CROSS CORRELATION 2D

```
def cross_correlation_2d(im, kernel, path='same', padding='zero'):
    '''
    Inputs:
        im: input image (RGB or grayscale)
        kernel: input kernel
        path: 'same', 'valid', 'full' filtering path
            padding: 'zero', 'replicate'
    Output:
        filtered image
    '''
    # Fill in
    k = int((len(kernel[0] - 1))/2)
    if padding != 'replicate':
        paddedim = cv2.copyMakeBorder(im, k, k, k, k, cv2.BORDER_CONSTANT, value=0)
    else:
        paddedim = cv2.copyMakeBorder(im, k, k, k, k, cv2.BORDER_REPLICATE)
    if np.any(kernel < 0):
        output = paddedim.astype(float)
    else:
        output = paddedim
    # Cross Correlate
    for i in range(k, paddedim.shape[0]-k):
        for j in range(k, paddedim.shape[1]-k):
            if len(im.shape) > 2:
                for c in range(paddedim.shape[2]):
                    output[i, j, c] = np.sum(kernel * paddedim[i-k : i+k+1, j-k : j+k+1, c])
            else:
                output[i, j] = np.sum(kernel * paddedim[i-k : i+k+1, j-k : j+k+1])

    if path == 'full':
        # Return full output image
        return output
    elif path == "valid":
        # Crop by 2k on all sides to get only valid pixels (remove padding & pixels that use padding)
        return output[2*k : output.shape[0] - 2*k, 2*k : output.shape[1] - 2 * k]
    else:
        # Crop by k on all sides to get same size image (remove padding)
        return output[k : output.shape[0] - k, k : output.shape[1] - k]
```

CONVOLVE 2D

```
def convolve_2d(im, kernel, path='same', padding='zero'):
    '''
    Inputs:
        im: input image (RGB or grayscale)
        kernel: input kernel
        path: 'same', 'valid', 'full' filtering path
            padding: 'zero', 'replicate'
    Output:
        filtered image
    '''
    # Flip kernel
    kernel = np.rot90(np.rot90(kernel))
    # Call cross_correlation_2d
    return cross_correlation_2d(im, kernel, path, padding)
```

GAUSSIAN BLUR KERNEL 2D

```
def gaussian_blur_kernel_2d(k_size, sigma):
    '''
    Inputs:
        k_size: kernel size
        sigma: standard deviation of Gaussian distribution
    Output:
        Gaussian kernel
    '''
    # Create 1D gaussian kernel
    one_d =  np.ones([k_size,1], dtype="float")
    for i in range(k_size):
        one_d[i] = math.exp(-(i - (k_size-1)/2)**2 / (2 * sigma**2))

    # Normalize kernel
    normalized = one_d / np.sum(one_d)

    # Multiply matrix by itself transposed to get 2D kernel
    return normalized @ np.matrix.transpose(normalized)
```

Original	**Blurred with kernel size = 11, sigma = 1.5**
	

IMAGE SHRINKING

```
def image_shrinking(im, dim):
    '''
    Inputs:
        im: input image (RGB or grayscale)
        dim: output image size
    Output:
        Downsampled image
    '''
    # Filter the input image using Gaussian kernel
    k_size = 11
    sigma = 1.5
    kernel = gaussian_blur_kernel_2d(k_size, sigma)
    convolved_im = convolve_2d(im, kernel, "same", "replicate")
    # Resize the filtered image to output size dim
    return cv2.resize(convolved_im, (int(dim[0]), int(dim[1])))
```

Original

Resized without Blur

Resized with Blur where kernel size = 11, sigma = 1.5

SOBEL KERNEL

```
def sobel_kernel():
    '''
    Output:
        Sobel kernels for x and y direction
    '''
    # Fill in
    sobelx = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]])
    sobely = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]])
    return sobelx, sobely
```

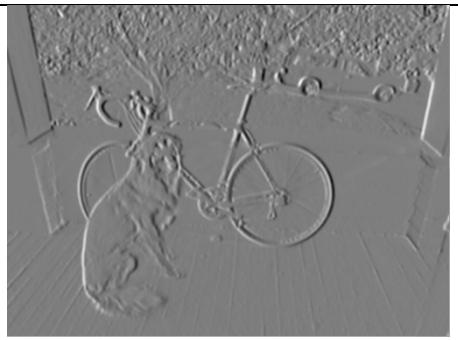
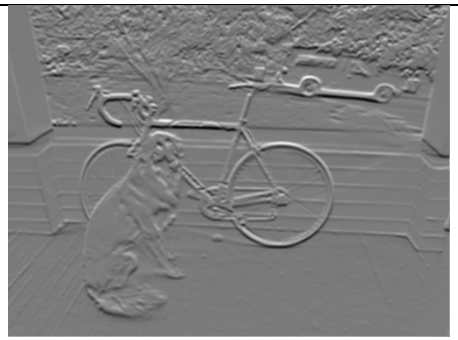
SOBEL IMAGE

```
def sobel_image(im):
    '''
    Inputs:
        im: input image (RGB or grayscale)
    Output:
        Gradient magnitude
        d of image in x direction
        d of image in y direction
        (All need to be normalized for visualization)
    '''


    # Convert image to grayscale if it is an RGB image
    im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
    # Fill in
    sobelx, sobely = sobel_kernel()

    dx = convolve_2d(im, sobelx, "valid", "replicate")
    dy = convolve_2d(im, sobely, "valid", "replicate")
    gradient_magnitude = np.sqrt(np.square(dx) + np.square(dy))

    # Normalize
    dx = (dx - np.min(dx))/(np.max(dx) - np.min(dx))
    dy = (dy - np.min(dy))/(np.max(dy) - np.min(dy))
    gradient_magnitude = (gradient_magnitude - np.min(gradient_magnitude))/(np.max(gradient_magnitude) -
np.min(gradient_magnitude))
    return gradient_magnitude * 255, dx * 255, dy * 255
```

			
Original	**Dx**	**Dy**	**Gradient Magnitude**