Name: Alexander Swerdlow

1: Image Processing

1.1

$$\begin{aligned}\exp(j(\omega_1 n_1 + \omega_2 n_2)) &= \exp(j\omega_1(n_1 + \Delta_1) + j\omega_2(n_2 + \Delta_2)) \\ &= \exp(j\omega_1(n_1 + \Delta_1))\exp(j\omega_2(n_2 + \Delta_2)) \\ &= (\cos(\omega_1(n_1 + \Delta_1)) + j\sin(\omega_1(n_1 + \Delta_1)))(\cos(\omega_2(n_2 + \Delta_2)) + j\sin(\omega_2(n_2 + \Delta_2))) \\ &= (\cos(\omega_1 n_1 + \omega_1\Delta_1) + j\sin(\omega_1 n_1 + \omega_1\Delta_1))(\cos(\omega_2 n_2 + \omega_2\Delta_2) + j\sin(\omega_2 n_2 + \omega_2\Delta_2))\end{aligned}$$

We can use the known periodicity of $\cos()$ and $\sin()$ and let Δ_1, Δ_2 be any multiple of 2π. Therefore, the function is periodic in space for all $\omega_1, \omega_2 \in \mathbb{Z}$

1.2

i) $T[x] = y$

$X_1 = [1\ 0\ 0]$ $\quad T[x_1] = a$

$X_2 = [0\ 1\ 0]$ $\quad T[x_2] = b$

$X_3 = [0\ 0\ 1]$ $\quad T[x_3] = c$

$X_4 = [5\ 4\ 3]$ $\quad T[x_4] = ?$

Since the system is LSI we can model it as product of matrices using our basis vectors and we know y is 1x1 (a scalar)

$$T[x] = y = Ax \qquad A = \overset{1\times 3}{\begin{pmatrix} a & b & c \end{pmatrix}} \qquad \overset{3\times 1}{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}}$$

$$T[x_4] = \begin{pmatrix} a & b & c \end{pmatrix}\begin{pmatrix} 5 \\ 4 \\ 3 \end{pmatrix} = \boxed{5a + 4b + 3c}$$

ii) **No.** we need the output given all 3 basis vectors to cover all 3 dimensions of the system

iii) Create a system of equations: $dX_1 + eX_2 + fX_3 = X_4$

$$\begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & -1 \\ 0 & 0 & -1 \end{pmatrix}\begin{pmatrix} d \\ e \\ f \end{pmatrix} = \begin{pmatrix} 5 \\ 4 \\ 3 \end{pmatrix} \Rightarrow \begin{pmatrix} d \\ e \\ f \end{pmatrix} = \begin{pmatrix} 8 \\ 1 \\ -3 \end{pmatrix}$$

$$X_4 = T[dX_1 + eX_2 + fX_3]$$

$$= \boxed{8a + b - 3d}$$

1.3

i) $T_1[x(n_1)] = 2x(n_1)$

$y(n_1) = 2x(n_1)$ ← What we need to show

$T_1[x(n_1 - n_0)] = y(n_1 - n_0)$

$T_1[x(n_1 - n_0)] = 2x(n_1 - n_0)$ ✓

$y(n_1 - n_0) = 2x(n_1 - n_0)$

LHS = RHS So T_1 is Space invariant

ii) $T_2[x(n_1)] = x(2n_1)$

$y(n_1) = x(2n_1)$

What we need to show →

$T_2[x(n_1 - n_0)] = y(n_1 - n_0)$

$T_2[x(n_1 - n_0)] = x(2(n_1 - n_0) = x(2n_1 - 2n_0)$

$y(n_1 - n_0) = x(2n_1 - n_0)$

LHS ≠ RHS So T_2 is not Space invariant

1.4 Convolutions

i) $y[n] = (h * f)[n] = (f * h)[n] = \sum_{m=-\infty}^{\infty} f(m)h(n-m)$

$$y[n] = \begin{cases} 1 \cdot -1 & \text{if } n=0 \\ 2 \cdot -1 & \text{if } n=1 \\ 3 \cdot -1 + 1 \cdot 1 & \text{if } n=2 \\ 4 \cdot -1 + 2 \cdot 1 & \text{if } n=3 \\ 3 \cdot 1 & \text{if } n=4 \\ 4 \cdot 1 & \text{if } n=5 \end{cases} = \begin{cases} -1 & \text{if } n=0 \\ -2 & \text{if } n=1 \\ -2 & \text{if } n=2 \\ -2 & \text{if } n=3 \\ 3 & \text{if } n=4 \\ 4 & \text{if } n=5 \\ 0 & \text{else} \end{cases}$$

ii) $$y = h * x = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \end{pmatrix} = \begin{pmatrix} -1 & -2 & -2 & -2 & 3 & 4 \end{pmatrix}$$

2: 2D-Convolution

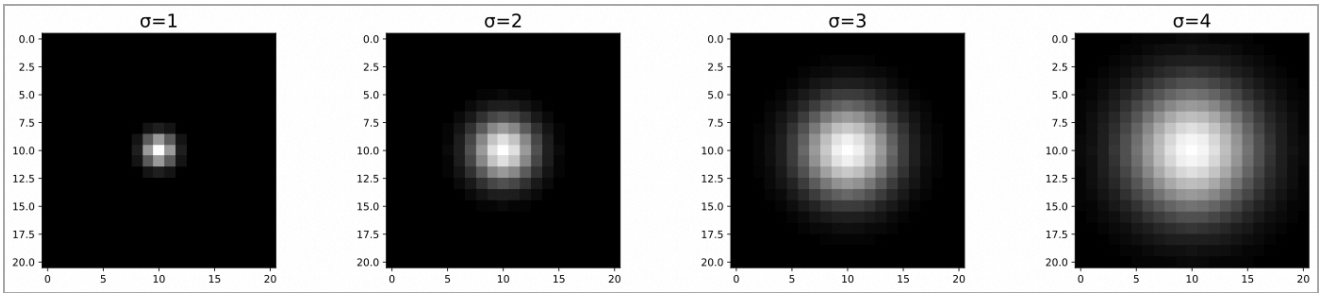
```
def conv2D(image: np.array, kernel: np.array = None):
    pad = kernel.shape[0] // 2
    padded_img = np.pad(image, ((pad,pad),(pad,pad)), 'constant')
    new_image = np.empty_like(image)
    for r in range(image.shape[0]):
        for c in range(image.shape[0]):
            new_image[r][c] = np.tensordot(padded_img[r:r +
kernel.shape[0], c:c + kernel.shape[0]], kernel)
    return new_image
```

3: Image Blurring and Denoising

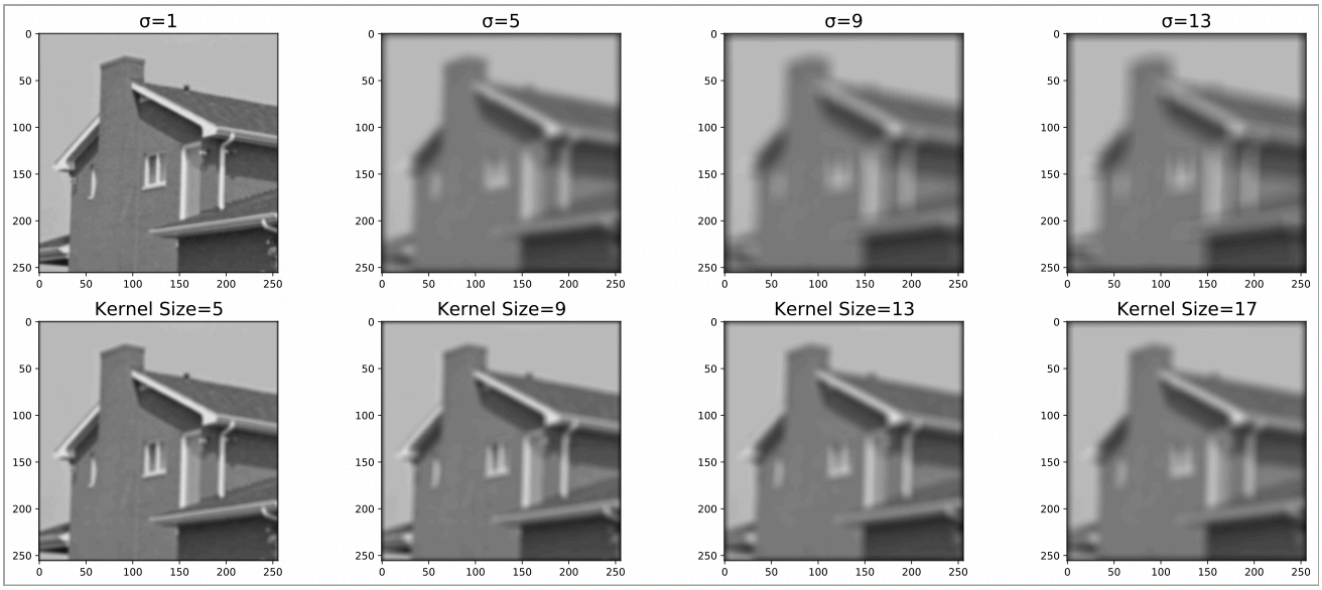
3.1

```
def gaussian_filter(size: int, sigma: float):
    filter = np.zeros((size, size))
    center = size//2
    for i in range(size):
        for j in range(size):
            filter[i][j] = (1/(2*np.pi*sigma**2)) * np.exp(-((j-
center)**2 + (i-center)**2)/(2 * sigma**2))
    return filter/filter.sum()
```

3.2



3.3



3.4

Increasing σ increases the Variance (Variance is σ^2) of the 2D Gaussian. This means values further from the center pixel [where "center pixel" is the center of the sliding window, i.e. the pixel whose new value we are calculating] are weighted more heavily. This can be seen in *3.2* where the white "circle" increases in size (lighter color -> higher value). Since increasing σ causes pixels further from the center to be weighted more heavily, we see that the photo becomes more blurred as edges and fine detail become washed out. Meanwhile, a small σ makes the Gaussian look more like a Dirac, causing little blur.

Increasing the kernel size causes a similar effect as a larger kernel means pixels further from the center are now taken into account during convolution. Increasing the kernel size will show less and less difference as the kernel size increases (since the 2D gaussian goes to 0 as the distance from the center goes to ∞). When we have a particularly small kernel size, we are chopping off parts of the gaussian with significant values that would blur the image but at a certain kernel size, the values are so small that it makes little visual difference.

However, the difference with larger kernel sizes can still be seen in the photos in *3.3*.

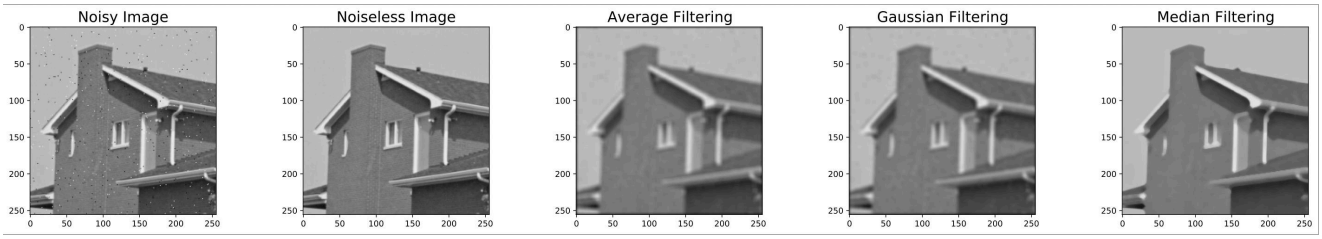
3.5

```
def median_filtering(image: np.array, kernel_size: int = None):
    def sanitize_bounds(edge: int):
        if edge < 0:
            return 0
        elif edge >= image.shape[0]:
            return image.shape[0] - 1
        else:
            return edge

    pad = kernel_size // 2
    new_image = np.empty_like(image)
    for r in range(image.shape[0]):
        for c in range(image.shape[0]):
```

```
            bottom_r = sanitize_bounds(r - pad)
            top_r = sanitize_bounds(r + pad + 1)
            left_c = sanitize_bounds(c - pad)
            right_c = sanitize_bounds(c + pad + 1)
            new_image[r][c] = np.median(image[bottom_r : top_r, left_c
: right_c])
    return new_image
```

3.6



3.7

Median filtering performed the best as we can see clearly from the relative absolute distance `rel_l1_dist` results where median filtering had a value of 0.033 with average and median filtering having values of 0.0652 and 0.0599 respectively.

Median filtering performs better as it removes the salt and pepper noise and preserves edges much better than Gaussian filtering (and certainly average filtering) does. It smoothes out the sky and the flat wall, filtering out the noise (extreme values). This is because gaussian filtering still takes into account extreme values, so an edge which transitions from $0 \rightarrow 1$ gets blurred because the gaussian only accounts for the spatial difference between the pixels. Median filtering, meanwhile, helps remove extreme values, so as long as the filter is slightly towards one side of the edge, the values on the other side are likely ignored. The same occurs in the sky portions of the image where a single extreme value will be ignored in a window of otherwise similar values. Average filtering performs the worst because it doesn't even take into account distance between the pixels and causes a uniform blur.

3.8

As mentioned in *3.7*, median filtering removes noise and preserves the edges better because it filters extreme values. Meaning that if the "sliding window" sees say, 10 white pixels but 4 fully black pixels, the black pixels will be totally ignored noise will be removed. This breaks down when the sliding window is exactly split (e.g. directly over the edge), but still performs much better than average/gaussian filtering and preserves edges better.

This does align with *3.7* as the main difference between the images is blurred edges and the smoother background.

4: Image Gradients

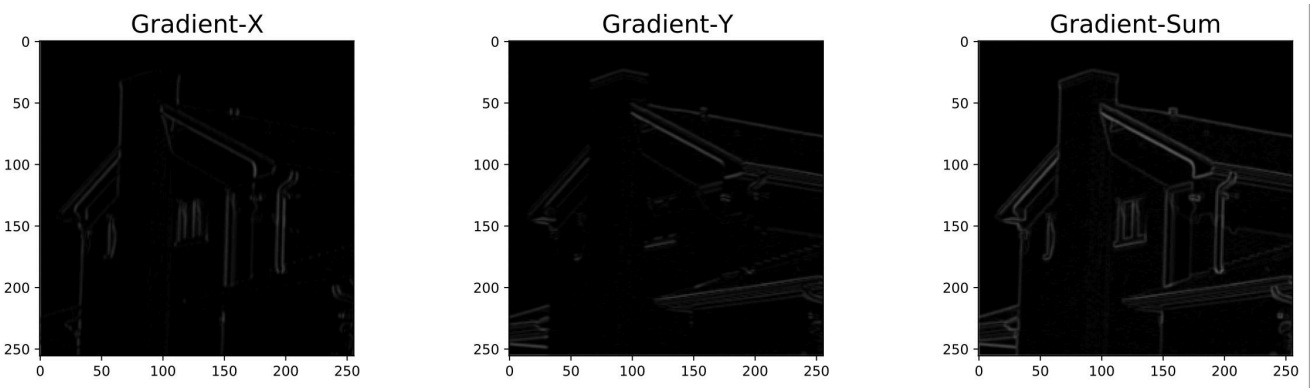
4.1

```
gradient_x = (1/6) * np.array([[-1, 0, 1],[-1, 0, 1],[-1, 0, 1]])
```

4.2

```
gradient_y = (1/6) * np.array([[-1, -1, -1],[0, 0, 0],[1, 1, 1]])
```

4.3



4.4

For any given pixel, (i, j), we can simply take $G_x(i, j)$ and $G_y(i, j)$ where G_x and G_y are the images after being convolved with `gradient_x` and `gradient_y` respectivley. The direction of the gradient is then:

$$\theta = tan^{-1}(G_y(i, j)/G_x(i, j))$$

4.5

Yes, the gradient filter is separable:

For the horizantal filter:

$$G_x = \frac{1}{6}\begin{bmatrix}1\\1\\1\end{bmatrix}\begin{bmatrix}1 & 0 & -1\end{bmatrix} = \frac{1}{6}\begin{pmatrix}1 & 0 & -1\\1 & 0 & -1\\1 & 0 & -1\end{pmatrix}$$

For the vertical filter:

$$G_y = \frac{1}{6}\begin{bmatrix}1 & 1 & 1\end{bmatrix}\begin{bmatrix}-1\\0\\1\end{bmatrix} = \frac{1}{6}\begin{pmatrix}-1 & -1 & -1\\0 & 0 & 0\\1 & 1 & 1\end{pmatrix}$$

5

5.1

Gaussian filtering only depends on the distances between the center pixel and the one being looked at. This means that Gaussian filter emphasizes closer pixels, but disregards intensity/color differences. This means that edges get blurred as pixels on the opposite side of an edge may still be close to the center pixel, just have very different intensities. In this case, that pixel may be highly weighted and result in the center pixel value being much closer to an average between the different sides of the edge (i.e. an edge that is a transition from $0 \rightarrow 1$ may result in a value closer to 0.5).

From a frequency domain perspective, the Gaussian filter only preserves low frequencies (similar to a lowpass filter), and it's frequency response goes to zero as the spatial frequency goes to ∞. Therefore, features of the image with high spatial frequency such as edges or noise are blurred by the Gaussian filter.

5.2

The distance from p to q, $||p-q||$ and the intensity difference, $||I_p - I_q||$ should determine the filter weight. Pixels closer in space are likely to represent similar features but we also need to account for the intensity of the pixels. Pixels close in space **and** close in value are almost certainly representing the same object/feature but if they have very different intensities, we may be observing an edge/noise.

5.3

$$GF[I_p] = \sum_{q \in S} G_\sigma\big(\|p-q\|\big) f\big(\|I_p -_I q\|\big) I_q.$$

Here, the function $f\big(I_p, I_q\big) = G_{\sigma_i}\big(\|I_p - I_q\|\big)$ where σ_i is modifies the variance of the 1D-Gaussian.

The function $G_{\sigma_i}\left(\|I_p - I_q\|\right)$ is monotonically decreasing (assuming our input is always $\|I_p - I_q\|$).

5.4

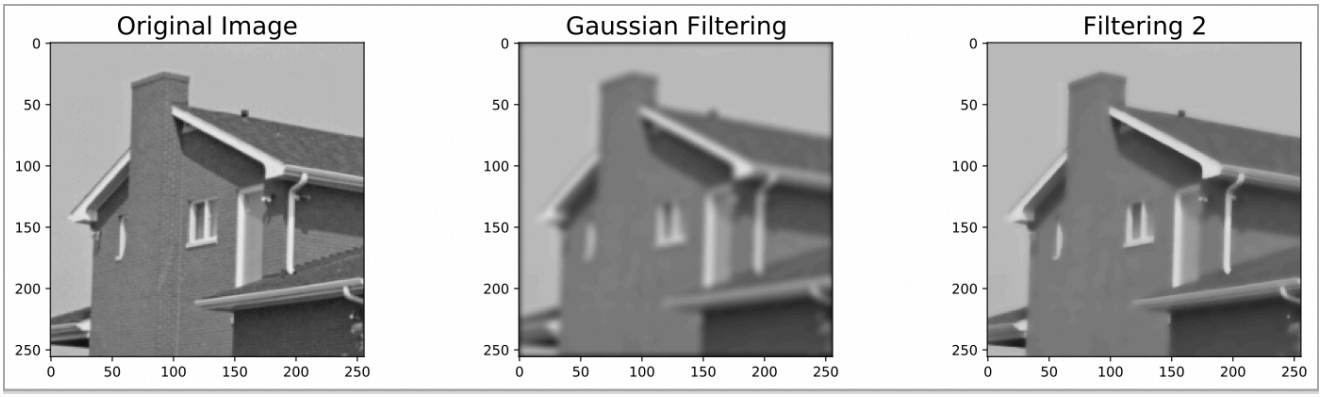
Yes, the 1D-Gaussian does satisfy the required properties.

$$GF[I_p] = \sum_{q \in S} G_{\sigma_p}\left(\|p - q\|\right) G_{\sigma_i}\left(\|I_p -_I q\|\right) I_q.$$

5.5

```
def filtering_2(image: np.array, kernel: np.array = None, sigma_int:
float = None, norm_fac: float = None):
    pad = kernel.shape[0] // 2
    new_image = np.zeros_like(image)
    for r in range(image.shape[0]):
        for c in range(image.shape[0]):
            sum = 0
            norm = 0
            for i in range(r - pad, r + pad + 1):
                for j in range(c - pad, c + pad + 1):
                    if i >= image.shape[0] or j >= image.shape[0] or i
< 0 or j < 0:
                        continue
                    intensity_diff = np.abs(image[r][c] - image[i][j])
                    intensity_gaussian =
((1/(np.sqrt(2*np.pi)*sigma_int)) * np.exp(-(intensity_diff**2)/(2 *
sigma_int**2)))
                    spatial_gaussian = kernel[i - (r - pad)][j - (c -
pad)]
                    sum += spatial_gaussian * intensity_gaussian *
image[i][j]
            new_image[r][c] = sum * norm_fac
    return new_image
```

5.6



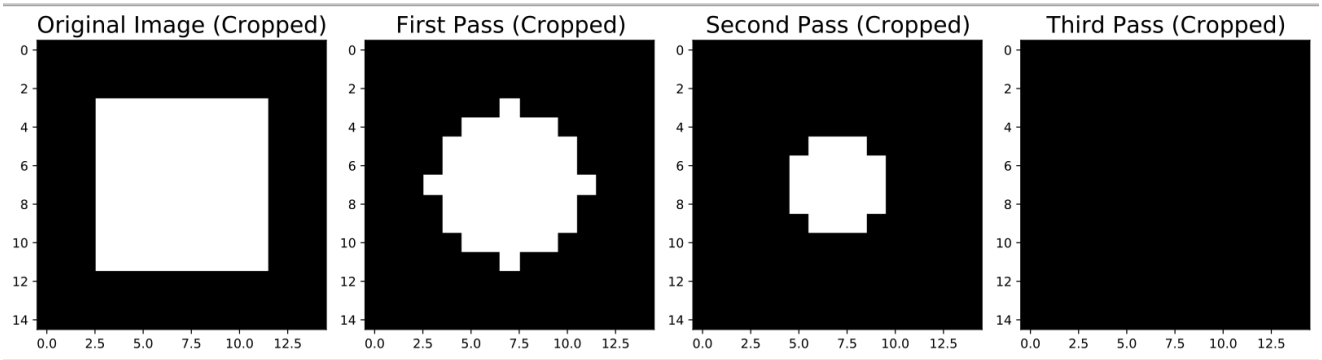
5.7

A bilateral filter followed by a median filter could work. The bilateral filter could help smooth out similarly colored areas using a small intensity sigma and the median filter would help smooth out some extreme values in the image to give it a more 'cartoon' look. An example is shown below but more tuning could make it even more 'cartoonish'.

6

As the median filter is run, the square will turn into a blocky circle and then disappear. This is because when the sliding window touches the corner of the square, $\frac{3}{4}$ of the window will be black pixels so the median value will be black (0). The same holds true (though not exactly with $\frac{3}{4}$) for pixels near the corner/side of the image. The pixel directly centered on the edges (exactly 4 of them) will remain white as the window will contain more white pixels than black pixels.

On the second pass, something similar happens, causing the blocky circle to become even smaller, and on the 3rd pass, the white disappears.



Code used to generate the diagram:

```
original_image = np.zeros((256, 256))
original_image[123:132,123:132] = 1.0
first_pass = median_filtering(original_image, 9)
second_pass = median_filtering(first_pass, 9)
third_pass = median_filtering(second_pass, 9)

fig, ax = plt.subplots(1,4,figsize=(1 + 3*4.5,4))
display_axis(ax[0], original_image[120:135,120:135], 'Original Image
(Cropped)')
display_axis(ax[1], first_pass[120:135,120:135], 'First Pass
(Cropped)')
display_axis(ax[2], second_pass[120:135,120:135], 'Second Pass
(Cropped)')
display_axis(ax[3], third_pass[120:135,120:135], 'Third Pass
(Cropped)')
fig.tight_layout()
fig.savefig('median.pdf', format='pdf', bbox_inches='tight')
```