

## Problem 1

There is a GUI through which the user can interact with the program to select images, Region of interest and the locations for blending.

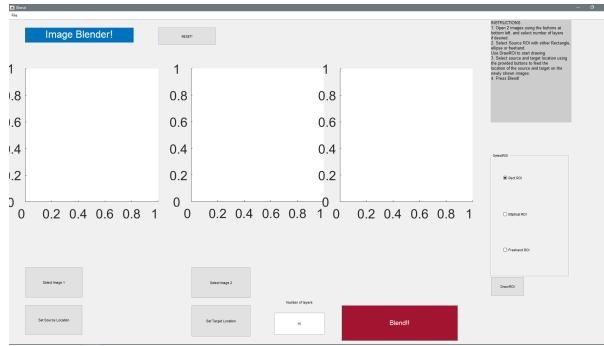


Figure 1: Example 1

```

1 function varargout = Blend(varargin)
2 % BLEND MATLAB code for Blend.fig
3 %     BLEND, by itself, creates a new BLEND or raises the existing
4 %     singleton*.
5 %
6 %     H = BLEND returns the handle to a new BLEND or the handle to
7 %     the existing singleton*.
8 %
9 %     BLEND('CALLBACK', hObject, eventData, handles,...) calls the local
10 %     function named CALLBACK in BLEND.M with the given input arguments.
11 %
12 %     BLEND('Property','Value',...) creates a new BLEND or raises the
13 %     existing singleton*. Starting from the left, property value pairs are
14 %     applied to the GUI before Blend_OpeningFcn gets called. An
15 %     unrecognized property name or invalid value makes property application
16 %     stop. All inputs are passed to Blend_OpeningFcn via varargin.
17 %
18 %     *See GUI Options on GUIDE's Tools menu. Choose "GUI allows only one
19 %     instance to run (singleton)".
20 %
21 % See also: GUIDE, GUIDATA, GUIHANDLES
22
23 % Edit the above text to modify the response to help Blend
24
25 % Last Modified by GUIDE v2.5 13-Nov-2019 11:25:45
26
27 % Begin initialization code - DO NOT EDIT
28 gui_Singleton = 1;
29 gui_State = struct('gui_Name',         mfilename, ...
30                     'gui_Singleton',    gui_Singleton, ...
31                     'gui_OpeningFcn',   @Blend_OpeningFcn, ...
32                     'gui_OutputFcn',    @Blend_OutputFcn, ...
33                     'gui_LayoutFcn',    [] , ...
34                     'gui_Callback',     []);
35 if nargin && ischar(varargin{1})
36     gui_State.gui_Callback = str2func(varargin{1});
37 end
38
39 if nargout
40     [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});

```

```
41 else
42     gui_mainfcn(gui_State, varargin{:});
43 end
44 % End initialization code - DO NOT EDIT
45
46 % --- Executes just before Blend is made visible.
47 function Blend_OpeningFcn(hObject, eventdata, handles, varargin)
48 % This function has no output args, see OutputFcn
49 % hObject    handle to figure
50 % eventdata   reserved - to be defined in a future version of MATLAB
51 % handles    structure with handles and user data (see GUIDATA)
52 % varargin   command line arguments to Blend (see VARARGIN)
53
54 % Choose default command line output for Blend
55 handles.output = hObject;
56
57 % Update handles structure
58 guidata(hObject, handles);
59
60 % This sets up the initial plot - only do when we are invisible
61 % so window can get raised using Blend.
62 if strcmp(get(hObject,'Visible'),'off')
63     plot(rand(5));
64 end
65
66 % UIWAIT makes Blend wait for user response (see UIRESUME)
67 % uiwait(handles.figure1);
68
69
70 % --- Outputs from this function are returned to the command line.
71 function varargout = Blend_OutputFcn(hObject, eventdata, handles)
72 % varargout cell array for returning output args (see VARARGOUT);
73 % hObject    handle to figure
74 % eventdata   reserved - to be defined in a future version of MATLAB
75 % handles    structure with handles and user data (see GUIDATA)
76
77 % Get default command line output from handles structure
78 varargout{1} = handles.output;
79
80 % -----
81 function OpenMenuItem_Callback(hObject, eventdata, handles)
82 % hObject    handle to OpenMenuItem (see GCBO)
83 % eventdata   reserved - to be defined in a future version of MATLAB
84 % handles    structure with handles and user data (see GUIDATA)
85 file = uigetfile('*.fig');
86 if ~isequal(file, 0)
87     open(file);
88 end
89 % --- Executes on button press in Img1.
90 % --- read images selected by the user
91 function Img1_Callback(hObject, eventdata, handles)
92 % hObject    handle to Img1 (see GCBO)
93 % eventdata   reserved - to be defined in a future version of MATLAB
94 % handles    structure with handles and user data (see GUIDATA)
95 [img1, path1, ~] = uigetfile('*.jpg; *.png; *.jpeg', 'Pick a source image');
96 if ~isequal(img1, 0)
97     fg = im2double(imread(horzcat(path1,img1)));
98 end
99 axes(handles.Source);
100 cla;
101 fg1 = fg;
102 hObject.UserData = fg;
103 imshow(fg)
104
105 % --- Executes on button press in Img2.
```

```
106 % --- read images selected by the user
107 function Img2_Callback(hObject, eventdata, handles)
108 % hObject    handle to Img2 (see GCBO)
109 % eventdata   reserved - to be defined in a future version of MATLAB
110 % handles    structure with handles and user data (see GUIDATA)
111 [img2, path2, ~] = uigetfile('*.jpg; *.png; *.jpeg', 'Pick a source image');
112 if ~isequal(img2, 0)
113     bg = im2double(imread(horzcat(path2,img2)));
114 end
115 axes(handles.Target);
116 cla;
117 imshow(bg)
118 hObject.UserData = bg;
119
120 % --- Executes during object creation, after setting all properties.
121 function Img2_CreateFcn(hObject, eventdata, handles)
122 % hObject    handle to Img2 (see GCBO)
123 % eventdata   reserved - to be defined in a future version of MATLAB
124 % handles    empty - handles not created until after all CreateFcns called
125
126
127 % --- Executes on button press in Rect.
128 % --- select rectangular ROI
129 function Rect_Callback(hObject, eventdata, handles)
130 % hObject    handle to Rect (see GCBO)
131 % eventdata   reserved - to be defined in a future version of MATLAB
132 % handles    structure with handles and user data (see GUIDATA)
133 el = findobj('Tag', 'Ellipse');
134 fr = findobj('Tag', 'FreeHnd');
135 crt = findobj('Tag', 'DrawROI');
136 axes(handles.Source);
137 try
138     roi = crt.UserData{3};
139     delete(roi);
140 catch
141 end
142 el.Value=0;
143 fr.Value=0;
144
145
146 % --- Executes on button press in Ellipse.
147 % --- select elliptical ROI
148 function Ellipse_Callback(hObject, eventdata, handles)
149 % hObject    handle to Ellipse (see GCBO)
150 % eventdata   reserved - to be defined in a future version of MATLAB
151 % handles    structure with handles and user data (see GUIDATA)
152 re = findobj('Tag', 'Rect');
153 fr = findobj('Tag', 'FreeHnd');
154 crt = findobj('Tag', 'DrawROI');
155 axes(handles.Source);
156 try
157     roi = crt.UserData{3};
158     delete(roi);
159 catch
160 end
161 re.Value=0;
162 fr.Value=0;
163
164 % --- Executes on button press in FreeHnd.
165 % --- select freehand ROI
166 function FreeHnd_Callback(hObject, eventdata, handles)
167 % hObject    handle to FreeHnd (see GCBO)
168 % eventdata   reserved - to be defined in a future version of MATLAB
169 % handles    structure with handles and user data (see GUIDATA)
170 re = findobj('Tag', 'Rect');
```

```

171 el = findobj('Tag', 'Ellipse');
172 crt = findobj('Tag', 'DrawROI');
173 axes(handles.Source);
174 try
175     roi = crt.UserData{3};
176     delete(roi);
177 catch
178 end
179 re.Value=0;
180 el.Value=0;
181
182 % --- Executes on button press in Blnd.
183 % --- Blend the selected images based on selected ROI
184 function Blnd_Callback(hObject, eventdata, handles)
185 % hObject    handle to Blnd (see GCBO)
186 % eventdata reserved - to be defined in a future version of MATLAB
187 % handles    structure with handles and user data (see GUIDATA)
188 h1 = findobj('Tag', 'Img1');
189 h2 = findobj('Tag', 'Img2');
190 nl = findobj('Tag', 'nlayers');
191 msk = findobj('Tag', 'DrawROI');
192 trg = findobj('Tag', 'SetTrgtLocn');
193 bg = h2.UserData;
194 try
195     fg1 = trg.UserData{1};
196     bw_mask = trg.UserData{2};
197 catch
198     fg1 = msk.UserData{1};
199     bw_mask = msk.UserData{2};
200 end
201 %handle RGB images.
202 if size(bg,3)>1
203     blendedImg(:,:,1) = blendPyramid(fg1(:,:,1),bg(:,:,1),bw_mask,str2double(nl.String));
204     blendedImg(:,:,2) = blendPyramid(fg1(:,:,2),bg(:,:,2),bw_mask,str2double(nl.String));
205     blendedImg(:,:,3) = blendPyramid(fg1(:,:,3),bg(:,:,3),bw_mask,str2double(nl.String));
206 else
207     blendedImg(:,:,1) = blendPyramid(fg1(:,:,1),bg(:,:,1),bw_mask,str2double(nl.String));
208 end
209 %
210 blendedImg = ScaleRGBValues(blendedImg);
211 axes(handles.Source);
212 imshow(h1.UserData);
213 axes(handles.Target);
214 imshow(bg);
215 axes(handles.Outputs);
216 imshow(blendedImg);
217 imwrite(blendedImg, 'blend.png');
218
219 % --- Executes on button press in SetTrgtLocn.
220 function SetTrgtLocn_Callback(hObject, eventdata, handles)
221 % hObject    handle to SetTrgtLocn (see GCBO)
222 % eventdata reserved - to be defined in a future version of MATLAB
223 % handles    structure with handles and user data (see GUIDATA)
224 droi = findobj('Tag', 'DrawROI');
225 fgp = droi.UserData{1};
226 bwm = droi.UserData{2};
227 src = findobj('Tag', 'SetSrcLocn');
228 axes(handles.Target);
229 trgtLoc = drawrectangle();
230 trgtPos = trgtLoc.Position;
231 try
232     srcPos = src.UserData;
233     transVec = trgtPos(1:2)-srcPos(1:2);
234 catch
235     transVec = [0,0];
236 end

```

```
236 %translate image to location selected by the user.
237 fgp=imtranslate(fgp, transVec);
238 bwm = imtranslate(bwm, transVec);
239 axes(handles.Source);
240 imshow(fgp);
241 axes(handles.Outputs);
242 imshow(bwm);
243 hObject.UserData{1} = fgp;
244 hObject.UserData{2} = bwm;
245 imwrite(bwm, 'mask.png');
246 imwrite(fgp, 'fgPadded.png');
247 delete(trgtLoc);
248
249 % --- Executes on button press in SetSrcLocn.
250 function SetSrcLocn_Callback(hObject, eventdata, handles)
251 % hObject handle to SetSrcLocn (see GCBO)
252 % eventdata reserved - to be defined in a future version of MATLAB
253 % handles structure with handles and user data (see GUIDATA)
254 axes(handles.Source);
255 srcLoc = drawrectangle();
256 srcPos = srcLoc.Position;
257 hObject.UserData = srcPos;
258 delete(srcLoc);
259
260 function nlayers_Callback(hObject, eventdata, handles)
261 % hObject handle to nlayers (see GCBO)
262 % eventdata reserved - to be defined in a future version of MATLAB
263 % handles structure with handles and user data (see GUIDATA)
264
265 % Hints: get(hObject,'String') returns contents of nlayers as text
266 % str2double(get(hObject,'String')) returns contents of nlayers as a double
267
268 % --- Executes during object creation, after setting all properties.
269 function nlayers_CreateFcn(hObject, eventdata, handles)
270 % hObject handle to nlayers (see GCBO)
271 % eventdata reserved - to be defined in a future version of MATLAB
272 % handles empty - handles not created until after all CreateFcns called
273
274 % Hint: edit controls usually have a white background on Windows.
275 % See ISPC and COMPUTER.
276 if ispc && isequal(get(hObject, 'BackgroundColor'), get(0, 'defaultUicontrolBackgroundColor'))
277 set(hObject, 'BackgroundColor', 'white');
278 end
279
280
281 % --- Executes on button press in DrawROI.
282 % --- draw ROI based on selected region type
283
284 function DrawROI_Callback(hObject, eventdata, handles)
285 % hObject handle to DrawROI (see GCBO)
286 % eventdata reserved - to be defined in a future version of MATLAB
287 % handles structure with handles and user data (see GUIDATA)
288 h1 = findobj('Tag', 'Img1');
289 h2 = findobj('Tag', 'Img2');
290 re = findobj('Tag', 'Rect');
291 el = findobj('Tag', 'Ellipse');
292 fr = findobj('Tag', 'FreeHnd');
293 src = findobj('Tag', 'SetSrcLocn');
294 trg = findobj('Tag', 'SetTrgtLocn');
295 fg1 = trg.UserData;
296 if isempty(fg1)
297   fg1 = h1.UserData;
298 end
299 bg = h2.UserData;
300 axes(handles.Source);
```

```

301     if re.Value == 1 && el.Value == 0 && fr.Value == 0
302         [fg1, roi, bw_mask] = createROIIMask(fg1, bg, 1);
303     elseif re.Value == 0 && el.Value == 1 && fr.Value == 0
304         [fg1, roi, bw_mask] = createROIIMask(fg1, bg, 2);
305     else
306         [fg1, roi, bw_mask] = createROIIMask(fg1, bg, 3);
307     end
308     axes(handles.Outputs);
309     imshow(bw_mask);
310     axes(handles.Source);
311     imshow(fg1);
312     imwrite(bw_mask, 'mask.png');
313     imwrite(fg1, 'fgPadded.png');
314     hObject.UserData{1} = fg1;
315    (hObject).UserData{2} = bw_mask;
316    (hObject).UserData{3} = roi;
317
318
319 % --- Executes on button press in Reset.
320 function Reset_Callback(hObject, eventdata, handles)
321 % hObject      handle to Reset (see GCBO)
322 % eventdata    reserved - to be defined in a future version of MATLAB
323 % handles      structure with handles and user data (see GUIDATA)
324     close(Blend);
325     run('Blend');
```

Code 1: GUI.m - created with GUIDE

a) Calculation of Gaussian and Laplacian Pyramid.

The Laplacian and Gaussian Pyramid computation is done as follows

**Result:** Gaussian and Laplacian Pyramid

Set base layer of gaussian pyramid as original;

```

while layer less than numLayers and layerSize greater than 1 do
    | gpyr(next) = gpyr(prev)-smooth(downscaled(gpyr(prev)));
end
lpyr(top) = gpyr(top);
while layer greater than 1 do
    | lpyr(prev) = gpyr(prev)-smooth(upscale(gpyr(curr)))
end
```

**Algorithm 1:** Gaussian/ Laplacian Pyramid calculation

The Gaussian Kernel is calculated using the fspecial function from Matlab. The function accepts sigma(variance) and k(kernel size) as arguments, which are calculated as follows.

$$\sigma = \frac{(2 * \text{downscale})}{6}$$

$$k = \text{int}(\text{truncate} * \sigma + 0.5)$$

Truncate taken to be 4.0 by default.

We can tune the hyper-parameter downscale to get desired results. This can be taken as a heuristic approach to finding the optimum window size.

As the transition while blending depends on the variance and kernel size of the Gaussian filter. It is important to tune these parameters to get a smooth transition.

The Gaussian pyramid of the mask is taken to ensure a smooth transition between the 2 images.

```

1 function [gPyr,lPyr] = ComputePyr(img1, num_layers)
2 %GENGAUSSIANPYR Calculate Gaussian and Laplacian Pyramids of the image.
3 % Calculate the Gaussian and Laplacian Pyramid based on the number of
4 % layers provided by the user/ till max possible layers (last layer being
5 % 1x1)
6
7 layer = 1;
8 %generate gaussian kernel
9 dscale = 2.5;
10 sigm = (2*dscale)./6;
11 kernS = floor(4*sigm+0.5);
12 h = fspecial('gaussian', kernS ,sigm);
13 % h = ([1;4;6;4;1]*[1 4 6 4 1])./256; %can be implemented as a
14 % separable filter as shown
15
16 %set the initial layer as the original image.
17 gPyr{layer} = img1;
18 %while size of pyramid is greater than 1x1 & greater than layers
19 %specified.
20
21 %can view the pyramids by uncommenting the imshow and figure functions.
22 %Takes up a lot of time.
23 while(sum(size(gPyr{layer}))^=2 && layer<=num_layers)
24     %smoothen and downscale
25     gPyr{layer+1} = conv2d(gPyr{layer}, h, 3);
26     gPyr{layer+1} = downscale(gPyr{layer+1},0.5);
27     layer = layer+1;
28 %
29 %     figure;
30 %     imshow(gPyr{layer});
31 end
32 lPyr{layer}=gPyr{layer};
33 while layer>1
34     lPyr{layer-1} = gPyr{layer-1}-conv2d(upscale(gPyr{layer},size(gPyr{layer-1})), h, 3);
35 %
36 %     figure;
37 %     imshow(lPyr{layer});
38     layer = layer-1;
39 end
40
41 end

```

Code 2: ComputePyr.m - Compute gaussian and laplacian pyramids

### Helper functions

Nearest Neighbor interpolation performed to upscale and downscale images through the pyramids. The upscale function uses newSize as argument to avoid dimension mismatch while blending the images. (Happens Due to integer rounding)

```

1 function [ResampledImage] = upscale(InputImage,newSize)
2 %UPSCALE used to upscale image to selected size
3 % newSize taken as input to avoid mismatch of dimensions while moving
4 % through the pyramid.
5 oldSize = size(InputImage);
6 scale = newSize./oldSize;
7
8 % nearest neighbor interpolation.
9 % calculate the new row and column indices for assignment of pixels
10 %subtract -0.5 to consider center of pixel. Add 0.5 to negate subtraction
11 %min function used to avoid any overflows
12 rowIndices = min(round(((1:newSize(1))-0.5)./scale(1)+0.5), oldSize(1));
13 colIndices = min(round(((1:newSize(2))-0.5)./scale(2)+0.5), oldSize(2));
14 %assignment based on the newly calculated indices
15 ResampledImage = InputImage(rowIndices, colIndices,:);
16 end

```

Code 3: Upscale

---

```

1 function [ResampledImage] = downscale(InputImage, ScalingFactor)
2 %IMRESAMPLE Resample image according to the given input
3 %   Resize the given image according to the given scaling factor, implement
4 %   nearest neighbor interpolation while upsampling to sample the images.
5
6 scale = [ScalingFactor, ScalingFactor];
7 oldSize = size(InputImage);
8 newSize = max(floor(scale.*oldSize(1:2)),1); %to avoid 0 index error
9 % nearest neighbor interpolation.
10 % calculate the new row and column indices for assignment of pixels
11 rowIndices = min(round(((1:newSize(1))-0.5)./scale(1)+0.5), oldSize(1));
12 colIndices = min(round(((1:newSize(2))-0.5)./scale(2)+0.5), oldSize(2));
13 ResampledImage = InputImage(rowIndices, colIndices,:);
14 end

```

---

Code 4: Downscale

Convolution and Padding functions from project 2 used. The convolution can be performed as a separable 1D convolution using 2 loops instead of nested loops which can improve the performance by a tad bit. In this case the convolution is with 3x3 gaussian filters generated with the fspecial command, hence the same function can be used to perform the convolution.

Moreover, a faster approach can be moving to frequency domain, and multiplying and again moving back to spatial domain.

---

```

1 function [op_img] = conv2d(img,kern, pad)
2 %CONV2 perform 2d convolution of an image with given kernel
3 %   Perform 2d convolution of an image.
4 [r,c] = size(img);
5 [rk, ck] = size(kern);
6 img_pad = SetPadding(img, kern, pad);
7 op_img = img;
8 if rk==1 && ck==2 %special case, horizontal derivative.
9     for i = 2:r+1
10         for j = 2:c+1
11             op_img(i-1,j-1) = img_pad(i-1,j-1).*kern(1,1)+ img_pad(i-1,j).*kern(1,2) ;
12         end
13     end
14 elseif rk==2 && ck==1 %special case vertical derivative.
15     for i = 2:r+1
16         for j = 2:c+1
17             op_img(i-1,j-1) = img_pad(i-1,j-1).*kern(1,1)+ img_pad(i,j-1).*kern(2,1) ;
18         end
19     end
20 elseif not(mod(rk,2) && mod(ck,2)) %even sized kernels (eg. 2x2, 4x4...)
21     for i = 1:r
22         for j = 1:c
23             for k = 1:rk
24                 for l = 1:ck
25                     su(k,l) = img_pad(i+k-1,j+l-1).*kern(k,l);
26                 end
27             end
28             op_img(i,j) = sum(sum(su));
29         end
30     end
31 else %for odd sized kernels (3x3, 5x5...)
32     for i = ceil(rk/2):r+1
33         for j = ceil(ck/2):c+1
34             for k = -floor(rk/2):floor(rk/2)
35                 for l = -floor(ck/2):floor(ck/2)
36                     su(k+ceil(rk/2),l+ceil(ck/2)) = img_pad(i+k,j+l).*kern(k+ceil(rk/2),l+
37                         ceil(ck/2));
38                 end
39             end
40         end
41     end

```

---

```

39         op_img(i-1,j-1) = sum(sum(su));
40     end
41   end
42 end
43

```

Code 5: Convolution of image

As with any convolution this requires padding of the image to handle edges and corners. The padding code from project 2 has been modified to handle all odd kernel sizes and exceptions as 1x1 and 2x1 image sizes.

```

1 function [img_pad] = SetPadding(img, kern, pad)
2 %SETPADDING Set padding based on kernel size
3 % Set padding to set up an image for convolution based on the kernel size
4 % of the chosen kernel. The size of the kernel is used as a reference for
5 % the amount of padding required. The padding is performed by considering
6 % padding type and then slicing appropriate indices from original image
7 % to copy them to the actual image.
8 [r,c] = size(img); %size of image
9 [rk, ck] = size(kern); %size of kernel
10 if rk==1 && ck==2 %special case for horizontal derivative filter
11   if pad == 1 %zero padding
12     img_pad = zeros(r,c+1); %add a column
13     img_pad(:, 1:end-1) = img;
14   elseif pad == 2 %copy edge
15     img_pad = zeros(r,c+1);
16     img_pad(:, 1:end-1) = img;
17     img_pad(:, end) = img(:,end);
18   elseif pad == 3 %wrap around
19     img_pad = zeros(r,c+1);
20     img_pad(:, 1:end-1) = img;
21     img_pad(:, end) = img(:,1);
22   elseif pad == 4 %reflect across edge
23     img_pad = zeros(r,c+1);
24     img_pad(:, 1:end-1) = img;
25     img_pad(:, end) = img(:,end);
26   end
27 elseif rk==2 && ck==1 %special case for horizontal derivative filter
28   if pad == 1 %zero padding
29     img_pad = zeros(r+1,c);
30     img_pad(1:end-1,:) = img;
31   elseif pad == 2 %copy edge
32     img_pad = zeros(r+1,c);
33     img_pad(1:end-1,:) = img;
34     img_pad(end,:) = img(end,:);
35   elseif pad == 3 %wrap around
36     img_pad = zeros(r+1,c);
37     img_pad(1:end-1,:) = img;
38     img_pad(end,:) = img(1,:);
39   elseif pad == 4 %reflect across edge
40     img_pad = zeros(r+1,c);
41     img_pad(1:end-1,:) = img;
42     img_pad(end,:) = img(end,:);
43   end
44 elseif (mod(rk,2) && mod(ck,2)) %for odd sized kernels. (3x3, 5x5...)
45   if pad == 1 %zero padding
46     img_pad = zeros(r+2*floor(rk/2),c+2*floor(ck/2)); %add ceil(kernelsize/2) number of
47     rows and cols
48     img_pad(ceil(rk/2):end-floor(rk/2), ceil(ck/2):end-floor(ck/2)) = img;
49   elseif pad == 2 %copy edge
50     %set up as zero padding
51     img_pad = zeros(r+2*floor(rk/2),c+2*floor(ck/2));
52     img_pad(ceil(rk/2):end-floor(rk/2), ceil(ck/2):end-floor(ck/2)) = img;

```

```

52     %copy edges to the newly added edges
53     img_pad(ceil(rk/2):end-floor(rk/2), 1:floor(ck/2)) = repmat(img(:,1),1,floor(ck/2));
54     img_pad(ceil(rk/2):end-floor(rk/2), end-floor(ck/2)+1:end) = repmat(img(:,end),1,floor(
55         rk/2));
56     img_pad(1:floor(rk/2),ceil(ck/2):end-floor(ck/2)) = repmat(img(1,:),floor(rk/2),1);
57     img_pad(end-floor(rk/2)+1:end,ceil(ck/2):end-floor(ck/2)) = repmat(img(end,:),floor(ck
58         /2),1);
59     %corner cases
60     img_pad(1:floor(rk/2), 1:floor(ck/2)) = repmat(img(1,1), floor(rk/2), floor(ck/2));
61     img_pad(end-floor(rk/2)+1:end, 1:floor(ck/2)) = repmat(img(end,1), floor(rk/2), floor(
62         ck/2));
63     img_pad(1:floor(rk/2), end-floor(ck/2)+1:end) = repmat(img(1,end), floor(rk/2), floor(
64         ck/2));
65     img_pad(end-floor(rk/2)+1:end, end-floor(ck/2)+1:end) = repmat(img(end,end), floor(rk
66         /2), floor(ck/2));
67 elseif pad == 3 %wrap around
68     %set up as zero padding
69     img_pad = zeros(r+2*floor(rk/2),c+2*floor(ck/2));
70     img_pad(ceil(rk/2):end-floor(rk/2), ceil(ck/2):end-floor(ck/2)) = img;
71     %wrap edges from other end to the newly added edges
72     try
73         img_pad(floor((rk+1)/2):end-floor(rk/2), 1:floor(ck/2)) = img(:,end-floor(ck/2)+1:
74             end);
75         img_pad(floor((rk+1)/2):end-floor(rk/2), end-floor(ck/2)+1:end) = img(:,1:floor(ck
76             /2));
77         img_pad(1:floor(rk/2),floor((ck+1)/2):end-floor(ck/2)) = img(end-floor(rk/2)+1:end
78             ,:);
79         img_pad(end-floor(rk/2)+1:end,floor((ck+1)/2):end-floor(ck/2)) = img(1:floor(rk/2)
80             ,:);
81         %corner cases
82         img_pad(1:floor(rk/2), 1:floor(ck/2)) = img(end-floor(rk/2)+1:end, end-floor(ck/2)
83             +1:end);
84         img_pad(end-floor(rk/2)+1:end, 1:floor(ck/2)) = img(1:floor(rk/2), end-floor(ck/2)
85             +1:end);
86         img_pad(1:floor(rk/2), end-floor(ck/2)+1:end) = img(end-floor(rk/2)+1:end, 1:floor(
87             ck/2));
88         img_pad(end-floor(rk/2)+1:end, end-floor(ck/2)+1:end) = img(1:floor(rk/2), 1:floor(
89             ck/2));
90         catch
91         end
92     %copy edges to the newly added edges
93     img_pad(ceil(rk/2):end-floor(rk/2), 1:floor(ck/2)) = repmat(img(:,1),1,floor(ck/2));
94     img_pad(ceil(rk/2):end-floor(rk/2), end-floor(ck/2)+1:end) = repmat(img(:,end),1,floor(
95         rk/2));
96     img_pad(1:floor(rk/2),ceil(ck/2):end-floor(ck/2)) = repmat(img(1,:),floor(rk/2),1);
97     img_pad(end-floor(rk/2)+1:end,ceil(ck/2):end-floor(ck/2)) = repmat(img(end,:),floor(ck
98         /2),1);
99     %corner cases
100    img_pad(1:floor(rk/2), 1:floor(ck/2)) = repmat(img(1,1), floor(rk/2), floor(ck/2));
101    img_pad(end-floor(rk/2)+1:end, 1:floor(ck/2)) = repmat(img(end,1), floor(rk/2), floor(
102        ck/2));
103    img_pad(1:floor(rk/2), end-floor(ck/2)+1:end) = repmat(img(1,end), floor(rk/2), floor(
104        ck/2));
105    img_pad(end-floor(rk/2)+1:end, end-floor(ck/2)+1:end) = repmat(img(end,end), floor(rk
106        /2), floor(ck/2));
107 elseif pad == 4 %reflect across edge
108     %set up as zero padding
109     img_pad = zeros(r+2*floor(rk/2),c+2*floor(ck/2));
110     img_pad(ceil(rk/2):end-floor(rk/2), ceil(ck/2):end-floor(ck/2)) = img;
111     try
112         %reflect across edges
113         img_pad(floor((rk+1)/2):end-floor(rk/2), 1:floor(ck/2)) = img(:,floor(ck/2):-1:1);
114         img_pad(floor((rk+1)/2):end-floor(rk/2), end-floor(ck/2)+1:end) = img(:,end:-1:end-
115             floor(ck/2)+1);
116         img_pad(1:floor(rk/2),floor((ck+1)/2):end-floor(ck/2)) = img(floor(rk/2):-1:1,:);
117     end

```

```

98     img_pad(end-floor(rk/2)+1:end,floor((ck+1)/2):end-floor(ck/2)) = img(end:-1:end-
99         floor(rk/2)+1,:);
100        %corner cases
101        img_pad(1:floor(rk/2), 1:floor(ck/2)) = img(floor(rk/2):-1:1, floor(ck/2):-1:1);
102        img_pad(end-floor(rk/2)+1:end, 1:floor(ck/2)) = img(end:-1:end-floor(rk/2)+1, floor(
103            ck/2):-1:1);
104        img_pad(1:floor(rk/2), end-floor(ck/2)+1:end) = img(floor(rk/2):-1:1, end:-1:end-
105        floor(ck/2)+1);
106        img_pad(end-floor(rk/2)+1:end, end-floor(ck/2)+1:end) = img(end:-1:end-floor(rk/2)
107        +1, end:-1:end-floor(ck/2)+1);
108        catch
109            %copy edges to the newly added edges
110            img_pad(ceil(rk/2):end-floor(rk/2), 1:floor(ck/2)) = repmat(img(:,1),1,floor(ck/2))
111        ;
112            img_pad(ceil(rk/2):end-floor(rk/2), end-floor(ck/2)+1:end) = repmat(img(:,end),1,
113                floor(rk/2));
114            img_pad(1:floor(rk/2),ceil(ck/2):end-floor(ck/2)) = repmat(img(1,:),floor(rk/2),1);
115            img_pad(end-floor(rk/2)+1:end,ceil(ck/2):end-floor(ck/2)) = repmat(img(end,:),floor(
116                ck/2),1);
117            %corner cases
118            img_pad(1:floor(rk/2), 1:floor(ck/2)) = repmat(img(1,1), floor(rk/2), floor(ck/2));
119            img_pad(end-floor(rk/2)+1:end, 1:floor(ck/2)) = repmat(img(end,1), floor(rk/2),
120                floor(ck/2));
121            img_pad(1:floor(rk/2), end-floor(ck/2)+1:end) = repmat(img(1,end), floor(rk/2),
122                floor(ck/2));
123            img_pad(end-floor(rk/2)+1:end, end-floor(ck/2)+1:end) = repmat(img(end,end), floor(
124                rk/2), floor(ck/2));
125        end
126    end
127 else %for even sized kernels (eg. 2x2, 4x4...)
128 if pad == 1 %zero padding
129     img_pad = zeros(r+rk/2,c+ck/2);
130     img_pad(1:end-(rk/2), 1:end-(ck/2)) = img;
131 elseif pad == 2 %copy edge
132     %set up as zero padding
133     img_pad = zeros(r+rk/2,c+ck/2);
134     img_pad(1:end-(rk/2), 1:end-(ck/2)) = img;
135     %copy edges to the image.
136     img_pad(1:end-(rk/2), end-ck/2+1:end) = repmat(img(:,end), floor(ck/2));
137     img_pad(end-rk/2+1:end,1:end-(ck/2)) = repmat(img(end,:), floor(rk/2));
138     %corner cases
139     img_pad(end-(rk/2)+1:end, end-(ck/2)+1:end) = repmat(img(end,end), floor(rk/2), floor(ck
140         /2));
141 elseif pad == 3 %wrap around
142     %set up as zero padding
143     img_pad = zeros(r+rk/2,c+ck/2);
144     img_pad(1:end-(rk/2), 1:end-(ck/2)) = img;
145     %wrap around edges
146     img_pad(1:end-1, end-ck/2+1:end) = img(:,1:ck/2);
147     img_pad(end-rk/2+1:end,1:end-1) = img(1:rk/2,:);
148     %corner case
149     img_pad(end-(rk/2)+1:end, end-(ck/2)+1:end) = img(1:(rk/2),1:(ck/2));
150 elseif pad == 4 %reflect across edge
151     img_pad = zeros(r+rk/2,c+ck/2);
152     img_pad(1:end-1, 1:end-1) = img;
153     %reflect across edge
154     img_pad(1:end-1, end-ck/2+1:end) = img(:,end:end-ck/2+1);
155     img_pad(end-rk/2+1:end,1:end-1) = img(end:end-rk/2+1,:);
156     %corner case
157     img_pad(end-(rk/2)+1:end, end-(ck/2)+1:end) = img(end:end-(rk/2)+1,end:end-(ck/2)+1);
158 end
159 end
160 end

```

Code 6: Set padding for convolution

b) Create binary mask - Rectangular, Elliptical or Freeform. The image resizing and padding has been handled. The user needs to point out source and target location wherever required to align the faces/ source-target pair properly.

```

1 function [fgo, roi, bw_mask1] = createROIMask(fg, bg, roitype)
2 %CREATEMASK Create Mask based on ROI type and returns the resized image
3 % and mask
4 % Returns the padded image and mask based on the selected ROI. The user
5 % has 3 options - rectangular, elliptical and freeform.
6 %check roitype
7 if(roitype == 1)
8     roi = drawrectangle('Label', 'ROI', 'color', [1 0 0]);
9 elseif(roitype == 2)
10    roi = drawellipse('Label', 'ROI', 'color', [1 0 0]);
11 else
12    roi = drawfreehand('Label', 'ROI', 'color', [1 0 0]);
13 end
14 %create mask from the selected ROI
15 bw_mask = createMask(roi);
16 [rb, cb, bch] = size(bg);
17 [rf, cf, fch] = size(fg);
18 %resize/ pad fg based on its original size.
19 if(rf*cf>rb*cb)
20     fgo=upscale(fg, [rb,cb,bch]);
21     bw_mask1 = upscale(bw_mask, [rb,cb]);
22 elseif(rf*cf<rb*cb)
23     fgo = zeros(rb,cb,bch);
24     bw_mask1 = zeros(rb,cb);
25     % center the smaller fg and its mask by zero padding to bg size
26     fgo=floor(rb/2)+(1:rf)-floor(rf/2), floor(cb/2)+(1:cf)-floor ...
27     (cf/2),:) = fg;
28     bw_mask1(floor(rb/2)+(1:rf)-floor(rf/2), floor(cb/2)+(1:cf)- ...
29     floor(cf/2)) = bw_mask;
30 else
31     fgo = fg;
32     bw_mask1 = bw_mask;
33 end
34 end

```

Code 7: CreateROIMask.m Subroutine to create Mask

c) Blending of images and alignment.

The blending can be treated as  $\alpha$  blending where the  $\alpha$  can be taken as the Gaussian pyramid of the mask.

The formula for blending can be taken as follows

$$\text{blendedImg}(i) = lPyrA(i) * gPyr(i) + (1 - gPyr(i)) * lPyrB(i)$$

We do the operations layer by layer and then collapse the pyramid.

```

1 function [blendedImg] = blendPyramid(bg, fg, bw_mask, nlayers)
2 %BLENDPYRAMID blend the pyramid based on selected bg fg and mask.
3 % Calculates the Laplacian Pyramids of images and Gaussian Pyramid of the
4 % mask and does alpha blending to get the blended pyramid. Then collapses
5 % the pyramid to recover the blended image.
6 [~, lPyrBG] = ComputePyr(bg, nlayers); %calculate laplacian pyr of 1st img
7 [~, lPyrFG] = ComputePyr(fg, nlayers); %calculate laplacian pyr of 2nd img
8 [gPyrMask,~] = ComputePyr(double(bw_mask), nlayers); %gaussian pyr of mask
9 for i = 1:length(gPyrMask)
10     %calculate blended pyramid based on alpha blending with mask gpyr
11     lblend{i} = (gPyrMask{i}) .* lPyrBG{i} + (1-(gPyrMask{i})).*lPyrFG{i};
12 end
13 layer = length(lblend);
14 %generate gaussian kernel
15 dscale = 2.5;
16 sigm = (2*dscale)./6;
17 kernS = floor(4*sigm+0.5);

```

```

18     h = fspecial('gaussian', kernS ,sigm);
19 %     h = ([1;4;6;4;1]*[1 4 6 4 1])./256; % can be implemented as separable
20 % filter as shown
21
22 %collapse pyramid
23 while(layer>1)
24     lblend{layer-1} = conv2d(upscale(lblend{layer}), size(lblend{layer-1})),h,3)+lblend{
25         layer-1;
26     end
27     blendedImg = lblend{1};
28 end

```

Code 8: Blending of images

## RESULTS

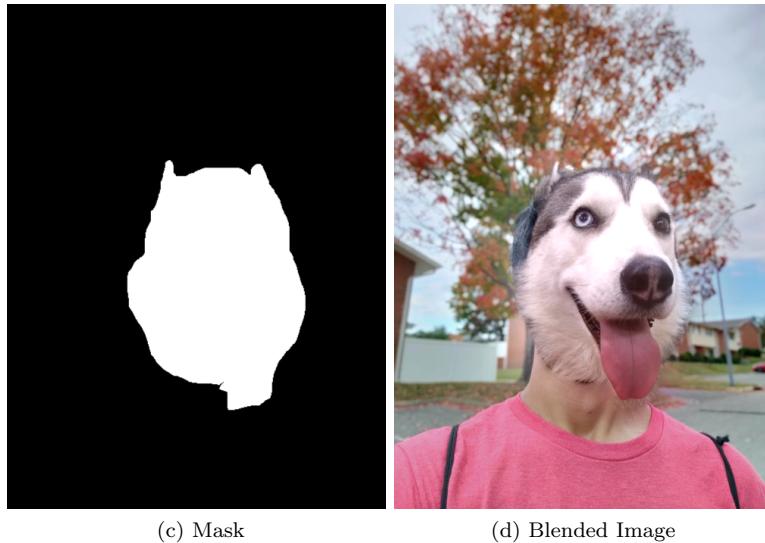
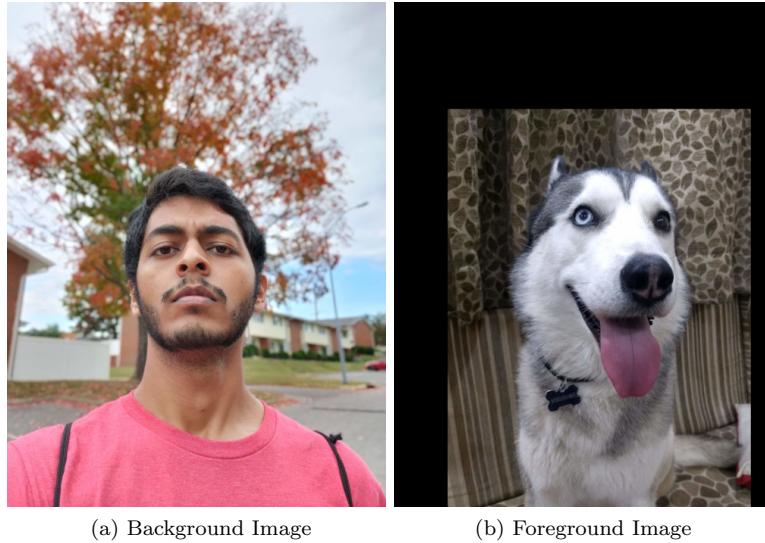


Figure 2: Example 1

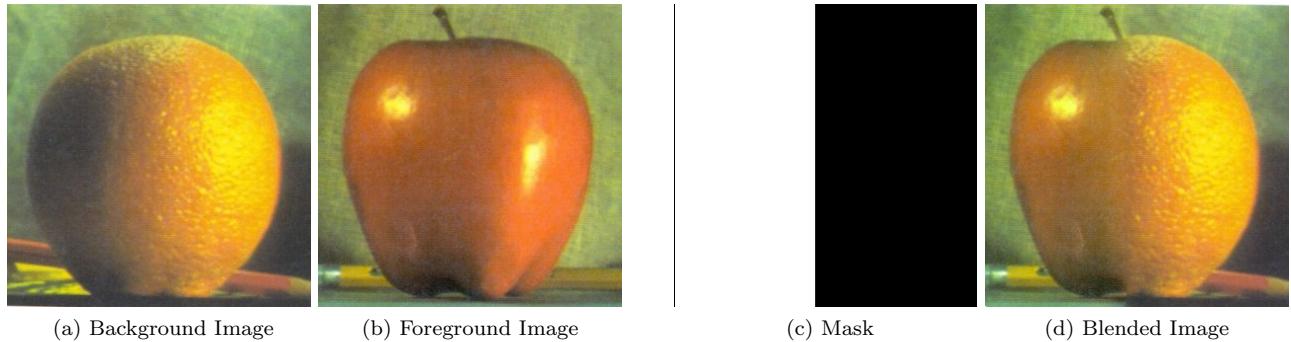
**Project 03**

Figure 3: Example 2

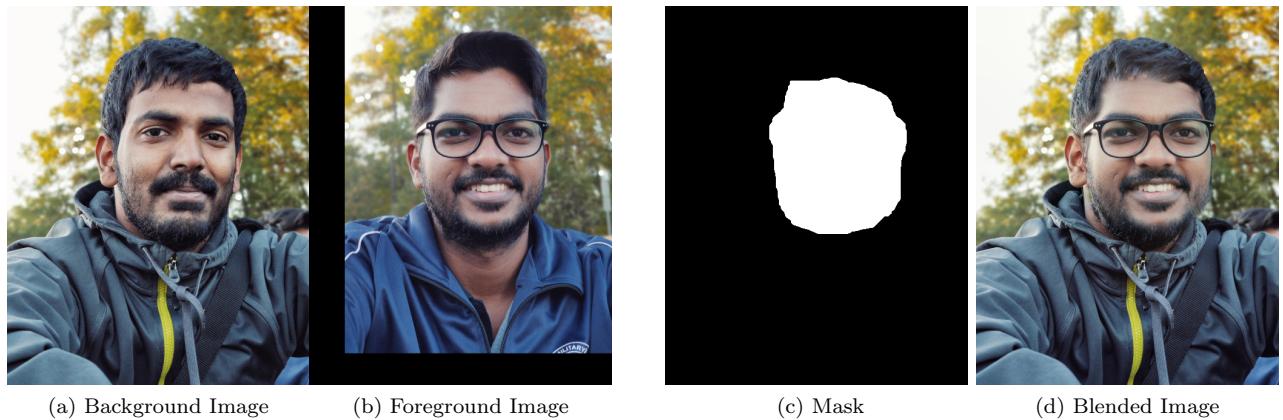


Figure 4: Example 3

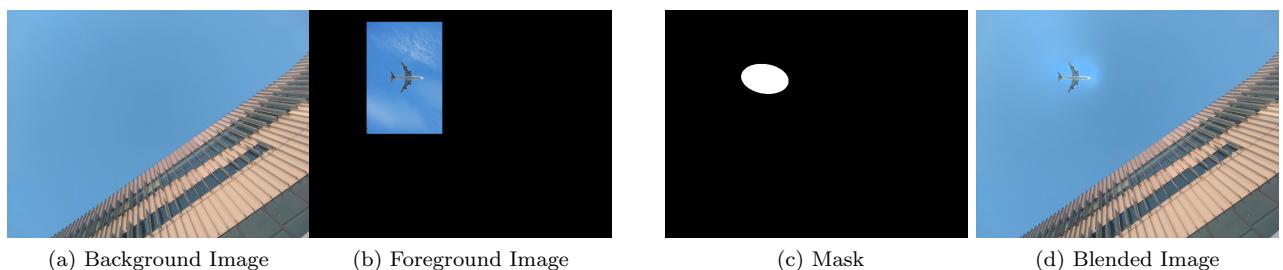


Figure 5: Example 4

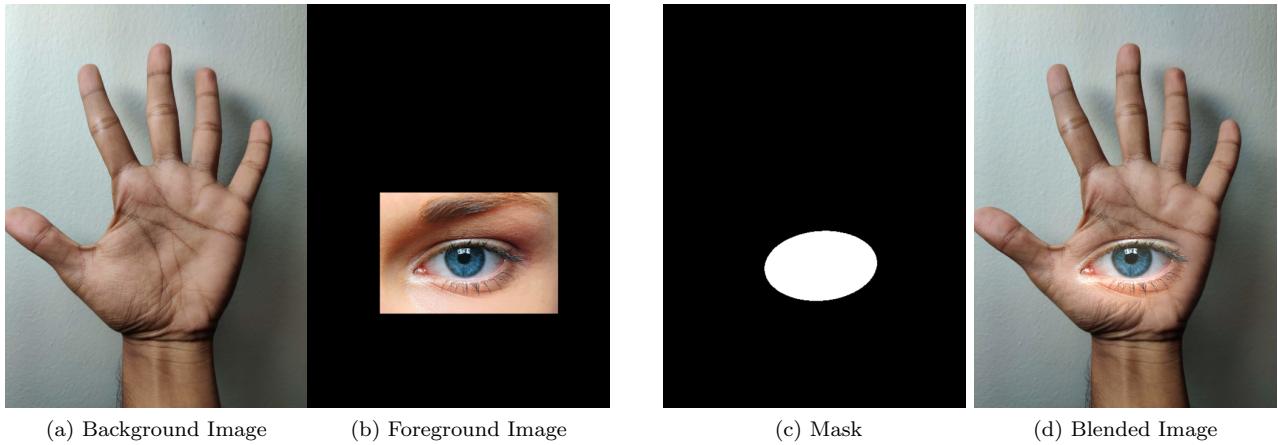


Figure 6: Example 5

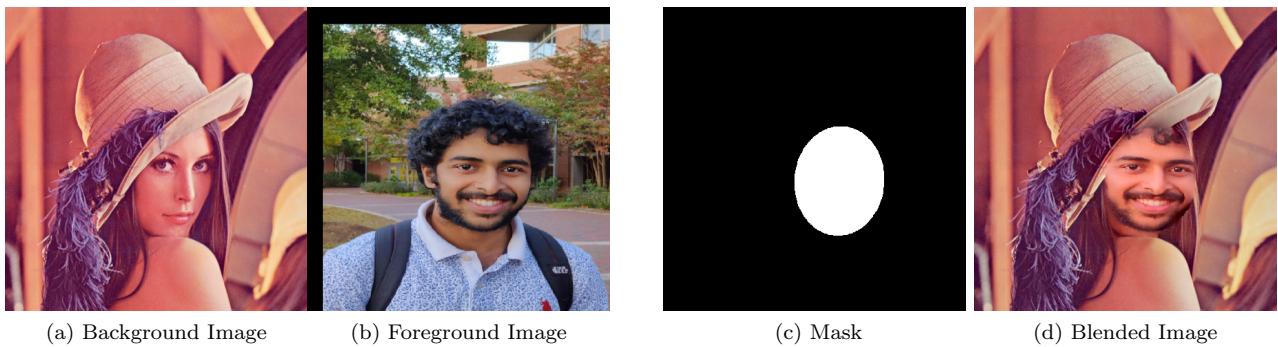


Figure 7: Example 6