1. Two-dimensional convolution

   (a) *Write a function to implement* `g = conv2(f, w, pad)`*, where* `f` *is an input image (grey or RGB),* `w` *is a 2-D kernel, and* `pad` *represents the four padding types: clip/zero-padding, wrap-around, copy-edge, and reflect* Using a kernel size of 20×20, the following images are the different padded results of the $512 \times 512 \times 3$ `lena.tiff`

   - **Clip/zero-padding**
     Using MATLAB, this was significantly the easiest one to implement. Before all other types of padding, zero-padding is always done to initialize the final size of the padded result to increase efficiency. First find the size of the kernel, and create a matrix of 0's sized

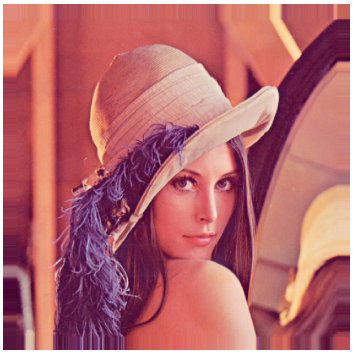
     $$[f_{height} + 2(w_{height} - 1),\ f_{width} + 2(w_{width} - 1),\ f_{channels}]$$

     Then, simply each channel of the input image f starting at index $[w_{height}, w_{width}]$. This can be seen in the 2 lines of MATLAB below

```
% amount to pad
h_pad = w_h - 1;
w_pad = w_w - 1;
% zero-padding (used for pre-initialization of all cases)
f_post = zeros(f_h + (2 * h_pad), f_w + (2 * w_pad), channels);
f_post((w_h:(w_h + f_h - 1)), (w_w:(w_w + f_w - 1)), :) = f;
```

clip/zero-padding

reflect



copy-edge

wrap-around

- **Wrap-around**
  In order to accomplish the rest of the types of padding, the zero-padded image is used and then further manipulated across each channel. For wrap-around, we simply have to create a 'tiling' effect by copying the edges from one side to another. The order of operations does not matter, but I went with doing left/right sides then top/bottom. The indices were calculated to be accurate and are often reused for all the padding types. The associated figure can be found above. The MATLAB listing is below, where each operation is done in one line.

```
% wrap-around
f_post(:, 1:w_pad, :) = f_post(:, (f_w + 1):(f_w + w_pad), :);
f_post(:, (f_w + w_pad + 1):end, :) = f_post(:, (w_pad + 1):(2 * ...
    w_pad), :);
f_post(1:h_pad, :, :) = f_post((f_h + 1):(f_h + h_pad), :, :);
f_post((f_h + h_pad + 1):end, :, :) = f_post((h_pad + 1):(2 * ...
    h_pad), :, :);
```

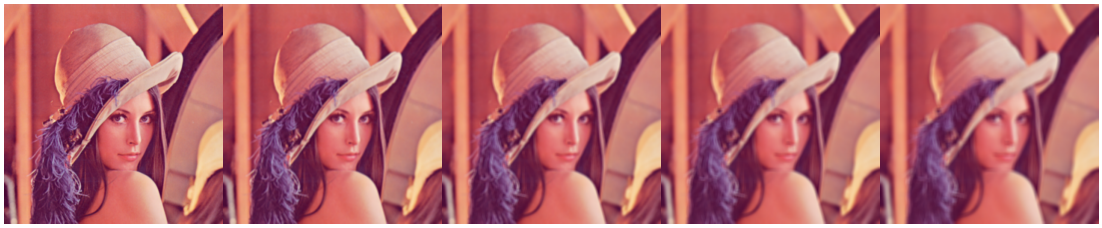
- **Copy-edge**
  Similar indices used in copy-edge, however, instead of copy and pasting a sub-matrix from one position to another, I take use the vector at outermost edge of the image and run it through a MATLAB function called `repmat`, which repeats the vector given my specified dimensions and number of repitions. The figure is above and the listing is below.

```
% copy-edge
f_post(:, 1:w_pad, :) = repmat(f_post(:, w_pad + 1, :), 1, w_pad);
f_post(:, (f_w + w_pad + 1):end, :) = repmat(f_post(:, f_w + ...
    w_pad, :), 1, w_pad);
f_post(1:h_pad, :, :) = repmat(f_post(h_pad + 1, :, :), h_pad, 1);
f_post((f_h + h_pad + 1):end, :, :) = repmat(f_post(f_h + h_pad, ...
    :, :), h_pad, 1);
```

- **Reflect**
  The indicies for a reflection did not have to be calculated, since we can simply use same indices from wrap-around. However, we would have to swap their positions and flip the array across some specified dimension. Using MATLABs `fliplr` and `flipud`, I was able to index each matrix and flip them across certain dimensions to create the reflective effect. The figure is above and the listing is below

```
% reflection
f_post(:, 1:w_pad, :) = fliplr(f_post(:, (w_pad + 1):(2 * ...
    w_pad), :));
f_post(:, (f_w + w_pad + 1):end, :) = fliplr(f_post(:, (f_w + ...
    1):(f_w + w_pad), :));
f_post(1:h_pad, :, :) = flipud(f_post((h_pad + 1):(2 * h_pad), ...
    :, :));
f_post((f_h + h_pad + 1):end, :, :) = flipud(f_post((f_h + ...
    1):(f_h + h_pad), :, :));
```

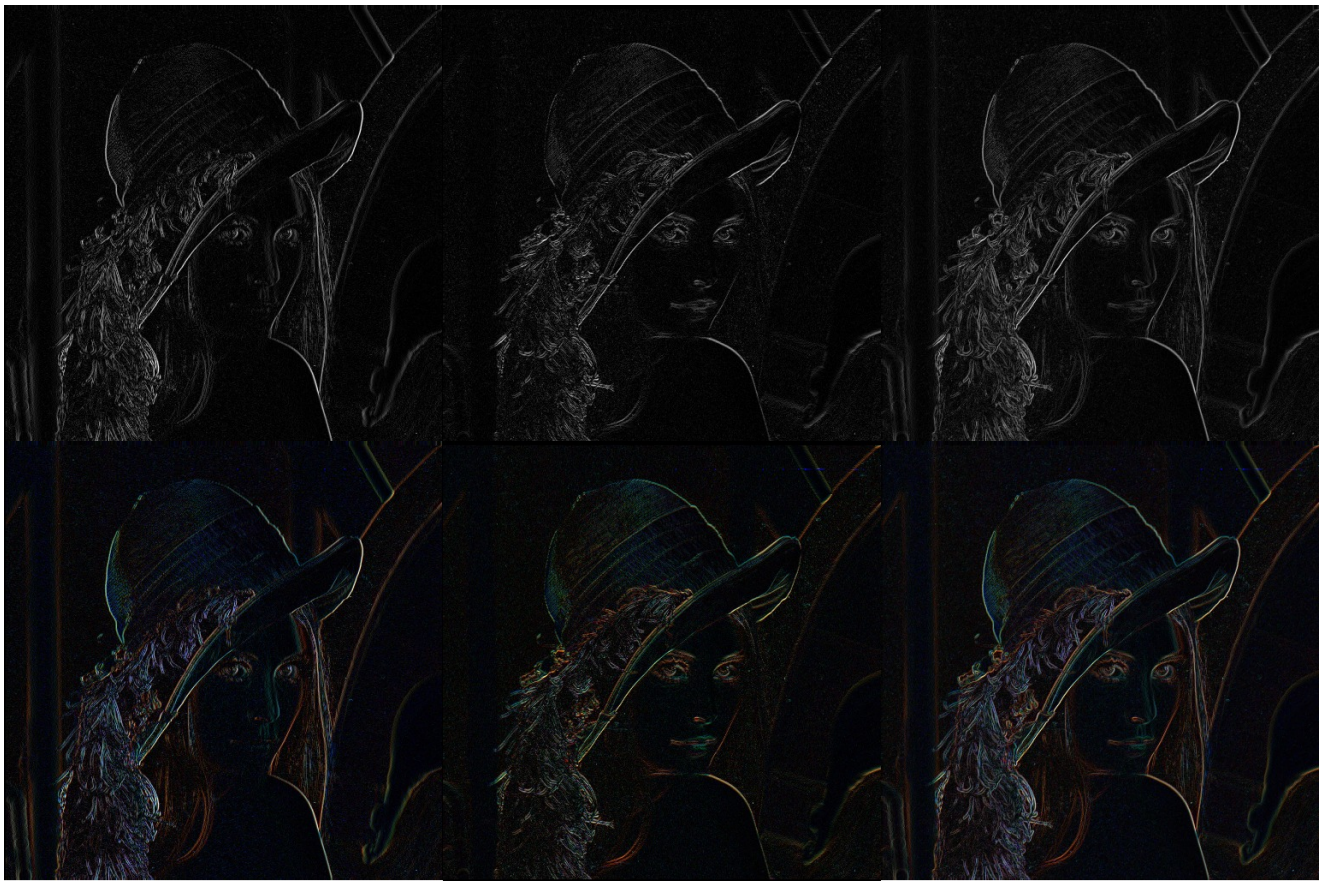
i. **Box filter**
Given a weighted 3×3 kernel of 1's, the result is a blurred form of the image. In Figure 1, I have various forms of Lena with different kernel sizes.



3 5 7 9 11

Figure 1: Box kernels applied to `lena.tiff` of various sizes

ii. **First order derivative filters**

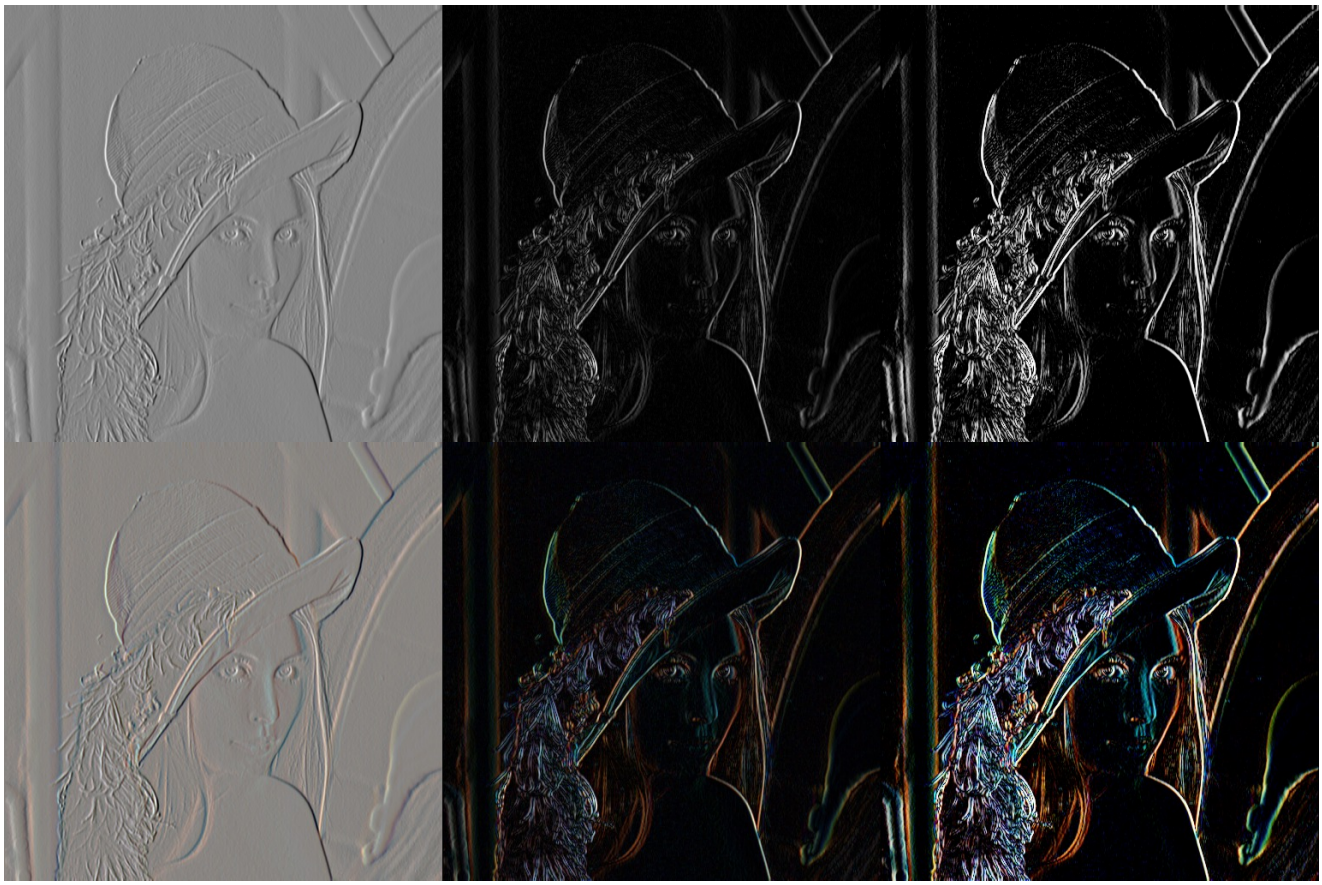


$\nabla_x f$ $\nabla_y f$ $|\nabla f|$

Figure 2: First order derivative filters applied to greyscale and RGB versions of `lena.tiff`

The first order derivative filters are simple filters represented by $w \in \mathbb{R}^{1\times 2}$ or $\mathbb{R}^{2\times 1}$. The images are displayed in the following manner: Left-most: gradient w.r.t. the horizontal direction, Middle: gradient w.r.t. the vertical direction, Right-most: the magnitude of both horizontal and vertical gradients.

iii. **Prewitt, Sobel, and Roberts operators**



Step 1. Step 2. Step 3.

Figure 3: Transformations after applying Prewitt operator w_x

The Prewitt operator, defined as

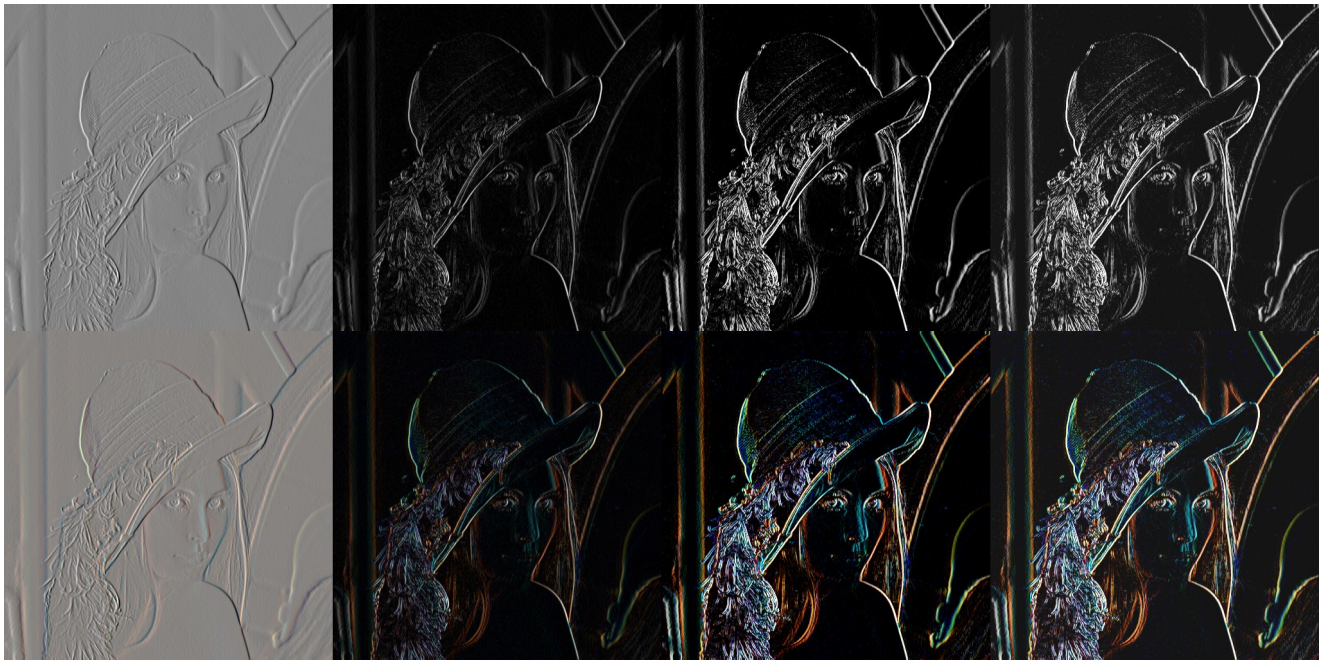
$$w_x = \begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}, \; w_y = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

Steps for Figure 3:

Step 1. Prewitt operator (x direction) on greyscale `lena.tiff`

Step 2. Square difference from median

Step 3. Histogram matching with exponential distribution



Step 1. Step 2. Step 3. Step 4.

Figure 4: Transformations after applying Sobel operator w_x

The Sobel operator, defined as

$$w_x = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}, \; w_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

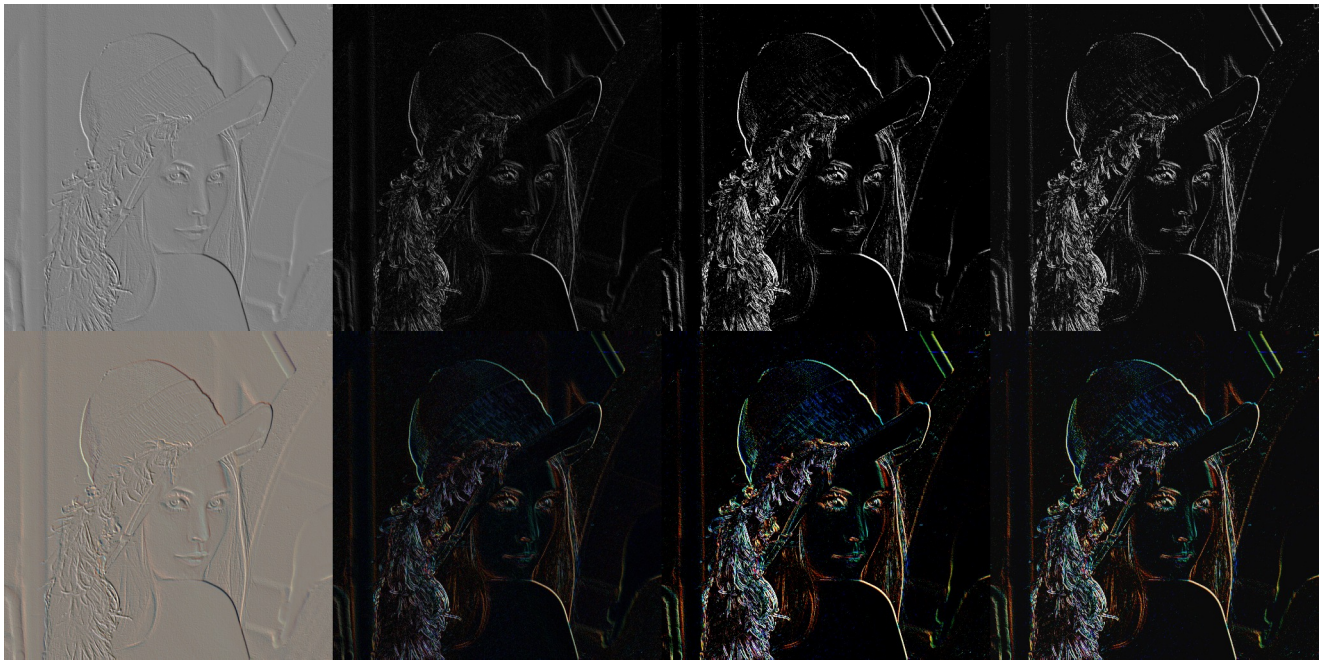
Steps for Figure 4:

Step 1. Sobel operator (x direction) on greyscale `lena.tiff`

Step 2. Square difference from median

Step 3. Histogram matching with exponential distribution

Step 4. Dilate range $\in [20,\ 255]$



Step 1. Step 2. Step 3. Step 4.

Figure 5: Transformations after applying Roberts operator w_x

The Roberts operator, defined as

$$w_x = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \; w_y = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Steps for Figure 5:

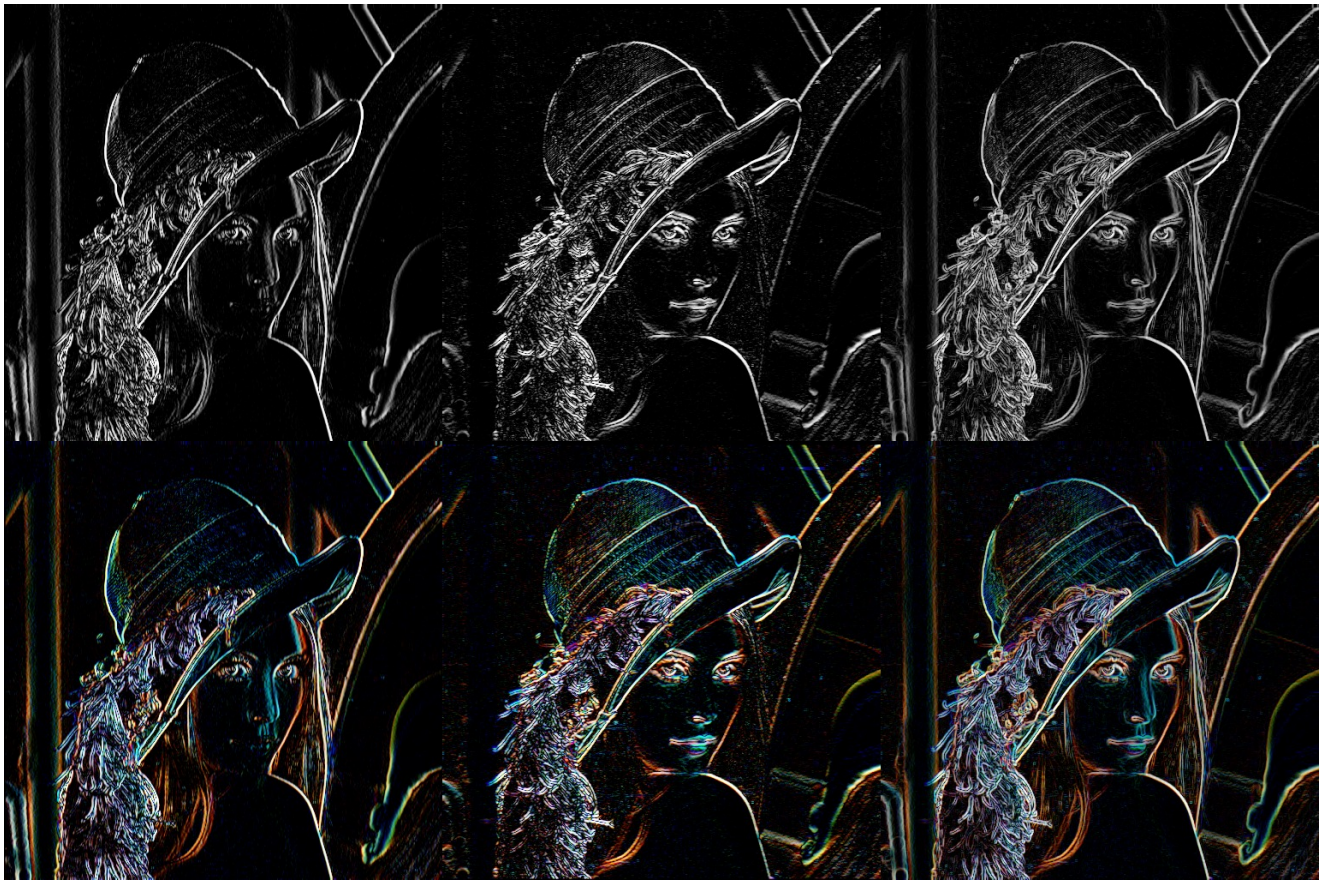
Step 1. Roberts operator (x direction) on greyscale `lena.tiff`

Step 2. Square difference from median

Step 3. Histogram matching with exponential distribution

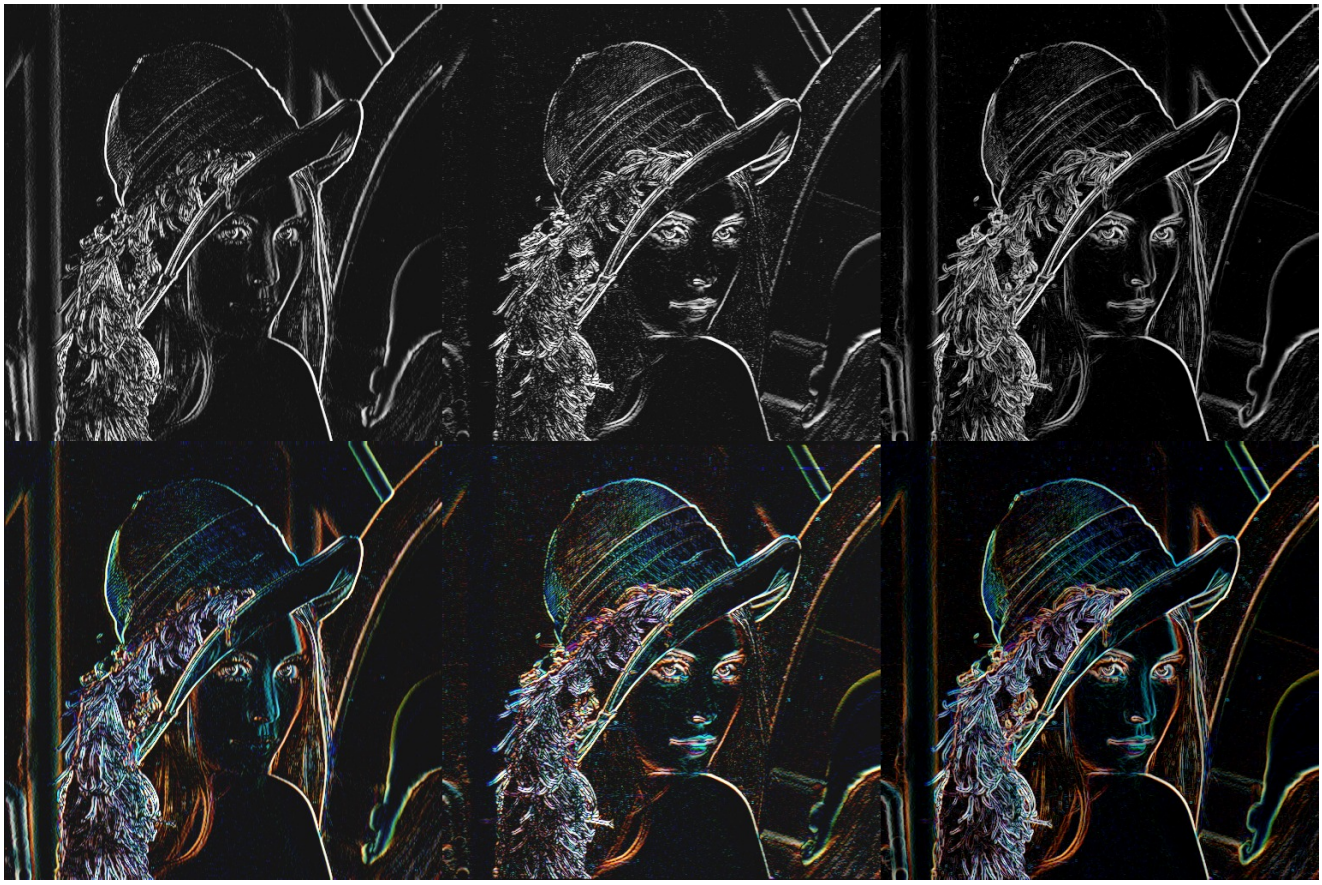
Step 4. Dilate range $\in [10,\ 220]$

The MATLAB listing for the `conv2me()` function is shown in the appendix, alongside the testbench used to create these images.



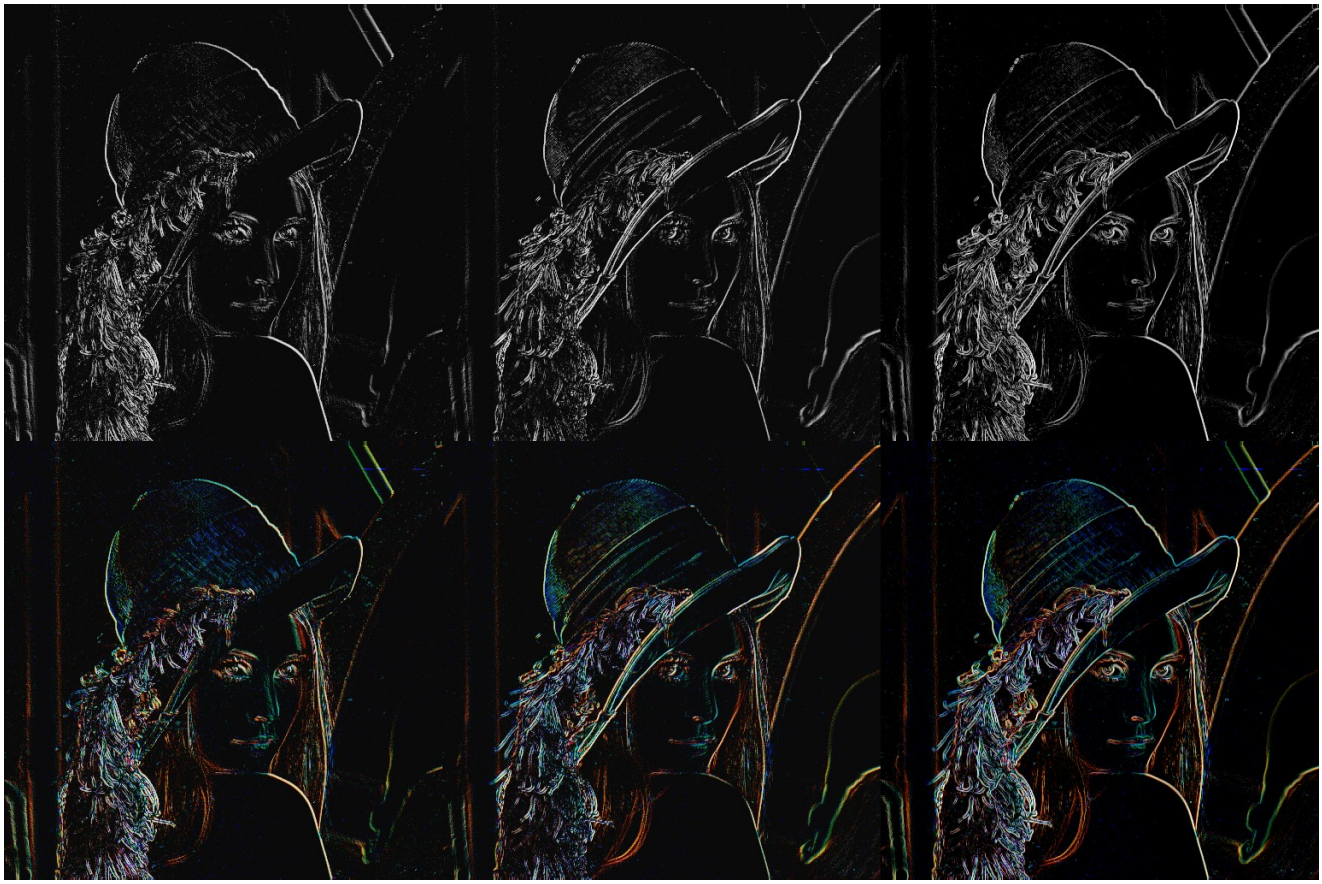
$\nabla_x f$ $\nabla_y f$ $|\nabla f|$

Figure 6: Prewitt operator applied to greyscale and RGB versions of `lena.tiff`



$\nabla_x f$ $\nabla_y f$ $|\nabla f|$

Figure 7: Sobel operator applied to greyscale and RGB versions of `lena.tiff`



$\nabla_x f$ $\nabla_y f$ $|\nabla f|$

Figure 8: Roberts operator applied to greyscale and RGB versions of `lena.tiff`

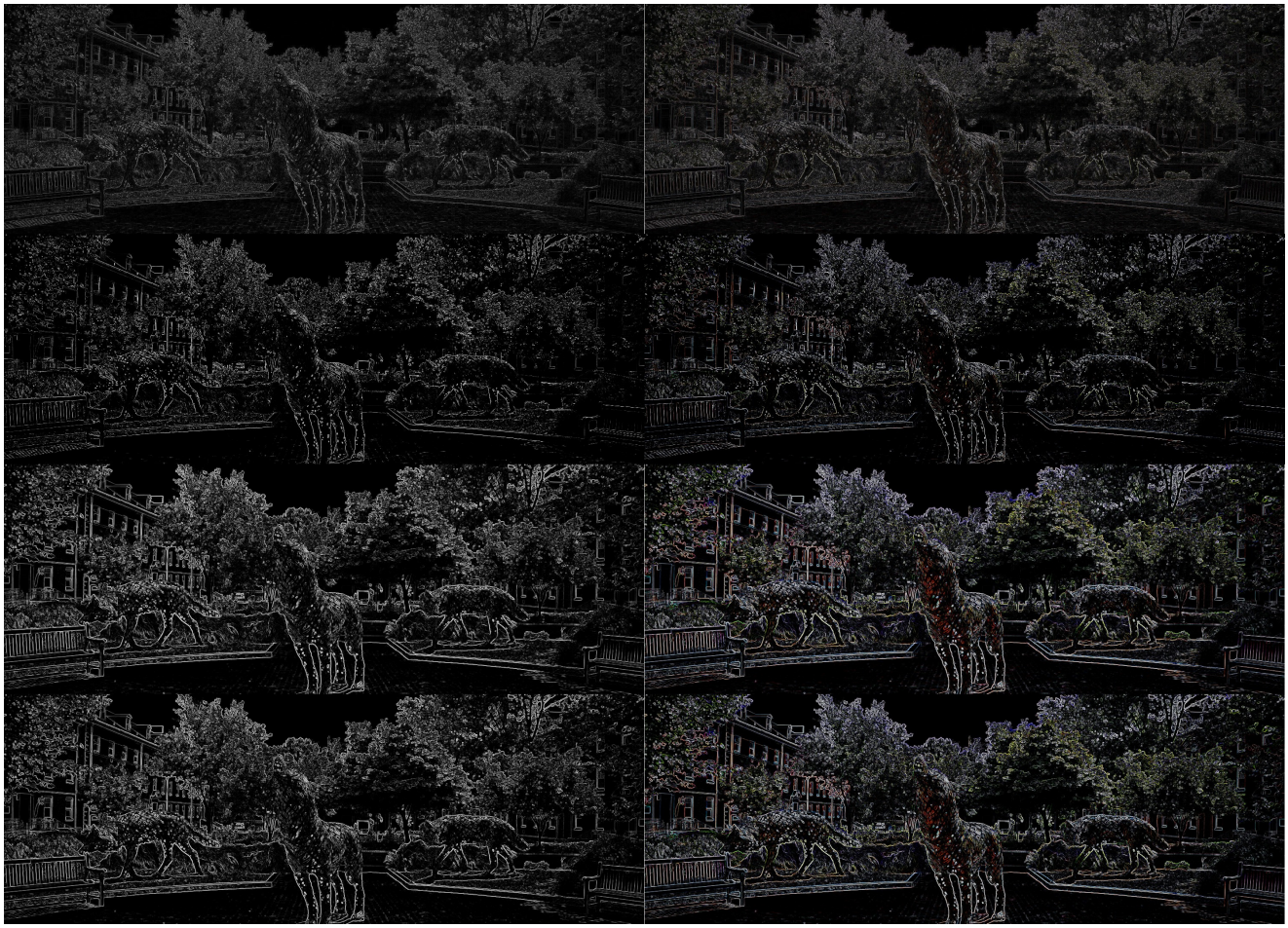


Figure 9: All operators applied to greyscale and RGB versions of `wolves.png` and resulting magnitudes. Top to bottom: First order derivative, Roberts, Prewitt, Sobel

(b) *Create a grey image of size 1024 × 1024 that consists of a unit impulse at the center of the image (512, 512) and zero elsewhere. Use this image and a kernel of your choice to confirm that your function is indeed performing convolution. Show your filter result and explain why your function is performing convolution.*

I decided to use `lena.tiff` as a kernel to show how my convolution function is working properly. Intuitively, if there is a unit impulse in the center of an image where the rest of the pixels are 0, then as the kernel sweeps through during convolution, only the pixel which lines up with the centered impulse will contribute to the weighted sum of the resulting convolution. This result is then stored in 1 pixel location within the output image. Meaning, that this gives the effect of a simple transformation. No matter how the kernel sweeps across the unit impulse, each pixel which coincides with the unit impulse is stored relatively closer to the center of the kernel, meaning that there is a reflection across the origin (also interpreted as a reflection across both the x and y planes, where the unit impulse acts as the origin). The result of the unit impulse image filtered using a downsampled greyscale version of `lena.tiff` is shown in Figure 10

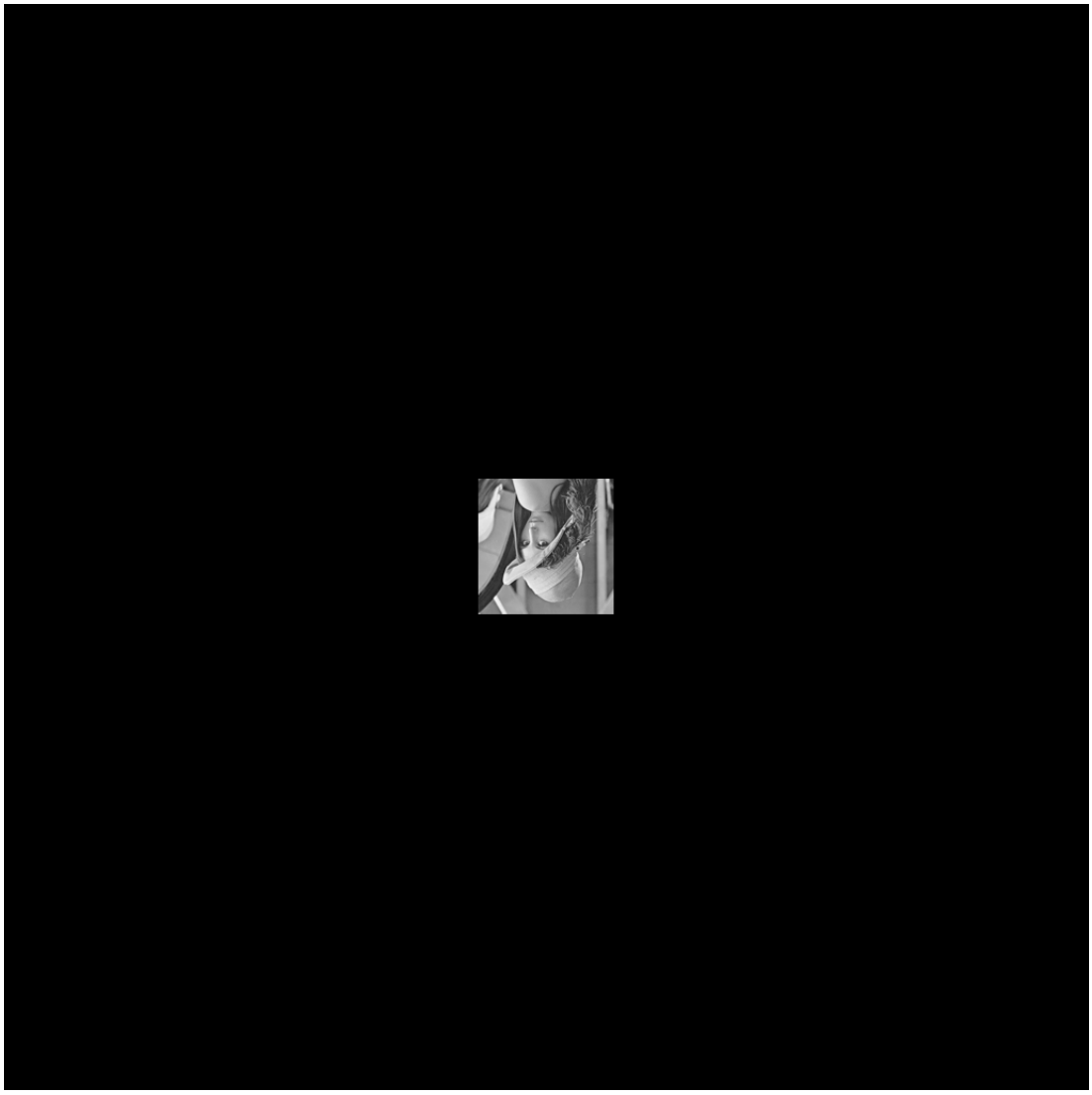


Figure 10: Unit impulse at [512, 512] and zero everywhere else, convolved with kernel of downsampled greyscale `lena.tiff`

2. Implementing and testing the 2-D FFT and its inverse using a built-in 1-D FFT algorithm

   (a) *Use the built-in 1-D FFT (MATLAB `fft()`) to implement `F = dft2(f)` from scratch, where `f` is an input grey image. Test your function with the provided lena.png and wolves.png (you can use built-in color conversion functions to convert them to grey images). Before apply the DFT2, you need to scale the grey image to the range [0, 1] (e.g., implement Problem 4 in HW02 and integrate it in your DFT2 function). Visualize the spectrum and phase angle image. When visualizing them, apply the transform $s = log(1 + |F|)$ or others as seen fit.*

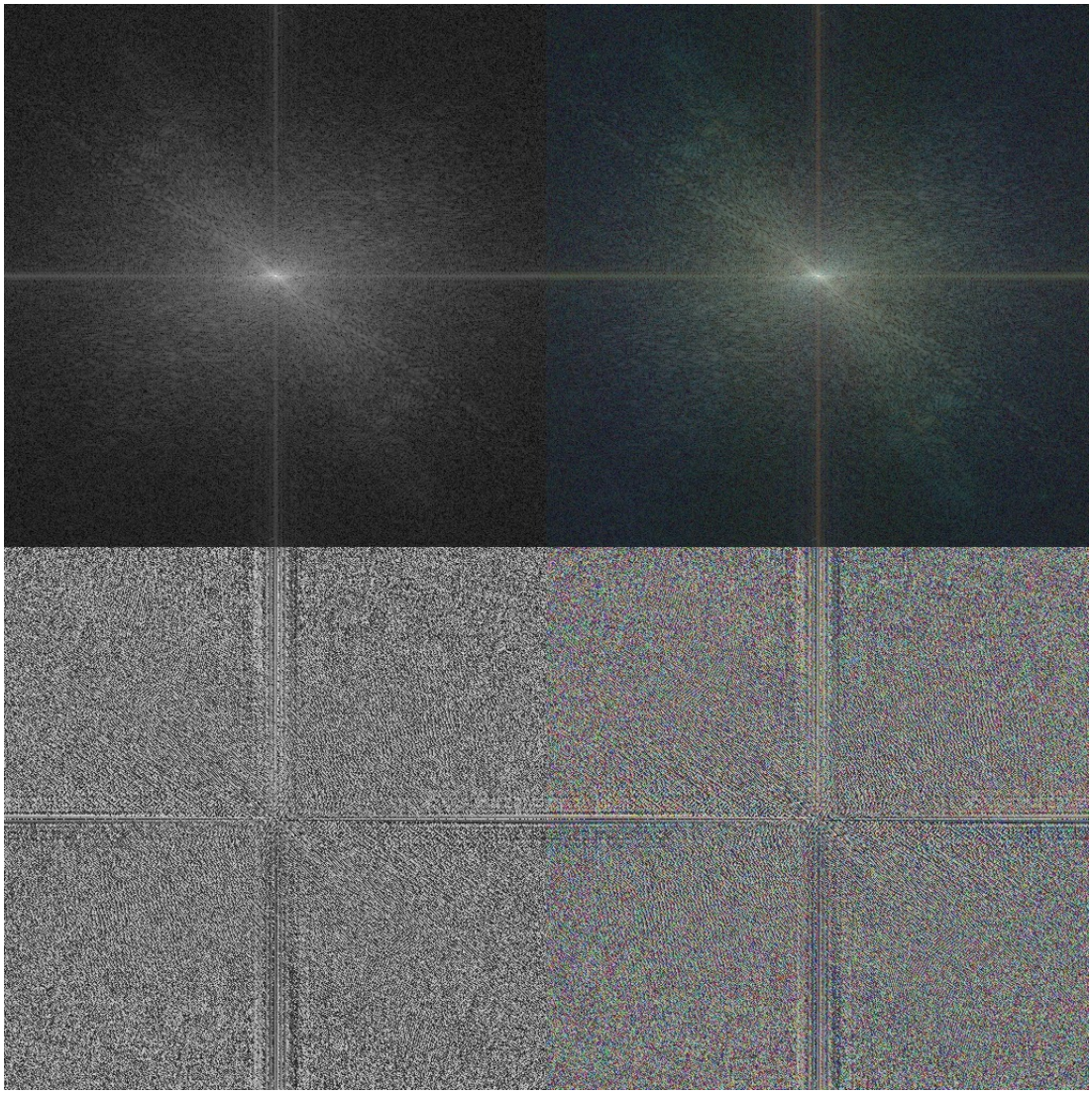


Figure 11: Log transformation on magnitude spectrum of greyscale and RGB `lena.tiff`. Below, phase of spectrum of greyscale and RGB `lena.tiff`

The MATLAB listing for the `dft2()` function is shown in the appendix. In essence, a 1D FFT had to be performed along each axis of the image. This resulted in my function `dft2()` to be identical to MATLAB's built in function `fft2()`, where the magnitude of error averages 10^{-12} near the peak of the spectrum and as low as 10^{-15} in the extremities. This is also replicated when comparing the phase.

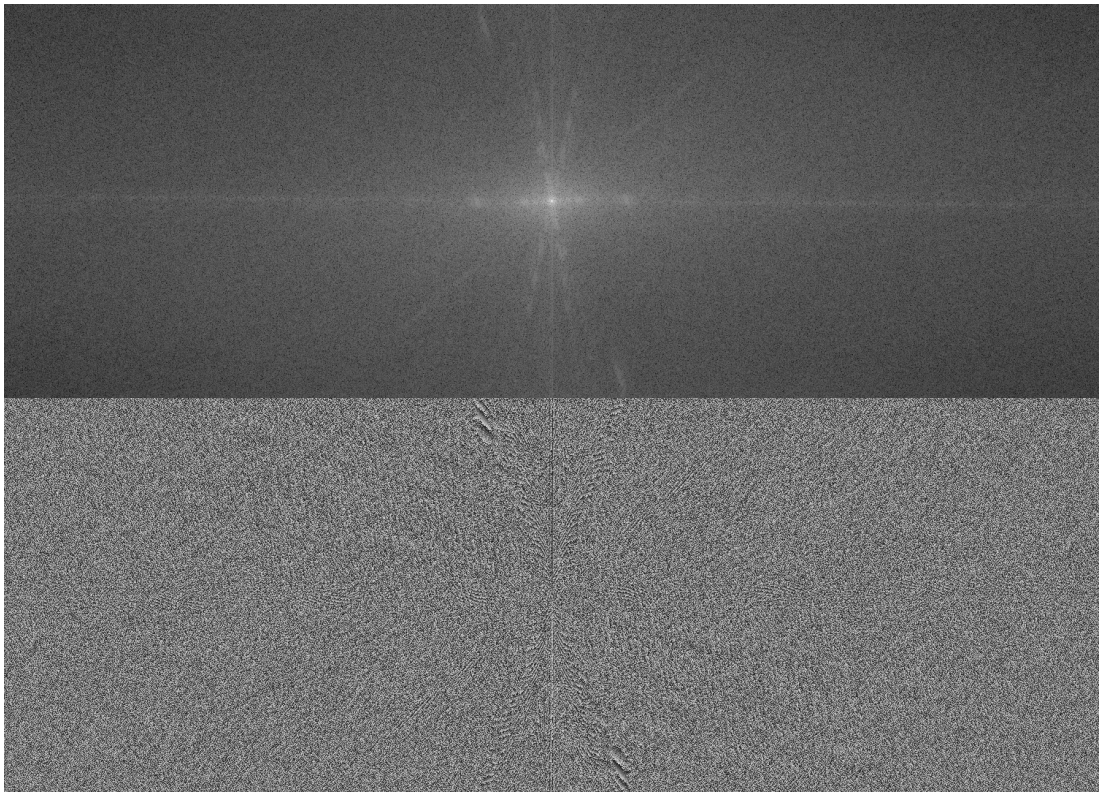


Figure 12: Log transformation on magnitude spectrum of greyscale `wolves.png`. Below, phase of spectrum of greyscale `wolves.png`

(b) *Use your `dtf2()` to implement the inverse FFT of an input transform `F`, `g = IDFT2(F)` from scratch.*

Since we know that

$$g = \text{swap}\left(\mathcal{F}\left\{\text{swap}\left(G\right)\right\}\right) = \mathcal{F}^{-1}\left\{G\right\} \tag{1}$$

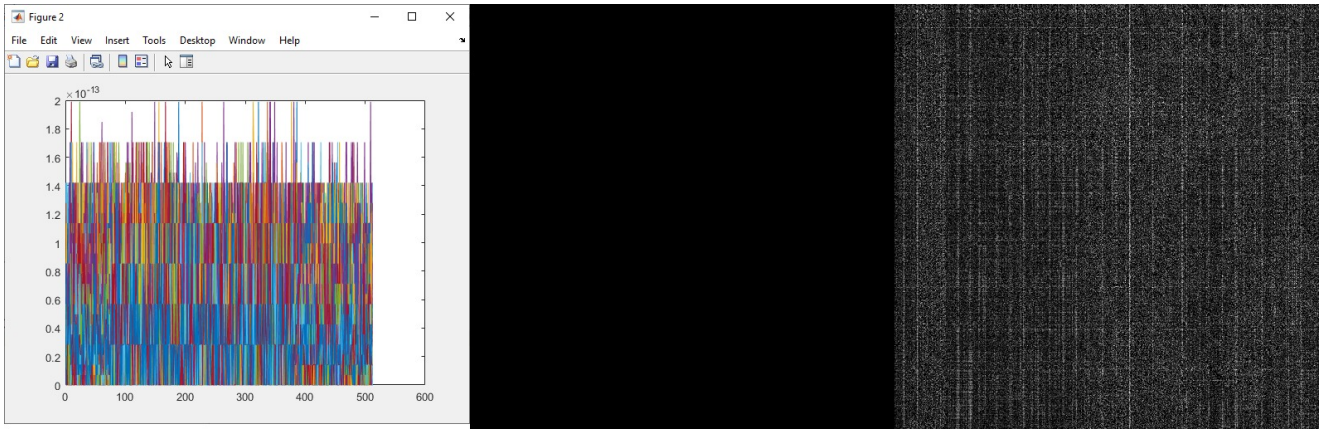
where

$$\text{swap}\left(a + ib\right) = b + ia$$

and we know that conjugation negates the imaginary component, so if we multiply by i and conjugate each data point, a 'swap' operation is performed. Knowing this, we can rewrite Equation 1 to have

$$g = 1i \times \overline{\mathcal{F}\left\{1i \times \overline{G}\right\}} = \mathcal{F}^{-1}\left\{G\right\} \tag{2}$$

This can very easily be incorporated into MATLAB. After scaling a greyscale `lena.tiff` to be between $[0,\ 1]$, the method in Equation 2 with my `dft2()` function is compared with MATLAB's built in `ifft2()` function.



Plot of difference Difference visualized (not scaled) Difference visualized (scaled)

Figure 13: Comparison of my `idft2()` function and MATLAB's `ifft2()` function. Plot of difference has peak magnitude of 2×10^{-13}, as seen in the graph

1 Appendix

1.1 MATLAB Testbench

All images and results are aggregated here

```
%% Project 1 - Arpad Voros
% pull lena and greyscale lena
lena = imread("lena.tiff");
lenabw = rgb2gray(lena);
[lenax, lenay] = size(lenabw);

% pull wolves and greyscale wolves
wolves = imread("wolves.png");
wolvesbw = rgb2gray(wolves);
[wolvesx, wolvesy] = size(wolvesbw);

%% box filter
box = ones(3);
figure(1);
res = imdisp(conv2me(lena, box, 4), "range", "none");
% imwrite(res, 'lena3_box.png');

%% simple derivative
% horizontal derivative/gradient
gradx = [-1 1];
% vertical derivative/gradient
grady = gradx';

% x, rgb lena
figure(1);
resx = imdisp(conv2me(lena, gradx, 4), "transform", "median_offset");
% imwrite(resx, 'lena_x.png');

% y, rgb lena
figure(2);
```

```
resy = imdisp(conv2me(lena, grady, 4), "transform", "median_offset");
% imwrite(resy, 'lena_y.png');

% simple mag, rgb lena
figure(3);
res_mag = sqrt(double(resx(1:lenax, 1:lenay, :)).^2 + double(resy(1:lenax, ...
    1:lenay, :)).^2);
res = imdisp(res_mag);
% imwrite(res, 'lena_simple_mag.png');

% x, greyscale lena
figure(4);
resx = imdisp(conv2me(lenabw, gradx, 4), "transform", "median_offset");
% imwrite(resx, 'lenabw_x.png');

% y, greyscale lena
figure(5);
resy = imdisp(conv2me(lenabw, grady, 4), "transform", "median_offset");
% imwrite(resy, 'lenabw_y.png');

% simple mag, greyscale lena
figure(6);
res_mag = sqrt(double(resx(1:lenax, 1:lenay, :)).^2 + double(resy(1:lenax, ...
    1:lenay, :)).^2);
res = imdisp(res_mag);
% imwrite(res, 'lenabw_simple_mag.png');

% % rgb wolves mag
% resx = imdisp(conv2me(wolves, gradx, 4), "transform", "median_offset");
% resy = imdisp(conv2me(wolves, grady, 4), "transform", "median_offset");
% res_mag = sqrt(double(resx(1:wolvesx, 1:wolvesy, :)).^2 + ...
    double(resy(1:wolvesx, 1:wolvesy, :)).^2);
% res = imdisp(res_mag);
% % imwrite(res, 'wolves_simple_mag.png');

% % greyscale wolves mag
% resx = imdisp(conv2me(wolvesbw, gradx, 4), "transform", "median_offset");
% resy = imdisp(conv2me(wolvesbw, grady, 4), "transform", "median_offset");
% res_mag = sqrt(double(resx(1:wolvesx, 1:wolvesy, :)).^2 + ...
    double(resy(1:wolvesx, 1:wolvesy, :)).^2);
% res = imdisp(res_mag);
% % imwrite(res, 'wolvesbw_simple_mag.png');


%% prewitt filter
% horizontal direction
prewittx = [-1 0 1; -1 0 1; -1 0 1];
% vertical direction
prewitty = flipud(prewittx');

% a good histogram distribution to match when displaying
alpha = 4;
pdf = exp(alpha * linspace(0, exp(1), 1000));

% prewittx, rgb lena
figure(1);
resx = imdisp(conv2me(lena, prewittx, 4), "transform", "median_offset", ...
    "histmatch", pdf);
% imwrite(resx, 'lena_prewittx.png');

% prewitty, rgb lena
figure(2);
```

```
88  resy = imdisp(conv2me(lena, prewitty, 4), "transform", "median_offset", ...
        "histmatch", pdf);
89  % imwrite(resy, 'lena_prewitty.png');
90
91  % prewitt mag, rgb lena
92  figure(3);
93  res_mag = sqrt(double(resx).^2 + double(resy).^2);
94  res = imdisp(res_mag);
95  % imwrite(res, 'lena_prewitt_mag.png');
96
97  % prewittx, greyscale lena
98  figure(4);
99  resx = imdisp(conv2me(lenabw, prewittx, 4), "transform", "median_offset", ...
        "histmatch", pdf);
100 % imwrite(resx, 'lenabw_prewittx.png');
101
102 % prewitty, greyscale lena
103 figure(5);
104 resy = imdisp(conv2me(lenabw, prewitty, 4), "transform", "median_offset", ...
        "histmatch", pdf);
105 % imwrite(resy, 'lenabw_prewitty.png');
106
107 % prewitt mag, greyscale lena
108 figure(6);
109 res_mag = sqrt(double(resx).^2 + double(resy).^2);
110 res = imdisp(res_mag);
111 % imwrite(res, 'lenabw_prewitt_mag.png');
112
113 % % rgb wolves mag
114 % resx = imdisp(conv2me(wolves, prewittx, 4), "transform", "median_offset", ...
        "histmatch", pdf);
115 % resy = imdisp(conv2me(wolves, prewitty, 4), "transform", "median_offset", ...
        "histmatch", pdf);
116 % res_mag = sqrt(double(resx).^2 + double(resy).^2);
117 % res = imdisp(res_mag);
118 % imwrite(res, 'wolves_prewitt_mag.png');
119 %
120 % % greyscale wolves mag
121 % resx = imdisp(conv2me(wolvesbw, prewittx, 4), "transform", "median_offset", ...
        "histmatch", pdf);
122 % resy = imdisp(conv2me(wolvesbw, prewitty, 4), "transform", "median_offset", ...
        "histmatch", pdf);
123 % res_mag = sqrt(double(resx).^2 + double(resy).^2);
124 % res = imdisp(res_mag);
125 % imwrite(res, 'wolvesbw_prewitt_mag.png');
126
127
128 %% sobel filter
129 % horizontal direction
130 sobelx = [-1 0 1; -2 0 2; -1 0 1];
131 % vertical direction
132 sobely = flipud(sobelx');
133
134 % a good histogram distribution to match when displaying
135 alpha = 4;
136 pdf = exp(alpha * linspace(0, exp(1), 1000));
137
138 % a good range to fit the final image
139 range = [20 255];
140
141 % sobelx, rgb lena
142 figure(1);
```

```
143 resx = imdisp(conv2me(lena, sobelx, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
144 % imwrite(resx, 'lena_sobelx.png');
145
146 % sobely, rgb lena
147 figure(2);
148 resy = imdisp(conv2me(lena, sobely, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
149 % imwrite(resy, 'lena_sobely.png');
150
151 % sobel mag, rgb lena
152 figure(3);
153 res_mag = sqrt(double(resx).^2 + double(resy).^2);
154 res = imdisp(res_mag);
155 % imwrite(res, 'lena_sobel_mag.png');
156
157 % sobelx, greyscale lena
158 figure(4);
159 resx = imdisp(conv2me(lenabw, sobelx, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
160 % imwrite(resx, 'lenabw_sobelx.png');
161
162 % sobely, greyscale lena
163 figure(5);
164 resy = imdisp(conv2me(lenabw, sobely, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
165 % imwrite(resy, 'lenabw_sobely.png');
166
167 % sobel mag, greyscale lena
168 figure(6);
169 res_mag = sqrt(double(resx).^2 + double(resy).^2);
170 res = imdisp(res_mag);
171 % imwrite(res, 'lenabw_sobel_mag.png');
172
173 % % rgb wolves mag
174 % resx = imdisp(conv2me(wolves, sobelx, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
175 % resy = imdisp(conv2me(wolves, sobely, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
176 % res_mag = sqrt(double(resx).^2 + double(resy).^2);
177 % res = imdisp(res_mag);
178 % imwrite(res, 'wolves_sobel_mag.png');
179 %
180 % % greyscale wolves mag
181 % resx = imdisp(conv2me(wolvesbw, sobelx, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
182 % resy = imdisp(conv2me(wolvesbw, sobely, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
183 % res_mag = sqrt(double(resx).^2 + double(resy).^2);
184 % res = imdisp(res_mag);
185 % imwrite(res, 'wolvesbw_sobel_mag.png');
186
187
188 %% roberts filter
189 % horizontal direction
190 robertsx = [0 1; -1 0];
191 % vertical direction
192 robertsy = flipud(robertsx');
193
194 % a good histogram distribution to match when displaying
195 alpha = 7;
196 pdf = exp(alpha * linspace(0, exp(1), 1000));
```

```
197
198 % a good range to fit the final image
199 range = [10 220];
200
201 % robertsx, rgb lena
202 figure(1);
203 resx = imdisp(conv2me(lena, robertsx, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
204 % imwrite(resx, 'lena_robertsx.png');
205
206 % robertsy, rgb lena
207 figure(2);
208 resy = imdisp(conv2me(lena, robertsy, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
209 % imwrite(resy, 'lena_robertsy.png');
210
211 % roberts mag, rgb lena
212 figure(3);
213 res_mag = sqrt(double(resx).^2 + double(resy).^2);
214 res = imdisp(res_mag);
215 % imwrite(res, 'lena_roberts_mag.png');
216
217 % robertsx, greyscale lena
218 figure(4);
219 resx = imdisp(conv2me(lenabw, robertsx, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
220 % imwrite(resx, 'lenabw_robertsx.png');
221
222 % robertsy, greyscale lena
223 figure(5);
224 resy = imdisp(conv2me(lenabw, robertsy, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
225 % imwrite(resy, 'lenabw_robertsy.png');
226
227 % roberts mag, greyscale lena
228 figure(6);
229 res_mag = sqrt(double(resx).^2 + double(resy).^2);
230 res = imdisp(res_mag);
231 % imwrite(res, 'lenabw_roberts_mag.png');
232
233 % % rgb wolves mag
234 % resx = imdisp(conv2me(wolves, robertsx, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
235 % resy = imdisp(conv2me(wolves, robertsy, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
236 % res_mag = sqrt(double(resx).^2 + double(resy).^2);
237 % res = imdisp(res_mag);
238 % imwrite(res, 'wolves_roberts_mag.png');
239 %
240 % % greyscale wolves mag
241 % resx = imdisp(conv2me(wolvesbw, robertsx, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
242 % resy = imdisp(conv2me(wolvesbw, robertsy, 4), "transform", "median_offset", ...
        "histmatch", pdf, "range", range);
243 % res_mag = sqrt(double(resx).^2 + double(resy).^2);
244 % res = imdisp(res_mag);
245 % imwrite(res, 'wolvesbw_roberts_mag.png');
246
247
248 %% unit impulse - lena kernel
249 lenabw_kernel = imresize(lenabw, 0.25);
250 img = zeros(1024);
```

```
251 img(512, 512) = 1;
252
253 res = imdisp(conv2me(img, lenabw_kernel, 4));
254 % imwrite(res, 'lena_unit.png');
255
256 %% 2D FFT
257
258 % rgb lena 2d fft mag
259 figure(1);
260 res = imdisp(dft2(lena, 1), "transform", "log");
261 % imwrite(res, 'lena_dft2.png');
262
263 % rgb lena 2d fft angle
264 figure(2);
265 res = imdisp(dft2(lena, 1), "disptype", "angle", "range", [20, 220]);
266 % imwrite(res, 'lena_dft2_angle.png');
267
268 % greyscale lena 2d fft mag
269 figure(3);
270 res = imdisp(dft2(lenabw, 1), "transform", "log");
271 % imwrite(res, 'lenabw_dft2.png');
272
273 % greyscale lena 2d fft angle
274 figure(4);
275 res = imdisp(dft2(lenabw, 1), "disptype", "angle", "range", [20, 220]);
276 % imwrite(res, 'lenabw_dft2_angle.png');
277
278 % rgb wolves 2d fft mag
279 figure(5);
280 res = imdisp(dft2(wolves, 1), "transform", "log");
281 % imwrite(res, 'wolves_dft2.png');
282
283 % rgb wolves 2d fft angle
284 figure(6);
285 res = imdisp(dft2(wolves, 1), "disptype", "angle", "range", [20, 220]);
286 % imwrite(res, 'wolves_dft2_angle.png');
287
288 % greyscale wolves 2d fft mag
289 figure(7);
290 res = imdisp(dft2(wolvesbw, 1), "transform", "log");
291 % imwrite(res, 'wolvesbw_dft2.png');
292
293 % greyscale wolves 2d fft angle
294 figure(8);
295 res = imdisp(dft2(wolvesbw, 1), "disptype", "angle", "range", [20, 220]);
296 % imwrite(res, 'wolvesbw_dft2_angle.png');
297
298 %% 2D FFT comparison
299
300 % scale lena 0 -> 1
301 lenabw_scaled = double(lenabw) ./ max(double(lenabw), [], 'all');
302
303 % 2D FFT using my function
304 my_dft = dft2(lenabw_scaled);
305 % 2D FFT using built-in MATLAB function
306 matlab_dft = fft2(lenabw_scaled);
307
308 % magnitude
309 figure(1);
310 plot(abs(my_dft - matlab_dft));
311 max_mag = max(abs(my_dft - matlab_dft), [], 'all');
312 fprintf("Maximum error in spectrum magnitude: %.20f\n", max_mag);
```

```
313
314 % phase
315 figure(2);
316 plot(abs(angle(my_dft) - angle(matlab_dft)));
317 max_ang = max(abs(angle(my_dft) - angle(matlab_dft)), [], 'all');
318 fprintf("Maximum error in spectrum phase: %.20f\n", max_ang);
319
320 %% 2D IFFT
321
322 % greyscale lena 2d ifft
323 F = dft2(lenabw, 1);
324
325 % lena displayable result
326 lenabw_transformed = 255 * double(lenabw) / max(double(lenabw), [], 'all');
327
328 % ifft displayable result
329 G = abs(idft2(F));
330 G = 255 * G / max(G, [], 'all');
331
332 % difference
333 diff = abs(double(lenabw_transformed) - G);
334
335 % display image of difference
336 figure(1);
337 diff_img = imdisp(uint8(diff), "range", "none");
338 % imwrite(diff_img, 'lena_idft2_uint8diff.png');
339
340 % plot of difference
341 figure(2);
342 plot(diff);
343
344 % scaled image of difference
345 figure(3);
346 diff_img = imdisp(diff);
347 % imwrite(diff_img, 'lena_idft2_diff.png');
```

1.2 conv2me.m

Could not name it `conv2()` since that is already a built-in MATLAB function

```
function g = conv2me(f, w, varargin)
% conv2me - ARPAD ATTILA VOROS
%           2D linear spatial filtering with various padding options
%   INPUTS:     f - input image, variable size (3 dimensional). 2D
%               convolution is applied to each channel independentally
%               w - kernel, 2D array
%               varargin{1} - pad (integer)
%               number representing type of padding (1, 2, 3, 4) where 1 is
%               zero-padding, 2 is wrap-around, 3 is copy-edge, and 4 is a
%               reflection
%   OUTPUTS:    g - resulting convolution of padded image and kernel
%                   (uint8)

% convert to double
f = double(f);
w = double(w);

% weight of the kernel
w_weight = 1 / numel(w);

% get kernel and pre-padded image dimensions
[w_h, w_w, w_c] = size(w);
if w_c > 1
    error("Kernel must be a 2-dimensional matrix.");
end

% get dims
[f_h, f_w, channels] = size(f);

% output image size depends on kernel size having center or not
% post cropping (assume cropping)
crop = 1;
out_h = f_h + ¬mod(w_h, 2);
out_w = f_w + ¬mod(w_w, 2);

% if there is padding, then pad the image accordingly
if ¬isempty(varargin)
    % input pad
    pad = varargin{1};
    if floor(pad) ≠ pad || pad < 1 || pad > 4
        error("Incorrect pad type selected, must be an integer 1 through 4");
    end

    % amount to pad
    h_pad = w_h - 1;
    w_pad = w_w - 1;

    % zero-padding (used for pre-initialization of all cases)
    f_post = zeros(f_h + (2 * h_pad), f_w + (2 * w_pad), channels);
    f_post((w_h:(w_h + f_h - 1)), (w_w:(w_w + f_w - 1)), :) = f;
    switch pad
        case 1
        case 2
            % wrap-around
            f_post(:, 1:w_pad, :) = f_post(:, (f_w + 1):(f_w + w_pad), :);
            f_post(:, (f_w + w_pad + 1):end, :) = f_post(:, (w_pad + 1):(2 * ...
                w_pad), :);
```

```
57              f_post(1:h_pad, :, :) = f_post((f_h + 1):(f_h + h_pad), :, :);
58              f_post((f_h + h_pad + 1):end, :, :) = f_post((h_pad + 1):(2 * ...
                    h_pad), :, :);
59          case 3
60              % copy-edge
61              f_post(:, 1:w_pad, :) = repmat(f_post(:, w_pad + 1, :), 1, w_pad);
62              f_post(:, (f_w + w_pad + 1):end, :) = repmat(f_post(:, f_w + ...
                    w_pad, :), 1, w_pad);
63              f_post(1:h_pad, :, :) = repmat(f_post(h_pad + 1, :, :), h_pad, 1);
64              f_post((f_h + h_pad + 1):end, :, :) = repmat(f_post(f_h + h_pad, ...
                    :, :), h_pad, 1);
65          case 4
66              % reflection
67              f_post(:, 1:w_pad, :) = fliplr(f_post(:, (w_pad + 1):(2 * w_pad), :));
68              f_post(:, (f_w + w_pad + 1):end, :) = fliplr(f_post(:, (f_w + ...
                    1):(f_w + w_pad), :));
69              f_post(1:h_pad, :, :) = flipud(f_post((h_pad + 1):(2 * h_pad), :, :));
70              f_post((f_h + h_pad + 1):end, :, :) = flipud(f_post((f_h + 1):(f_h ...
                    + h_pad), :, :));
71          otherwise
72              f_post = f;
73      end
74      % get post padded image dimensions
75      [f_h, f_w, ¬] = size(f_post);
76  else
77      crop = 0;
78      f_post = f;
79  end
80
81  % uncropped height and width
82  uncrop_h = f_h - w_h + 1;
83  uncrop_w = f_w - w_w + 1;
84  % initialize result array
85  g = zeros(uncrop_h, uncrop_w, channels);
86  % do convolution
87  for l = 1:channels
88      for r = 1:(f_h - w_h + 1)
89          for c = 1:(f_w - w_w + 1)
90              g(r, c, l) = sum(f_post(r:(r + w_h - 1), c:(c + w_w - 1), l) .* w, ...
                    'all');
91          end
92      end
93  end
94  % apply weight of the kernel
95  g = g * w_weight;
96
97  if crop
98      % calculate how to crop
99      h_space = round(0.5 * (uncrop_h - out_h));
100     w_space = round(0.5 * (uncrop_w - out_w));
101     crop_h_idx = (h_space + 1):(out_h + h_space);
102     crop_w_idx = (w_space + 1):(out_w + w_space);
103
104     % crop the image and return
105     g = g(crop_h_idx, crop_w_idx, :);
106 end
107
108 end
```

1.3 dft2.m

```
function Y = dft2(X, varargin)
% dft2 -    ARPAD ATTILA VOROS
%           Performs 2D fast-Fourier transform on input matrix X
%   INPUTS:     X - input matrix, 2D or 3D
%               if 3D, does 2D fft on each channel respectively
%               varargin{1} - fftshift enable
%               boolean/logical, disabled by default
%   OUTPUTS:    Y - resulting 2D FFT of X

% check X dims
if ndims(X) > 3
    error("Input must be a 2 or 3-dimensional matrice.");
end

% get size, initialize output
[X_r, X_c, X_channels] = size(X);
Y = zeros(X_r, X_c, X_channels);

% fftshift
if ¬isempty(varargin)
    fftshift_enable = varargin{1};
else
    fftshift_enable = 0;
end

% transform X to a double
% scale input 0 -> 1
X = double(X) ./ max(double(X), [], 'all');

% 1. column-wise fft on X
% 2. then perform non-conjugate transpose
% 3. another column-wise fft on transposed X (so it does the fft on the
% rows
% 4. non-conjugate transpose again to get orientation right
for channel = 1:X_channels
    Y(:, :, channel) = fft(fft(X(:, :, channel)).').';
end
% fftshift 1st and 2nd dimensions, if enabled
if fftshift_enable
    Y = fftshift(fftshift(Y, 1), 2);
end

end
```

1.4 idft2.m

```
function Y = idft2(X)
% idft2 -   ARPAD ATTILA VOROS
%           Performs 2D inverse fast-Fourier transform on input matrix X
%   INPUTS:     X - input matrix, 2D or 3D
%               if 3D, does 2D ifft on each channel respectively
%   OUTPUTS:    Y - resulting 2D IFFT of X

% check X dims
if ndims(X) > 3
    error("Input must be a 2 or 3-dimensional matrice.");
end

% get size, initialize output
[X_r, X_c, X_channels] = size(X);
Y = zeros(X_r, X_c, X_channels);

% calls dft2 for each channel of input X
for channel = 1:X_channels
    % uses swaps imaginary/real components before 2D fft, and swaps back
    % this performs a 2D ifft
    Y(:, :, channel) = 1j * conj(dft2(1j * conj(X(:, :, channel))));
end

end
```

1.5 imdisp.m

Used to display images with built-in preprocessing, so that a function call to `imdisp()` will properly display an image. `imshow()` has to constantly be formatted before it displays anything properly. Used extensively by the Project 1 Testbench

```
function varargout = imdisp(X, varargin)
% imdisp -  ARPAD ATTILA VOROS
%           Displays a 2-dimensional matrix 0-255 (unless specified
%           otherwise), or 3-dimensional matrice
%               If 3-dimensional, only the first 3 channels are displayed
%               as RGB. If lacking last channel (B), replaced with 0's
%   INPUTS:     X - input matrix, 2 or 3-dimensional
%               varargin -
%                   DispType -
%                       What to display
%                       If none specified, display raw. However if complex,
%                       defaults to magnitude
%                       1. "abs", "mag", 1
%                           displays magnitude
%                       2. "angle", "phase", 2
%                           displays phase
%                   Transform -
%                   TRANSFORMS APPLIED IN ORDER OF INPUT!
%                       0. "log", 1
%                           transforms output with natural log:
%                           log(1 + abs(result))
%                       1. "log10", 2
%                           transforms output with log base 10:
%                           log10(1 + abs(result))
%                       2. "histeq", 3
%                           histogram equalization
%                       4. "histmatch", 4
%                           histogram matching with reference array
%                           - input any-dimensional array afterward
%                               e.g. (..."Transform", "histmatch", ref...)
%                               where ref is any-d reference array
%                       5. "median_offset", 5
%                           transforms output with median offset
%                           sqrt((result - median(result)).^2)
%                       6. "mean_offset", 6
%                           transforms output with mean offset
%                           sqrt((result - mean(result)).^2)
%                       7. "mode_offset", 7
%                           transforms output with mode offset
%                           sqrt((result - mode(result)).^2)
%               DispSetting -
%                       1. "inv", "invert", 1
%               Range - [min, max], where min and max are converted to
%                       uint8. if none specified AND no histeq/histmatch,
%                       then defaults to [0, 255]
%               RGB -   if 2-dimensional, displays result on specified
%                       channel
%                       if 3-dimensional, displays ONLY selected channel
%                       1. "r", 1
%                           red channel
%                       2. "g", 2
%                           green channel
%                       3. "b", 3
%                           blue channel
```

```
55 %   OUTPUTS:    Y - output image, uint8
56 %
57 %   An example function call:
58 %       output = imdisp(input, "DispType", "mag", "DispSetting", "inv", ...
59 %                       "Transform", "log", "histmatch", histmatch_ref, ...
60 %                       "median_offset");
61 %   An EQUIVALENT call to the one above
62 %       output = imdisp(input, "DispSetting", 1, "disptype", "abs", ...
63 %                       "tRANSFORM", 1, "histmatch", histmatch_ref, ...
64 %                       5);
65
66 % check dimensions
67 if ndims(X) < 2 || ndims(X) > 3
68     error("Input must be a 2 or 3-dimensional matrice.");
69 end
70
71 % varargin categories
72 var_cat = [ "DispType", ...
73             "Transform", ...
74             "DispSetting", ...
75             "Range", ...
76             "RGB" ...
77             ];
78 % varargin varargin subcategories
79 var_subcat = {  {{"mag", "abs", 1}, {"angle", "phase", 2}}, ...
80                 {{"log", 1}, {"log10", 2}, ...
81                     {"histeq", 3}, {"histmatch", 4} ...
82                     {"median_offset", 5}, {"mean_offset", 6}, {"mode_offset", ...
                         7}}, ...
83                 {{"inv", "invert", 1}}, ...
84                 {{}, {"none"}}, ...
85                 {{"r", 1}, {"g", 2}, {"b", 3}} ...
86                 };
87 % default values if none specified
88 var_defaults = {{"none"}, ...
89                 {"none"}, ...
90                 {"none"}, ...
91                 {[0 255]}, ... % {[0 255]} {"none"}
92                 {"none"} ...
93                 }; %#ok<*STRSCALR>
94 % number of inputs (including subcategory) after category
95 % 1 unless specified otherwise
96 var_num_additional_input = {{[2, 4], 1}};
97 % mutually-exclusive display setting
98 var_uniq = logical([1, 0, 0, 1, 1]);
99
100 % parse varargin, get struct
101 [varg, vargord] = parsevarargin(varargin, var_cat, var_subcat, var_defaults, ...
        var_num_additional_input, var_uniq);
102
103 % clip Range 0 -> 255 if OOB
104 specified_range = false(1);
105 if numel(varg.Range) == 2
106     varg.Range = double(uint8(varg.Range));
107     specified_range = true(1);
108 end
109
110 % size of input
111 [xh, xw, xc] = size(X);
112 if ¬isreal(X)
113     % complex values, display magnitude if no DispType specified
114     if ¬varg.DispType
```

```
115         varg.DispType = true(1);
116         varg.mag = true(1);
117     end
118 end
119 
120 % turn input into double for future operations
121 X = double(X);
122 % initialize output
123 Y = uint8(zeros(xh, xw, xc));
124 % loop through each channel
125 for c = 1:xc
126     if varg.DispType
127         if varg.mag
128             x = abs(X(:, :, c));
129         elseif varg.angle
130             x = angle(X(:, :, c));
131         else
132             x = X(:, :, c);
133         end
134     else
135         x = X(:, :, c);
136     end
137     % preprocessing display options
138     if varg.Transform
139         for transform = vargord.Transform
140             switch transform
141                 case "log"
142                     x = log(1 + abs(x));
143                 case "log10"
144                     x = log10(1 + abs(x));
145                 case "histeq"
146                     x = histeq(x ./ max(x, [], 'all'));
147                 case "histmatch"
148                     x = imhistmatch(x ./ max(x, [], 'all'), varg.histmatch ./ ...
                            max(varg.histmatch, [], 'all'));
149                 case "median_offset"
150                     x = sqrt((x - median(x, 'all')).^2);
151                 case "mean_offset"
152                     x = sqrt((x - mean(x, 'all')).^2);
153                 case "mode_offset"
154                     x = sqrt((x - mode(x, 'all')).^2);
155                 otherwise
156             end
157         end
158     end
159     % if range is specified
160     if specified_range
161         x = x - min(x, [], 'all');
162         Y(:, :, c) = uint8(((varg.Range(2) - varg.Range(1)) * x / max(x, [], ...
                'all')) + varg.Range(1));
163     else
164         Y(:, :, c) = uint8(x);
165     end
166 end
167 % display the image
168 if varg.inv
169     Y = 255 - Y;
170 end
171 if varg.RGB
172     % 3-dimensional
173     if xc > 1
174         switch vargord.RGB
```

```
175             case "r"
176                 Y(:, :, 2:3) = 0;
177             case "g"
178                 Y(:, :, 1:2:3) = 0;
179             case "b"
180                 Y(:, :, 1:2) = 0;
181             otherwise
182         end
183     % 2-dimensional
184     else
185         switch vargord.RGB
186             case "r"
187                 Y = cat(3, Y, zeros(xh, xw, 2));
188             case "g"
189                 Y = cat(3, zeros(xh, xw, 1), Y, zeros(xh, xw, 1));
190             case "b"
191                 Y = cat(3, zeros(xh, xw, 2), Y);
192             otherwise
193         end
194     end
195 end
196 imshow(Y);
197 varargout{1} = Y;
198 end
```

1.6 parsevarargin.m

Used by **imdisp**

```
function [val_struct, varargout] = parsevarargin(varargin_input, categories, ...
    subcategories, defaults, num_additional_input_arr, uniq)
% parsevarargin -   ARPAD ATTILA VOROS
%   INPUTS:     categories - string array of main categories
%               subcategories - 2-3x nested cell array of potential
%                               subcategories for each category
%                               1st dim - cells for each category
%                               2nd dim - within each subcat, cells for
%                                         each equivalent option
%                               3rd dim - optional, ordered input
%                   defaults - default values for varargin. cell array size
%                              categories. each category has default value.
%                              "none" string indicates falsehood
%                   num_additional_input_arr - number of additional inputs
%                                              for a specified subcategory
%                   uniq - categories which have mutually exclusive
%                          subcategories
%   OUTPUTS:    parsed - parsed input into struct format
%
% EXAMPLE OF INPUT PARAEMTER FORMAT
% % varargin categories
% categories = ["DispType", ...
%               "Transform", ...
%               "DispSetting", ...
%               "Range", ...
%               "RGB" ...
%               ];
% % varargin subcategories
% subcategories = { {{"mag", "abs", 1}, {"angle", "phase", 2}}, ...
%                   {{"log", 1}, {"log10", 2}, ...
%                    {"histeq", 3}, {"histmatch", 4} ...
%                    {"median_offset", 5}, {"mean_offset", 6}, {"mode_offset", ...
    7}}, ...
%                   {{"inv", "invert", 1}}, ...
%                   {{}}, ...
%                   {{"r", 1}, {"g", 2}, {"b", 3}} ...
%                   };
% % default values if none specified
% defaults = {  {"none"}, ...
%               {"none"}, ...
%               {"none"}, ...
%               {[0 255]}, ...
%               {"none"} ...
%               }; %#ok<*STRSCALR>
% % number of inputs (including subcategory) after category
% % 1 unless specified otherwise
% num_additional_input_arr = {{[2, 4], 1}};
% % mutually-exclusive display setting
% uniq = logical([1, 0, 0, 1, 1]);

% parsing varargin for display parameters
% number of categories
num_categories = length(categories);
% number irregular input sizes
idx_numinput_len = length(num_additional_input_arr);
% number of total possible fields (include main categories incase
% non qualitative response)
```

```
56  field_count = num_categories;
57  % qualitative input, i.e. not numeric
58  qualidx = 1:num_categories;
59  for cat_idx = 1:num_categories
60      % categories without subcategories, i.e. quantitative
61      if isempty(subcategories{cat_idx}{1}) % numel(subcategories{cat_idx}) == 1 &&
62          % if quantitative, remove from the list
63          qualidx(qualidx == cat_idx) = [];
64      else
65          % add all subcategories to field count
66          field_count = field_count + numel(subcategories{cat_idx});
67      end
68  end
69
70  % START FSM
71  state = 0;
72  prev_state = 0;
73  vidx = 0;
74  arg = 0;
75  subcat_idx = 0;
76  num_additional_inputs = 0;
77
78  % varargin length
79  varargin_length = length(varargin_input);
80  % for creating varargin struct
81  varargin_field_index = 1;
82  varargin_field_id = string(cellfun(@(x) "", cell(1, field_count), ...
        'UniformOutput', false));
83  % populate field ids
84  for cat_idx = 1:num_categories
85      varargin_field_id(varargin_field_index) = categories(cat_idx);
86      varargin_field_index = varargin_field_index + 1;
87      for subcat_idx = 1:numel(subcategories{cat_idx})
88          % if there are subcategories, append to field id
89          if ¬isempty(subcategories{cat_idx}{subcat_idx})
90              % field id is first option
91              varargin_field_id(varargin_field_index) = ...
                    lower(string(subcategories{cat_idx}{subcat_idx}{1}));
92              varargin_field_index = varargin_field_index + 1;
93          elseif subcat_idx == 1
94              % first index is empty, meaning QUANTITATIVE or NONE so leave
95              break;
96          end
97      end
98  end
99  % initialize field values
100 varargin_field_value = cell(1, field_count);
101 % initialize subcat field order
102 varargin_subcat_field_order = zeros(1, field_count);
103 % FSM, parsing varargin like regex
104 while vidx < varargin_length + 1
105     switch state
106         % checking for category
107         case 0
108             if prev_state == 0
109                 vidx = vidx + 1;
110                 if vidx > varargin_length
111                     break;
112                 end
113                 % get new input
114                 try
115                     arg = lower(varargin_input{vidx});
```

```
116                 catch
117                     arg = varargin_input{vidx};
118                 end
119                 if ismember(arg, lower(categories))
120                     % update category
121                     cat_idx = arg == lower(categories);
122                     % move states
123                     prev_state = state;
124                     state = 1;
125                 else
126                     error("Error parsing display settings: check spelling.");
127                 end
128             elseif ismember(arg, lower(categories))
129                 % update category
130                 cat_idx = arg == lower(categories);
131                 % move states
132                 prev_state = state;
133                 state = 1;
134             else
135                 error("Error parsing display settings: check spelling.");
136             end
137         % checking new instance of subcategory
138         case 1
139             vidx = vidx + 1;
140             if vidx > varargin_length
141                 break;
142             end
143             % get new input
144             try
145                 arg = lower(varargin_input{vidx});
146             catch
147                 arg = varargin_input{vidx};
148             end
149             % accumulate all subcategory possibilities
150             subcat_possibilities = cellfun(@(x) x(:), ...
                    subcategories{cat_idx}(:), 'UniformOutput', false);
151             % find if empty subcategories allowed - i.e. ANY input allowed,
152             % so quantitative possibility
153             subcat_empties = false(1);
154             for subcat_poss_idx = 1:numel(subcat_possibilities)
155                 if isempty(subcat_possibilities{subcat_poss_idx})
156                     subcat_empties = true(1);
157                     break;
158                 end
159             end
160             subcat_possibilities = vertcat(subcat_possibilities{:});
161             % input argument is category
162             if isstring(arg) && ismember(arg, lower(categories))
163                 prev_state = state;
164                 state = 0;
165             % input argument is a potential subcategory
166             elseif isstring(arg) && ismember(lower(arg), ...
                    string(subcat_possibilities)) % any(cellfun(@(x) ...
                    isequal(lower(x), arg), subcat_possibilities))
167                 % quantitative possibility, but equals other subcategory
168                 % possibilities. if none, set false, otherwise add the
169                 % subcategory to the lists (previously ignored)
170                 if subcat_empties && isequal(lower(arg), "none")
171                     varargin_field_value{varargin_field_id == ...
                            categories(cat_idx)} = false(1);
172                     prev_state = state;
173                     state = 1;
```

```
174                 else
175                     if subcat_empties
176                         % quantitative OR some other thing which was
177                         % previously ignored, now appended
178                         field_count = field_count + 1;
179                         varargin_field_id(field_count) = lower(arg);
180                         varargin_field_value{field_count} = [];
181                     end
182                     % if category is empty, set it to true
183                     if isempty(varargin_field_value{varargin_field_id == ...
                            categories(cat_idx)})
184                         varargin_field_value{varargin_field_id == ...
                                categories(cat_idx)} = true(1);
185                     end
186                     % find subcategory
187                     subcat_idx = 1;
188                     for subcat_parse_idx = 1:numel(subcategories{cat_idx})
189                         if max(cellfun(@(x) isequal(lower(x), arg), ...
                                subcategories{cat_idx}{subcat_parse_idx}))
190                             subcat_idx = subcat_parse_idx;
191                         end
192                     end
193                     % update subcat field order
194                     varargin_subcat_field_order(varargin_field_id == ...
                            subcategories{cat_idx}{subcat_idx}{1}) = vidx;
195                     % find the number of additional inputs given the subcat
196                     num_additional_inputs = 0;
197                     for numinput_idx = 1:idx_numinput_len
198                         if find(cat_idx) == ...
                                num_additional_input_arr{numinput_idx}{1}(1) && ...
                                subcat_idx == ...
                                num_additional_input_arr{numinput_idx}{1}(2)
199                             num_additional_inputs = ...
                                    num_additional_input_arr{numinput_idx}{2};
200                         end
201                     end
202                     if num_additional_inputs > 0
203                         % qualitative found, next is quantitative
204                         vidx = vidx + 1;
205                         if vidx > varargin_length
206                             break;
207                         end
208                         % get new input
209                         try
210                             arg = lower(varargin_input{vidx});
211                         catch
212                             arg = varargin_input{vidx};
213                         end
214                         prev_state = state;
215                         state = 2;
216                     else
217                         % qualitative
218                         varargin_field_value{varargin_field_id == ...
                                subcategories{cat_idx}{subcat_idx}{1}} = true(1);
219                         if uniq(cat_idx)
220                             prev_state = state;
221                             state = 3;
222                         else
223                             prev_state = state;
224                             state = 1;
225                         end
226                     end
```

```
227                 end
228             % no subcategory + quantitative due to empty possibilities
229             elseif subcat_empties
230                 % quantitative, therefore no qualitative subcategory
231                 subcat_idx = 1;
232                 % find the number of additional inputs
233                 num_additional_inputs = 1;
234                 for numinput_idx = 1:idx_numinput_len
235                     if find(cat_idx) == ...
                            num_additional_input_arr{numinput_idx}{1}(1) && ...
                            subcat_idx == num_additional_input_arr{numinput_idx}{1}(2)
236                         num_additional_inputs = ...
                                num_additional_input_arr{numinput_idx}{2};
237                     end
238                 end
239                 % move states
240                 prev_state = state;
241                 state = 2;
242             else
243                 error("Error parsing display settings: check formatting.");
244             end
245         % looping through fixed number of additional inputs
246         case 2
247             % first entered looping of additional inputs
248             if prev_state == 1
249                 if numel(subcategories{cat_idx}{subcat_idx})
250                     varargin_field_value{varargin_field_id == ...
                            subcategories{cat_idx}{subcat_idx}{1}} = ...
251                         [varargin_field_value{varargin_field_id == ...
                                subcategories{cat_idx}{subcat_idx}{1}}, {arg}];
252                 else
253                     varargin_field_value{varargin_field_id == ...
                            categories(cat_idx)} = ...
254                         [varargin_field_value{varargin_field_id == ...
                                categories(cat_idx)}, {arg}];
255                 end
256
257                 num_additional_inputs = num_additional_inputs - 1;
258                 % change state
259                 if num_additional_inputs == 0
260                     prev_state = state;
261                     state = 1;
262                 else
263                     prev_state = state;
264                     state = 2;
265                 end
266             % has been looping of additional inputs
267             elseif prev_state == 2
268                 vidx = vidx + 1;
269                 if vidx > varargin_length
270                     break;
271                 end
272                 % get new input
273                 try
274                     arg = lower(varargin_input{vidx});
275                 catch
276                     arg = varargin_input{vidx};
277                 end
278                 % append argument to output
279                 varargin_field_value{varargin_field_id == ...
                        subcategories{cat_idx}{subcat_idx}{1}} = ...
280                     [varargin_field_value{varargin_field_id == ...
```

```
                        subcategories{cat_idx}{subcat_idx}{1}}, {arg}];
281                 % update subcategory field order, if not already
282                 if varargin_subcat_field_order(varargin_field_id == ...
                        subcategories{cat_idx}{subcat_idx}{1}) == 0
283                     varargin_subcat_field_order(varargin_field_id == ...
                            subcategories{cat_idx}{subcat_idx}{1}) = vidx;
284                 end
285                 % update number of additional inputs
286                 num_additional_inputs = num_additional_inputs - 1;
287                 % change state
288                 if num_additional_inputs == 0
289                     prev_state = state;
290                     state = 1;
291                 else
292                     prev_state = state;
293                     state = 2;
294                 end
295             else
296                 error("Illegal FSM configuration.");
297             end
298         % if unique subcategory, set all other subcategories of same
299         % category to false
300         case 3
301             for subcat_parse_idx = 1:numel(subcategories{cat_idx})
302                 if subcat_parse_idx ≠ subcat_idx
303                     if ¬isempty(subcategories{cat_idx}{subcat_parse_idx}) && ...
                            any(varargin_field_id == ...
                            subcategories{cat_idx}{subcat_parse_idx}{1})
304                         varargin_field_value{varargin_field_id == ...
                                subcategories{cat_idx}{subcat_parse_idx}{1}} = [];
305                         varargin_subcat_field_order(varargin_field_id == ...
                                subcategories{cat_idx}{subcat_parse_idx}{1}) = 0;
306                     end
307                 end
308             end
309             prev_state = state;
310             state = 1;
311         otherwise
312             state = 0;
313     end
314 end
315 % transform subcat field order properly
316 ord_cell = cell(num_categories, 1);
317 vidx = 1;
318 for cat_idx = 1:num_categories
319     num_subcat = numel(subcategories{cat_idx});
320     if isempty(subcategories{cat_idx}{1}) % num_subcat == 1 &&
321         num_subcat = 0;
322     end
323     idx_range = vidx + (0:num_subcat);
324     vec = varargin_subcat_field_order(idx_range);
325     if length(idx_range) > 1
326         ord_mask = [false(1), vec(2:end) > 0];
327         [¬, ord_idx] = sort(vec(ord_mask));
328         if ¬isempty(ord_idx)
329             ord_cell{cat_idx} = varargin_field_id(idx_range(ord_mask));
330             ord_cell{cat_idx} = ord_cell{cat_idx}(ord_idx);
331         end
332     else
333         ord_cell{cat_idx} = [];
334     end
335     vidx = idx_range(end) + 1;
```

```
336 end
337 % if empty, use default
338 cat_idx = 0;
339 for varargin_field_index = 1:length(varargin_field_id)
340     % find the indicies
341     if ismember(varargin_field_id(varargin_field_index), categories)
342         cat_idx = cat_idx + 1;
343         subcat_idx = 0;
344         def_vals = defaults{cat_idx};
345     else
346         subcat_idx = subcat_idx + 1;
347     end
348     % if no value used, append defaults to field values
349     if isempty(varargin_field_value{varargin_field_index})
350         if ¬subcat_idx
351             % a category that is empty
352             if length(def_vals) == 1 && isequal(def_vals{1}, "none")
353                 % whole category is FALSE if default is 1x1 "none"
354                 varargin_field_value{varargin_field_index} = false(1);
355             end
356             % if STILL EMPTY, then there is a value to append
357             if isempty(varargin_field_value{varargin_field_index})
358                 % difference between quantitative and qualitative
359                 if isempty(subcategories{cat_idx}{1}) % ...
                        numel(subcategories{cat_idx}) == 1 &&
360                     % if qualitative & NO subcategory values, set category
361                     % field value to false. otherwise, quantitative, so
362                     % must have DEFAULT DATA
363                     if ismember(cat_idx, qualidx)
364                         % set value to false since no subcategories
365                         varargin_field_value{varargin_field_index} = false(1);
366                     else
367                         % set value to default value since no subcategories
368                         varargin_field_value{varargin_field_index} = ...
                                defaults{cat_idx}{1};
369                     end
370                 else
371                     % set value to true if there are default subcategories
372                     varargin_field_value{varargin_field_index} = true(1);
373                 end
374             end
375         else
376             % set field values accordingly
377             % overwrite default value array, depending on whether unique
378             % or not. use first index if unique
379             if uniq(cat_idx)
380                 def_vals_overwrite = def_vals(1);
381             else
382                 def_vals_overwrite = def_vals;
383             end
384             % loop through default values
385             for def_vals_idx = 1:length(def_vals_overwrite)
386                 % if qualitative, ALL struct values must be boolean
387                 if ismember(cat_idx, qualidx)
388                     % if equal to the index of the subcategory, then set
389                     % field value to true and ALL OTHERS to false
390                     if isequal(def_vals{def_vals_idx}, ...
                            varargin_field_id(varargin_field_index))
391                         varargin_field_value{varargin_field_index} = true(1);
392                     else
393                         varargin_field_value{varargin_field_index} = false(1);
394                     end
```

```
395                 % if quantitative, chosen struct value must be DATA, others
396                 % default to false
397                 else
398                     % if equal to the index of the subcategory, then set
399                     % field value to default and ALL OTHERS to false
400                     if isequal(def_vals{def_vals_idx}, ...
                            varargin_field_id(varargin_field_index))
401                         varargin_field_value{varargin_field_index} = ...
                                defaults{cat_idx}{subcat_idx};
402                     else
403                         varargin_field_value{varargin_field_index} = false(1);
404                     end
405                 end
406             end
407         end
408     elseif iscell(varargin_field_value{varargin_field_index}) && ...
            numel(varargin_field_value{varargin_field_index}) == 1
409         varargin_field_value{varargin_field_index} = ...
                varargin_field_value{varargin_field_index}{1};
410     end
411 end
412 % create struct for parsing
413 struct_labels = cellstr(varargin_field_id);
414 varargin_field_value = varargin_field_value(:);
415 val_struct = cell2struct(varargin_field_value, struct_labels, 1);
416 ord_struct = cell2struct(ord_cell, cellstr(categories), 1);
417 varargout{1} = ord_struct;
418 end
```