then we get an image that contains almost entirely edges (as long as b is quite small). This demonstrates the edge detection properties of a highpass filter.

Other interesting effects can be obtained with intermediate values for G_0 and b.

8.3 The script `ph_filter`

This script, shown in fig. 8.8, adopts the same general form as previous ones. In this case it comprises 7 cases selected by `mode`. The cases are:

Init initialises the **Filter** function. It opens a command figure (2) of the form shown at the top of fig. 8.8. Then it defines the 3 sliders and edit boxes for the 3 filter parameters, and their titles. It also defines the **Lowpass filter**, **Highpass filter**, **Revert** and **Close** buttons. Finally, it initialises the filter parameters and writes them to the sliders and edit boxes, and sets the **After** image equal to **Before**.

Slider is called when any of the 3 sliders are activated. It reads the slider values and converts them to filter parameters (b for the lowpass filter, b and G_0 for the highpass filter) using elements of the vector `sc(1:3)` as scaling factors. It also writes the parameters to the edit boxes.

Edit box is called when any of the 3 edit boxes are used to define any of the filter parameters. It reads the parameters and updates the sliders accordingly by dividing by `sc(1:3)`.

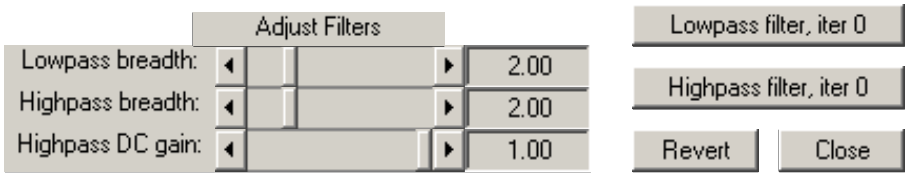
Lowpass is called when the **Lowpass filter** button is pressed. It updates the lowpass iteration counter and shows it on the button label. It then performs one iteration of lowpass filtering using `im_lowpass()`, and updates the output image.

Highpass is called when the **Highpass filter** button is pressed. It operates similarly to the lowpass button. The filtering is now implemented by equ. (8.13) and also calls `im_lowpass()`.

Revert is called when this button is pressed. It causes the output image to revert to being the input image and resets the iteration counters and their displayed values to zero.

Close is called when this button is pressed. It closes figure 2 and redisplays **Before** and **After** in figure 1 to show the input and filtered images respectively.

Command window figure(2):




```
% ph_filter.m
% Routine called by 'Filter' menu item in photo editor.
% Applies lowpass and higpass filters to an image.
%
% Nick Kingsbury, Cambridge University, 2006.

switch mode

    case 'Init'
        % Set up figure window for commands
        figure(2);
        set(gcf,'position',[400  624  450  100]);
        set(gcf,'numbertitle','off','name','Filter Image');

        % Initialise filter parameters for breadths and G0.
        sc = [10 10 1]; % Scale factors for breadths and G0.
        if exist('filpar')~=1, filpar = [2 2 1]'; end

        % Buttons for different filter parameters.
        label = {'Lowpass breadth:','Highpass breadth:','Highpass DC gain:'};
        for k = 1:3,
          ypos = 70 - 20*k; xpos = 10;
          slide(k) = uicontrol(gcf,'style','slider','sliderstep',[1 10]/100,...
            'pos',[xpos+100 ypos 120 20],'call','mode=''Slider''; ph_filter');
          uicontrol(gcf,'style','text','str',label{k},'pos',[xpos ypos 100 20]);
          edbox(k) = uicontrol(gcf,'style','edit','pos',[xpos+220 ypos 60 20],...
            'call','mode=''Edit box''; ph_filter');
        end
        uicontrol(gcf,'style','text','str','Adjust Filters','pos',[100 70 120 16]);

        % Other buttons: Denoise, Enhance, Revert and Close.
        lowpass = uicontrol(gcf,'pos',[310 70 130 20],...
            'call','mode=''Lowpass''; ph_filter');
        highpass = uicontrol(gcf,'pos',[310 40 130 20],...
            'call','mode=''Highpass''; ph_filter');
        revert = uicontrol(gcf,'str','Revert','pos',[310 10 60 20],...
            'call','mode=''Revert''; ph_filter');
        closebtn = uicontrol(gcf,'str','Close','pos',[380 10 60 20],...
            'call','mode=''Close''; ph_filter');
```

Fig. 8.8 continued overleaf.

```
        % Initialise variables and set yui = xui.
        y = double(xui);
        iter_Lo = 0;
        iter_Hi = 0;
        set(lowpass,'str',sprintf('Lowpass filter, iter %d',iter_Lo));
        set(highpass,'str',sprintf('Highpass filter, iter %d',iter_Hi));
        for k=1:3,
            set(slide(k),'value',filpar(k)/sc(k));
            set(edbox(k),'string',sprintf('%.2f',filpar(k)));
        end;
        yui = uint8(y + 0.5);
        update_y;
        figure(2);

    case 'Slider'
        for k=1:3,
            filpar(k) = sc(k)*get(slide(k),'value');
            set(edbox(k),'string',sprintf('%.2f',filpar(k)));
        end;

    case 'Edit box'
        for k=1:3,
            filpar(k) = sscanf(get(edbox(k),'string'),'%f');
            set(slide(k),'value',filpar(k)/sc(k));
            set(edbox(k),'string',sprintf('%.2f',filpar(k)));
        end;

    case 'Lowpass'
        iter_Lo = iter_Lo + 1;
        set(lowpass,'str',sprintf('Lowpass filter, iter %d',iter_Lo));
        y = im_lowpass(y,filpar(1));
        yui=uint8(y+0.5);
        update_y;
        figure(2);

    case 'Highpass'
        iter_Hi = iter_Hi + 1;
        set(highpass,'str',sprintf('Highpass filter, iter %d',iter_Hi));
        G0 = filpar(3);
        y = (1+G0)*y - im_lowpass(y,filpar(2)) + (1-G0)*128;
        yui=uint8(y+0.5);
        update_y;
        figure(2);
```

Fig. 8.8 continued overleaf.

```
case 'Revert'
    y = double(xui);
    yui=uint8(y);
    iter_Lo = 0;
    iter_Hi = 0;
    update_y;
    figure(2);
    set(lowpass,'str',sprintf('Lowpass filter, iter %d',iter_Lo));
    set(highpass,'str',sprintf('Highpass filter, iter %d',iter_Hi));

case 'Close'
    close(2);
    newbefore = 1;
    showimages;

end
```

Fig. 8.8: Script to filter an image: `ph_filter.m`

8.4 Fourier transform of a Gaussian pulse

Here we derive result that the Fourier transform of a Gaussian pulse in the time domain is a Gaussian-shaped function in the frequency domain.

Let the pulse be

$$g(t) = \frac{1}{\sqrt{2\pi}} \exp\left(\frac{-t^2}{2}\right) \tag{8.14}$$

Then its Fourier transform is

$$G(\omega) = \int_{-\infty}^{\infty} g(t) \exp(-j\omega t)\, dt = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \exp\left(\frac{-t^2}{2}\right) \exp(-j\omega t)\, dt \tag{8.15}$$

Now we do a trick known as 'completing the square of the exponent', so that we can do this integral. This relies on the relation:

$$\exp\left(\frac{-t^2}{2}\right) \exp(-j\omega t) = \exp\left(\frac{-(t^2 + 2j\omega t)}{2}\right) = \exp\left(\frac{-(t+j\omega)^2}{2}\right) \exp\left(\frac{-\omega^2}{2}\right) \tag{8.16}$$

We put this expression in the integral of (8.15) and make the substitution $\tau = t + j\omega$, to obtain:

$$\begin{aligned} G(\omega) &= \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \exp\left(\frac{-(t+j\omega)^2}{2}\right) \exp\left(\frac{-\omega^2}{2}\right) dt \\ &= \exp\left(\frac{-\omega^2}{2}\right) \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \exp\left(\frac{-\tau^2}{2}\right) d\tau \\ &= \exp\left(\frac{-\omega^2}{2}\right) G(0) \quad \text{(from equ. (8.15) with } \omega = 0) \end{aligned} \tag{8.17}$$

This gives us the *shape* of $G(\omega)$, but with an unknown constant scale factor $G(0)$. The integral to obtain $G(0)$ can be evaluated analytically, and it turns out that the result is $G(0) = 1$, because

$$\int_{-\infty}^{\infty} \exp\left(\frac{-\tau^2}{2}\right) d\tau = \sqrt{2\pi} \tag{8.18}$$

However it requires another trick to show this! The problem is that $\exp(-\tau^2/2)$ cannot be integrated analytically. The trick is to calculate the square of this definite integral and then to change the variables to $\tau = x$ and $\tau = y$ in the two separate integrals to obtain:

$$\begin{aligned} I^2 &= \left(\int_{-\infty}^{\infty} \exp\left(\frac{-\tau^2}{2}\right) d\tau\right)^2 \\ &= \int_{-\infty}^{\infty} \exp\left(\frac{-x^2}{2}\right) dx \quad \int_{-\infty}^{\infty} \exp\left(\frac{-y^2}{2}\right) dy \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} \exp\left(\frac{-(x^2+y^2)}{2}\right) dx\, dy \end{aligned} \tag{8.19}$$

This is now a 2-D integral over the whole infinite plane, which is still difficult in cartesian coordinates, but which can be done more easily if we change to polar coordinates. This is because the exponential is then only a function of r^2 and an extra factor of r appears in the integral that makes it 'do-able'. If we let $x = r\cos\theta$ and $y = r\sin\theta$, then the elemental area changes from $dx\,dy$ to $r\,d\theta\,dr$ and the double integral becomes:

$$\begin{aligned} I^2 &= \int_0^{\infty}\int_{-\pi}^{\pi} \exp\left(\frac{-r^2}{2}\right) r\,d\theta\,dr \;=\; \int_0^{\infty}\int_{-\pi}^{\pi} d\theta\; r\exp\left(\frac{-r^2}{2}\right) dr \\ &= 2\pi\left[-\exp\left(\frac{-r^2}{2}\right)\right]_0^{\infty} \;=\; 2\pi[0+1] \;=\; 2\pi \end{aligned} \tag{8.20}$$

Therefore

$$I = \int_{-\infty}^{\infty} \exp\left(\frac{-\tau^2}{2}\right) d\tau = \sqrt{2\pi} \tag{8.21}$$

and so, as required,

$$G(0) = \frac{I}{\sqrt{2\pi}} = 1 \tag{8.22}$$

Substituting this result into equ. (8.17) gives the Fourier transform relation we have been looking for:

$$g(t) = \frac{1}{\sqrt{2\pi}}\ \exp\left(\frac{-t^2}{2}\right) \qquad \rightleftharpoons \qquad G(\omega) = \exp\left(\frac{-\omega^2}{2}\right) \tag{8.23}$$

9 Enhancing the image: `ph_enhance`

The script `ph_enhance` extends the capabilities of the lowpass and highpass filters of section 8 so that they can reduce noise in smoother areas of the image and at the same time sharpen edges in other places.

Before looking at the code of `ph_enhance`, we shall look at how vertical and horizontal edges can be reliably detected, and how this edge information can spatially adapt the filtering that we apply to our image.

9.1 Detecting horizontal and vertical edges

Since, for efficiency reasons, we will be performing our adaptive filtering separately on the rows and columns of the image, it is desirable to detect horizontal edges separately from vertical ones. This will allow us, for example, to smooth in a direction parallel to an edge and to sharpen in a direction normal to the edge – thus achieving a measure of denoising and sharpening at the same time. Edges are detected by the function `im_edges(xui,thresh)`, where `xui` is the input image and `thresh` is a vector of two thresholds that determine whether an edge is significant for (a) denoising or (b) edge sharpening. This function is shown in fig. 9.1.

To detect horizontal (or near-horizontal) edges, we need to measure vertical gradients in the image. For simplicity we perform this on the luminance image `L1`, which is calculated first. In a sampled system like a digital image, this can be done by taking the difference between vertically adjacent pairs of pixels. However this tends to be rather sensitive to any noise in the image, so we first apply a small amount of Gaussian smoothing down the columns of the image before taking the differences. We use Gaussian filters with breadth $b = 0.8$ pel for this to get a good compromise between edge sensitivity and noise sensitivity. This is performed by the code

```
edv = diff(colfilter(L1,gaussian(0.8)));
```

To calculate the horizontal gradients we use similar code, but with `L1` transposed:

```
edh = diff(colfilter(L1.',gaussian(0.8))).';
```

The Matlab function `diff()` takes differences between adjacent pairs of elements down the columns of a matrix and reduces the height of the matrix by one row, since, if there are V elements in each column, there are only $V-1$ adjacent pairs. Hence for an input image of size $V \times U$ pels, the matrices `edv` and `edh` are of size $(V-1) \times U$ and $V \times (U-1)$ respectively.

In the loop that follows, two pairs of output matrices, `gainv` and `gainh`, are calculated from `edv` and `edh`. The lower slice of `gainv` represents the amount of lowpass filtering to be applied in the vertical direction when we do denoising, and likewise the lower slice of `gainh` represents the amount in the horizontal direction. The upper slices of `gainv` and `gainh` control the highpass filtering for sharpening in a similar (but opposite) way.

These gain matrices have elements between 0 and 1, calculated according to the rule:

$$gain_k = \frac{1}{1 + (ed/thresh_k)^2} \tag{9.1}$$

where ed is an edge gradient calculated above in `edv` or `edh`, and $thresh_k$ is the chosen threshold for denoising ($k = 1$) or sharpening ($k = 2$). This is a **soft thresholding** rule, which gives an output that tends to unity if the edge gradient is much smaller than the threshold, tends to zero if it is much larger, and equals 0.5 if they are the same. Hard thresholding would produce an abrupt change from unit gain to zero gain as the gradient becomes greater than the threshold, and would tend to introduce more noticeable artifacts into the processed image.

The above calculation is performed for a whole gain matrix by the lines of code

```
c = 1 / thresh(k).^2;
gainv(:,:,k) = ones(size(edv)) ./ (1 + c*edv.*edv);
```

and similar code for `gainh`.

Within this loop, we also show the gain matrices as images, such that black represents a gain of zero and white is a gain of unity. Figure(3) shows the two gain matrices for denoising, while figure(4) shows the two for sharpening. These images tend to show the detected edges as black ($gain_k = 0$) and the smoother regions as white ($gain_k = 1$), as shown for figure(3) in fig. 9.2. This is good for controlling the denoising filters, but for the sharpening filters we need to use the complement of these gains so that maximum sharpening is done around the edges and no sharpening is done in smooth regions. This is achieved by subtracting the relevant gain matrices from unity; and we shall see later that this is done when `im_enhance()` is called from within the **Sharpen** part of the code in the script `ph_enhance`.

```
function [gainh,gainv] = im_edges(xui,thresh)
% function [gainh,gainv] = im_edges(xui,thresh)
%
% Calculate gain matrices corresponding to the edges of image xui.
% gainh contains gains which depend on the horizontal gradients of xui
% and tends to zero where the gradients are << thresh and to one where
% the gradients are >> thresh.
% Similarly for gainv and vertical gradients.
% If thresh is a vector, then separate planes of gainh and gainv are
% created for each threshold element.
%
% Nick Kingsbury, Cambridge University, 2006.

% Calculate luminance image.
L1 = 0.3*double(xui(:,:,1)) + 0.6*double(xui(:,:,2)) + 0.1*double(xui(:,:,3));
```

Fig. 9.1 continued overleaf.

```
% Calculate edges.
edv = diff(colfilter(L1,gaussian(0.8)));
edh = diff(colfilter(L1.',gaussian(0.8))).';

% Calculate vertical and horizontal gain matrices for each threshold.
K = length(thresh);
gainv = zeros([size(edv) K]);
gainh = zeros([size(edh) K]);
for k = 1:K, % Loop for each threshold k (K should equal 2).

    % Calculate the gain matrices for threshold(k).
    c = 1 / thresh(k).^2;
    gainv(:,:,k) = ones(size(edv)) ./ (1 + c*edv.*edv);
    gainh(:,:,k) = ones(size(edh)) ./ (1 + c*edh.*edh);

    % Plot the gain matrices as greyscale images.
    figure(k+2);
    set(gcf,'position',[100  30+(2-k)*100   910   470]);
    set(gcf,'numbertitle','off','name','Edge gains');
    subplot('Position',[0.03 0.05 0.45 0.9]);
    image(gainv(:,:,k)*255 + 1); colormap(gray(256));
    axis image; axis ij;
    if k==1, title('Vertical Edge Gain - Denoising (white areas)',...
            'FontSize',14);
    else title('Vertical Edge Gain - Sharpening (black areas)',...
            'FontSize',14);
    end
    subplot('Position',[0.53 0.05 0.45 0.9]);
    image(gainh(:,:,k)*255 + 1); colormap(gray(256));
    axis image; axis ij;
    if k==1, title('Horizontal Edge Gain - Denoising (white areas)',...
            'FontSize',14);
    else title('Horizontal Edge Gain - Sharpening (black areas)',...
            'FontSize',14);
    end
    drawnow
end

figure(2);
return
```

Fig. 9.1: Function to detect horizontal and vertical edges of an image: `im_edges.m`

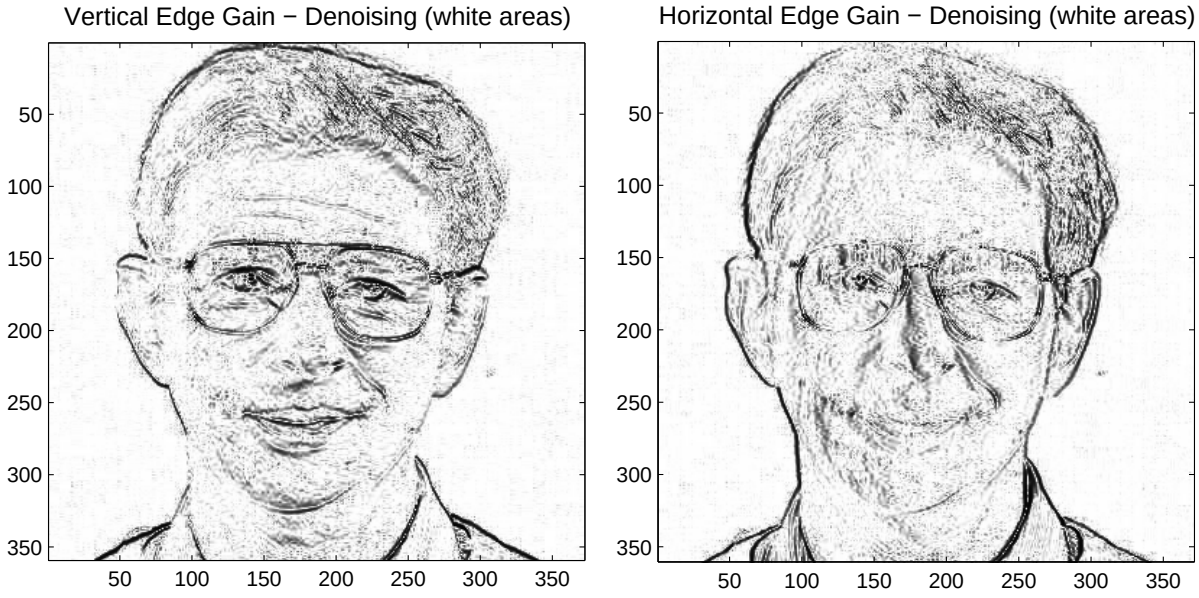


Fig. 9.2: Vertical and horizontal gain images for adaptive denoising.

9.2 Adaptive filters for denoising

To achieve adaptive filtering, we can no longer use the function `colfilter()` and the Matlab function `conv2()` which it calls. This is because `conv2()` assumes that the same filter is being used over the whole of the image. Hence we must go into more detail as to how filters for sampled signals work so we can write our own code to do the adaptive filtering.

A filter for continuous signals is defined by its impulse response $g(t)$ and the convolution integral as:

$$y(t) = \int_{-\infty}^{\infty} x(t-\tau)\ g(\tau)\ d\tau \tag{9.2}$$

In sampled (discrete) form the equivalent expression becomes:

$$y_n = \sum_{k=-M}^{M} x_{n-k}\ g_k \tag{9.3}$$

where x_n and y_n are the samples of the input and output signals at location (time) n, and the filter response g_n is assumed to be non-zero only between the finite limits $-M \le n \le M$. Hence this is known as a finite-impulse-response (FIR) discrete (or digital) filter with $2M+1$ samples (or taps). The filters we have been using on this course have all been of this form.

For example, the command `g = gaussian(1)` produces the following 7-sample filter with breadth $b = 1$:

$$\mathtt{g} = [g_{-3}\ \ldots\ g_3]^T = [0.0044\ \ 0.0540\ \ 0.2420\ \ 0.3991\ \ 0.2420\ \ 0.0540\ \ 0.0044]^T \tag{9.4}$$

For smaller values of b the filter gets shorter. With $b = 0.6$, the command `g = gaussian(0.6)` produces a filter with only 3 significant samples:

$$\mathtt{g} = [g_{-1}\ \ g_0\ \ g_1]^T = [0.1664\ \ 0.6672\ \ 0.1664]^T \tag{9.5}$$

The **Central Limit Theorem** of statistics shows that if many uncorrelated signals with similar pdfs (probability density functions) are added together, then the pdf of the result will be approximately a normal (Gaussian) distribution. In this situation the pdfs are all convolved together. Hence this also shows that the result of cascading many similar lowpass filters together will produce a filter whose impulse response is approximately Gaussian. (The only condition here is that the filter coefficients should all be non-negative – like a pdf.)

If the original impulse responses are Gaussian then the result is exact and the variance of the output Gaussian response will be the sum of the variances of the individual responses. The variance of a Gaussian response of breadth b is just b^2. Hence for N cascaded filters:

$$b_{out}^2 = N\ b_{in}^2 \qquad \text{or} \qquad b_{out} = \sqrt{N}\ b_{in} \tag{9.6}$$

Hence we may use several iterations of a short lowpass filter to be equivalent to a longer filter with approximately Gaussian shape. We will now concentrate on making a short (3-sample) filter with an adaptive breadth, that we can then use iteratively on our image to reach any desired strength of adaptive smoothing.

We can make a 3-sample lowpass filter with varying *strength* with respect to samples on each side of the mid point using the following impulse response:

$$\mathtt{g} = [\alpha_1 \quad (1 - \alpha_1 - \alpha_2) \quad \alpha_2]^T \tag{9.7}$$

where α_1 controls the smoothing applied to one side of the mid-point, while α_2 controls smoothing to the other side. The central term ensures that the sum of the 3 terms is unity and hence that the low-frequency gain of the filter is unity for any choice of α_1, α_2.

The maximum useful level of smoothing with this filter occurs when $\alpha_1 = \alpha_2 = \frac{1}{4}$ and `g` becomes $[\frac{1}{4} \quad \frac{1}{2} \quad \frac{1}{4}]^T$. Selecting lower values of α than this produces less smoothing of the signal. Hence we can make an edge-adaptive filter by using `gainh` to control the value of α_1 and α_2 on either side of every pixel when we filter across the rows, and similarly use `gainv` to control α_1 and α_2 above and below every pixel when we filter down the columns.

The discrete convolution of the above `g` with an input signal x_n is given by:

$$\begin{aligned} y_n &= \sum_{k=-1}^{1} x_{n-k}\ g_k \\ &= \alpha_1\ x_{n+1} + (1 - \alpha_1 - \alpha_2)x_n + \alpha_2\ x_{n-1} \\ &= \alpha_1(x_{n+1} - x_n) + x_n - \alpha_2(x_n - x_{n-1}) \end{aligned} \tag{9.8}$$

By grouping the α_1 and α_2 terms together we get a simpler implementation as $(x_{n+1} - x_n)$ for one sample at n is the same as $(x_n - x_{n-1})$ for the next sample at $n + 1$ and hence only needs to be calculated once, as shown below.

Fig. 9.3 lists the function `im_enhance()` which implements the above adaptive filtering. To denoise the image, it is called from within the **Denoise** case of the script `ph_enhance` by the following command:

```
cg = thresh(3)/400;
y = im_enhance(y,cg*gainh(:,:,1),cg*gainv(:,:,1));
```

This applies one iteration of the adaptive filter to the real version of the image `y`. The second and third arguments become `alphah` and `alphav` in `im_enhance()` and are directly proportional to the matrices `gainh` and `gainv`, which were calculated by `im_edges()` using the denoising threshold. The constant of proportionality `cg` is set by the user with the **Strength** slider to be between 0 and 0.25. When 100% strength is selected (`thresh(3)=100`), the α values are 0.25 in smooth areas of the image and reduce to zero near edges.

Within `im_enhance()`, the filtering is performed inside the for-loop for each colour slice. First the input slice is copied to `xk`. Then the vertical filtering, as in equ. (9.8), is applied to the columns of the whole `xk` matrix by

```
xd = diff(xk) .* alphav;
xk = xk - [zr; xd] + [xd; zr]; % Apply vertical filtering
```

where `xd` is a matrix of pixel differences of the form $(x_{n+1} - x_n)$, scaled by the vertical α values in `alphav`. The second line implements equ. (9.8) as follows: `xk` represents the x_n and y_n terms; `[zr; xd]` represents the $\alpha_2(x_n - x_{n-1})$ term; and `[xd; zr]` represents the $\alpha_1(x_{n+1} - x_n)$ term. The row of zeros, `zr`, is used to pad out the $(V-1) \times U$ matrix `xd` to size $V \times U$, either at the top or at the bottom, depending on whether we are generating $\alpha_2(x_n - x_{n-1})$ or $\alpha_1(x_{n+1} - x_n)$ for each pixel. We are again assuming symmetric extension (mirroring) of the image at its boundaries, so the pixel differences across the boundary are zero.

Adaptive filtering across the rows of the image is achieved by the next two lines of code in the for-loop

```
xd = diff(xk.').' .* alphah;
y(:,:,k) = xk - [zc  xd] + [xd  zc]; % Apply horizontal filtering
```

which operates in a similar way, except that the differences are calculated on the transpose of `xk`, and `xd` is now padded with a column of zeros `zc` at either its left or right end. The result is placed in colour slice `k` of `y`.

Thus we see how a single iteration of the adaptive filter is performed. If stronger filtering is needed to achieve greater denoising, then we simply pass the resulting image back through the filter one or more additional times. The edge adaptation ensures that smoothing is not applied across detected edges, and so very little blurring is introduced by this denoising process (unlike the non-adaptive lowpass filter of section 8).

```
function y = im_enhance(x,alphah,alphav)
% function y = im_enhance(x,alphah,alphav)
%
% Enhance the image x using lowpass filtering with edge inhibition.
% alphah and alphav specify the horizontal and vertical coefficients
% for the enhancement process.
% If alpha is positive (and <= 0.25) then the image is denoised by
% lowpass filtering in proportion to the alpha at each location.
% If alpha is negative (and >= -0.25) then the image edges are enhanced
% by highpass filtering in proportion to -alpha at each location.
% The magnitude of the alphas should not exceed 0.25 for proper operation
% of the filters.
%
% Nick Kingsbury, Cambridge University, 2006.

sx = size(x);
y = x;

if sum(alphav(:)) >0, wh = waitbar(0,'Denoising image');
else wh = waitbar(0,'Sharpening image');
end

% Zero row and column for padding out and offsetting the difference images
% at the image edges.
zr = zeros(1,sx(2));
zc = zeros(sx(1),1);

% Apply adaptive filtering, controlled by alphav and alphah, to each colour
% layer.
for k = 1:3,
    waitbar(k/4);
    xk = x(:,:,k);
    xd = diff(xk) .* alphav;
    xk = xk - [zr; xd] + [xd; zr]; % Apply vertical filtering
    xd = diff(xk.').' .* alphah;
    y(:,:,k) = xk - [zc  xd] + [xd  zc]; % Apply horizontal filtering
end

close(wh) % Close wait bar
return
```

Fig. 9.3: Function to perform adaptive denoising and sharpening of an image: `im_enhance()`

9.3 Adaptive filters for edge sharpening

As we saw in section 8.2, a highpass filter, which amplifies the higher frequency components in a signal and sharpens edges, can be created from a lowpass filter by simple weighted subtraction. If we apply this idea to equ. (9.8) with weights of 2 and 1 so as to keep unit gain at low frequencies, then we get a highpass adaptive filter:

$$\begin{aligned} y_n &= 2x_n - \sum_{k=-1}^{1} x_{n-k}\ g_k \\ &= -\alpha_1\ x_{n+1} + (2 - 1 + \alpha_1 + \alpha_2)x_n - \alpha_2\ x_{n-1} \\ &= -\alpha_1(x_{n+1} - x_n) + x_n + \alpha_2(x_n - x_{n-1}) \end{aligned} \tag{9.9}$$

This is identical to the result of equ. (9.8), except that α_1 and α_2 have been negated.

Hence we can still use our function `im_enhance()`, as long as we pass it *negative* values in its arguments `alphah` and `alphav`. Since we want to apply edge enhancement only in areas of the image that are close to detected edges, we also have to complement the soft gain values from `im_edges()`, as discussed in section 9.1. Both of these tasks are accomplished in the following two lines of code, from the **Sharpen** case of `ph_enhance`:

```
cg = thresh(3)/400;
y = im_enhance(y,cg*(gainh(:,:,2)-1),cg*(gainv(:,:,2)-1));
```

By subtracting 1 from `gainh` and `gainv`, we make them negative and complement the gain law at the same time.

This adaptive sharpening filter can be applied iteratively to increase the amount of sharpening to blurred images, just as we did with the denoising filter to reduce the noise further. However we have to be careful to avoid introducing excessive high frequency gain by many iterations which can produce undesirable artifacts at some edges in the output image.

The key here is that the adaptive filter only applies high frequency enhancement in the close vicinity of detected edges, and only in a direction that is approximately normal to each edge. This avoids amplifying noise in other areas of the image.

9.4 The script `ph_enhance`

This script, shown in fig. 9.4, adopts the same general form as previous ones and is very similar to the previous script `ph_filter`. It again comprises 7 cases selected by `mode`. The cases are:

Init initialises the **Enhance** function. It is virtually identical to the same case in `ph_filter`, except that the Lowpass and Highpass buttons are replaced by **Reduce noise** and **Sharpen edges** buttons.

Slider is called when any of the 3 sliders are activated. It is the same as in `ph_filter`, except that it also calls the function `im_edges()` to generate the four gain matrices `gainh(:,:,1:2)` and `gainv(:,:,1:2)`, as described in section 9.1.

Edit box is called when any of the 3 edit boxes are used to define any of the filter parameters. It is the same as in `ph_filter`, except that again it also calls the function `im_edges()`.

Denoise is called when the **Reduce noise** button is pressed. It updates the denoising iteration counter and shows it on the button label. It then performs one iteration of adaptive denoising using `im_enhance()` with positive values for the filter coefficients (the gain matrices, scaled to the range 0 ... 0.25 by the **Strength** percentage in `thresh(3)`) and updates the output image.

Sharpen is called when the **Sharpen edges** button is pressed. It operates similarly to the Denoise case, except that 1 is subtracted from the gain matrices, so that the filter coefficients are now scaled to the range -0.25 ... 0 and the adaptive filters perform highpass filtering close to the detected edges in the image, instead of lowpass filtering in the smoother regions, as discussed in section 9.3.

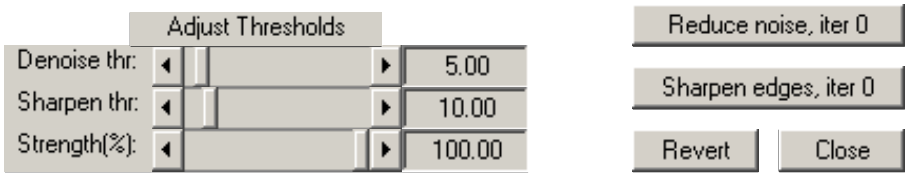
Revert is called when this button is pressed. It causes the output image to revert to being the input image and resets the iteration counters and their displayed values to zero.

Close is called when this button is pressed. It closes figure 2 and redisplays **Before** and **After** in figure 1 to show the input and filtered images respectively.

This completes the description of our Photo Editor. Feel free to use it and extend as you wish. If you develop any good new functions or scripts, do please let me know!

Nick Kingsbury.
April 9, 2016

Command window figure(2):




```
% ph_enhance.m
% Routine called by 'Enhance' menu item in photo editor.
%
% Nick Kingsbury, Cambridge University, 2006.

switch mode

    case 'Init'
        % Set up figure window for commands
        figure(2);
        set(gcf,'position',[400  624  450  100]);
        set(gcf,'numbertitle','off','name','Enhance Image');

        % Initialise denoising and sharpening edge detection thresholds.
        sc = 100;
        if exist('thresh') ~= 1, thresh = sc*[0.05 0.1 1.0]'; end

        % Buttons for different thresholds and filter strength.
        label = {'Denoise thr:','Sharpen thr:','Strength(%):'};
        for k = 1:3,
          ypos = 70 - 20*k; xpos = 10;
          slide(k) = uicontrol(gcf,'style','slider','sliderstep',[1 5]/100,...
            'pos',[xpos+70 ypos 120 20],'call','mode=''Slider''; ph_enhance');
          uicontrol(gcf,'style','text','str',label{k},'pos',[xpos ypos 70 20]);
          edbox(k) = uicontrol(gcf,'style','edit','pos',[xpos+190 ypos 60 20],...
            'call','mode=''Edit box''; ph_enhance');
        end
        uicontrol(gcf,'style','text','str','Adjust Thresholds',...
            'pos',[70 70 120 16]);

        % Other buttons: Denoise, Enhance, Revert and Close.
        denoise = uicontrol(gcf,'pos',[310 70 130 20],...
            'call','mode=''Denoise''; ph_enhance');
        sharpen = uicontrol(gcf,'pos',[310 40 130 20],...
            'call','mode=''Sharpen''; ph_enhance');
        revert = uicontrol(gcf,'str','Revert','pos',[310 10 60 20],...
            'call','mode=''Revert''; ph_enhance');
        closebtn = uicontrol(gcf,'str','Close','pos',[380 10 60 20],...
            'call','mode=''Close''; ph_enhance');
```

Fig. 9.4 continued overleaf.

```
        % Initialise variables and set yui = xui.
        y = double(xui) + 0.5;
        iter_n = 0;
        iter_s = 0;
        set(denoise,'str',sprintf('Reduce noise, iter %d',iter_n));
        set(sharpen,'str',sprintf('Sharpen edges, iter %d',iter_s));
        for k=1:3,
            set(slide(k),'value',thresh(k)/sc);
            set(edbox(k),'string',sprintf('%.2f',thresh(k)));
        end;
        yui=uint8(y);
        % Calculate edge maps for both thresholds.
        [gainh,gainv] = im_edges(xui,thresh(1:2));
        update_y;
        figure(2);

    case 'Slider'
        for k=1:3,
            thresh(k) = sc*get(slide(k),'value');
            set(edbox(k),'string',sprintf('%.2f',thresh(k)));
        end;
        [gainh,gainv] = im_edges(xui,thresh(1:2));

    case 'Edit box'
        for k=1:3,
            thresh(k) = sscanf(get(edbox(k),'string'),'%f');
            set(slide(k),'value',thresh(k)/sc);
            set(edbox(k),'string',sprintf('%.2f',thresh(k)));
        end;
        [gainh,gainv] = im_edges(xui,thresh(1:2));

    case 'Denoise'
        iter_n = iter_n + 1;
        set(denoise,'str',sprintf('Reduce noise, iter %d',iter_n));
        cg = thresh(3)/400;
        y = im_enhance(y,cg*gainh(:,:,1),cg*gainv(:,:,1));
        yui=uint8(y);
        update_y;
        figure(2);

    case 'Sharpen'
        iter_s = iter_s + 1;
        set(sharpen,'str',sprintf('Sharpen edges, iter %d',iter_s));
        cg = thresh(3)/400;
        y = im_enhance(y,cg*(gainh(:,:,2)-1),cg*(gainv(:,:,2)-1));
        yui=uint8(y);
        update_y;
        figure(2);
```

Fig. 9.4 continued overleaf.

```
case 'Revert'
    y = double(xui)+0.5;
    yui=uint8(y);
    iter_n = 0;
    iter_s = 0;
    update_y;
    figure(2);
    set(denoise,'str',sprintf('Reduce noise, iter %d',iter_n));
    set(sharpen,'str',sprintf('Sharpen edges, iter %d',iter_s));

case 'Close'
    close(2);
    newbefore = 1;
    showimages;

end
```

Fig. 9.4: Script to enhance an image: `ph_enhance.m`