In [ ]:
```python
import numpy as np
from PIL import Image
```

In [ ]:
```python
# 2D-convolution on single-channel input with single-channel output
# Zero-padding on input is used such that output size equals input size
# Input: m x l numpy array X containing grayscale pixel data, and k x k kernel
# K encoding convolution filter weights (assume k is odd number)
# Output: m x l numpy array of convolution filter activations. Outputs are
# clipped to the interval [0,255]
def conv2d(X,kernel):
    kernel = np.array(kernel)
    m, l = X.shape
    k = kernel.shape[0]
    output_size_m = m - k + 1
    output_size_l = l - k + 1
    output = np.zeros((output_size_m, output_size_l))
    for i in range(output_size_m):
        for j in range(output_size_l):
            # Extract the region of interest from the input
            region = X[i:i+k, j:j+k]

            # Perform element-wise multiplication and sum
            activation = np.sum(region * kernel)

            # Clip the output to the interval [0, 255]
            output[i, j] = np.clip(activation, 0, 255)

    return output
```

In [ ]:
```python
# Converts PIL image to numpy array
def img_to_array(img):
  return np.asarray(img).astype('float32')

# Converts numpy array back to PIL image
def array_to_img(arr):
  return Image.fromarray(arr.astype('uint8'))
```

In [ ]:
```python
img1 = Image.open('images/image1.png')
img2 = Image.open('images/image2.png')
```

```
print('Image 1:')
display(img1)
print('Image 2:')
display(img2)
```

```
Image 1:
```

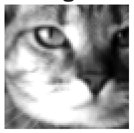
```
Image 2:
```

In [ ]:

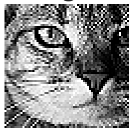
```
# Different 3 x 3 convolution filters
K_identity = np.matrix('0 0 0; 0 1 0; 0 0 0')
K_blur = np.matrix('0.0625 0.125 0.0625; 0.125 0.5 0.125; 0.0625 0.125 0.0625')
K_sharpen = np.matrix('0 -1 0; -1 5 -1; 0 -1 0')
K_ver_edges = np.matrix('-1 0 1; -1 0 1; -1 0 1')
K_hor_edges = np.matrix('-1 -1 -1; 0 0 0; 1 1 1')

# Applying filters to example images
print('Image 1 original:')
display(array_to_img(conv2d(img_to_array(img1),K_identity)))
print('Image 1 blurred:')
display(array_to_img(conv2d(img_to_array(img1),K_blur)))
print('Image 1 sharpened:')
display(array_to_img(conv2d(img_to_array(img1),K_sharpen)))
print('Image 2 original:')
display(array_to_img(conv2d(img_to_array(img2),K_identity)))
print('Image 2 vertical edges:')
display(array_to_img(conv2d(img_to_array(img2),K_ver_edges)))
print('Image 2 horizontal edges:')
display(array_to_img(conv2d(img_to_array(img2),K_hor_edges)))
```

```
Image 1 original:
```

```
Image 1 blurred:
```

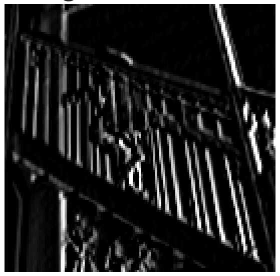


```
Image 1 sharpened:
```

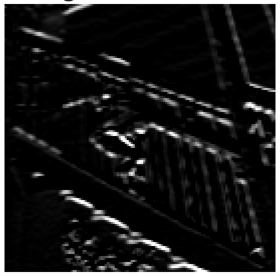


```
Image 2 original:
```

```
Image 2 vertical edges:
```



```
Image 2 horizontal edges:
```



```
In [ ]:
```

Task 2 and 3

The thing is, I solved the task 2 by implementing the general case. So, basically the answer to both questions are given below:

In [ ]:
```
import numpy as np

# derived from previous part
# used for calculating Y
def conv2d(X,kernel):
    m, l, d = X.shape
    k, k, d = kernel.shape
    output_size_m = m - k + 1
    output_size_l = l - k + 1
    output_size_d = d - d + 1
    output = np.zeros((output_size_m, output_size_l, output_size_d))
    for i in range(output_size_m):
        for j in range(output_size_l):
            for r in range(output_size_d):
                region = X[i:i+k, j:j+k, r:r+d]
                output[i, j] = np.sum(region * kernel)
    return output
```

In [ ]:
```
m = 4
l = 4
d = 2
k = 2
# creating the matrixes and initializing them
X = np.zeros((m, l, d))
K = np.zeros((k,k,d))
counter = 1
for r in range(d):
    for i in range(k):
        for j in range(k):
            K[i,j,r] = counter
            counter+=1
counter+=2
for r in range(d):
    for i in range(m):
        for j in range(l):
            X[i,j,r] = counter
            counter+=1
Y = conv2d(X, K)

# this is the function for the general case - which also does the job for task 2
def equivalent_matrix_vector_conv(X,kernel):
    m, l, d = X.shape
    k, k, d = kernel.shape
    output_size_m = m - k + 1
    output_size_l = l - k + 1
    output_size_d = d - d + 1
    # creating the vector b
    b = kernel.reshape((-1,))
    A = np.zeros((
        output_size_m * output_size_l, b.shape[0]
    ))
    index = 0
    # creating the matrix A
    for i in range(output_size_m):
        for j in range(output_size_l):
            for r in range(output_size_d):
                region = X[i:i+k, j:j+k, r:r+d].reshape((-1,))
                A[index] = region
                index+=1
    # the linear multiplication
    output = A@b
    print(f'A is : {A}')
    print(f'b is : {b}')
    return output.reshape((output_size_m, output_size_l, output_size_d))

Y_prime = equivalent_matrix_vector_conv(X,K)

print(f'Y is : {Y}')
print(f'Y_prime is : {Y_prime}')

print(f'These were the same\nThe general algorithm is also implemented by the function "equivalent_matrix_vector_conv
A is : [[11. 27. 12. 28. 15. 31. 16. 32.]
 [12. 28. 13. 29. 16. 32. 17. 33.]
 [13. 29. 14. 30. 17. 33. 18. 34.]
 [15. 31. 16. 32. 19. 35. 20. 36.]
 [16. 32. 17. 33. 20. 36. 21. 37.]
 [17. 33. 18. 34. 21. 37. 22. 38.]
 [19. 35. 20. 36. 23. 39. 24. 40.]
 [20. 36. 21. 37. 24. 40. 25. 41.]
 [21. 37. 22. 38. 25. 41. 26. 42.]]
b is : [1. 5. 2. 6. 3. 7. 4. 8.]
Y is : [[[ 920.]
  [ 956.]
  [ 992.]]

 [[1064.]
  [1100.]
  [1136.]]

 [[1208.]
  [1244.]
  [1280.]]]
Y_prime is : [[[ 920.]
  [ 956.]
  [ 992.]]

 [[1064.]
  [1100.]
  [1136.]]

 [[1208.]
  [1244.]
  [1280.]]]
These were the same
The general algorithm is also implemented by the function "equivalent_matrix_vector_conv"
```

the matrix A has the dimentions: (m - k + 1)*(l - k + 1) times (k*k*d) the vecor b has the dimentions: k*k*d

```
In [ ]:
```