image analysis lab

Image analysis applications in Data Science

Images exists in all areas of workspace, personal life and social context. Images are a widely used unstructured data type, and analysis and preparation of images is a common data science task.

Explore an image

In exploring the properties of a greyscale image, images are cached as files in the working directory:

```
import os
import urllib
import urllib2
url = "https://github.com/MicrosoftLearning/Applied-Machine-
Learning/raw/master/Labs/Faces/Steve.jpg"
fileobject = urllib2.urlopen(url)

from scipy import misc
steve = misc.imread(fileobject, mode = 'L')
```

The image will be stored as ordinary Numpy array of unsigned 8-bit integers of dimension 1661x1113.

```
type(steve)
```

numpy.ndarray

```
steve.dtype
```

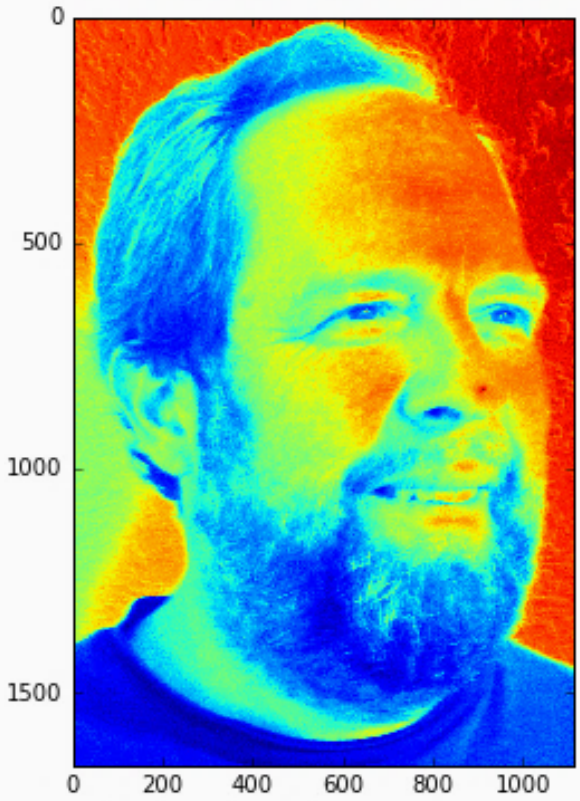
dtype('uint8)

```
steve.shape
```

(1661,1113)

Plotting a Numpy array image using the imshow function from the matplotlib package:

```
%matplotlib inline
def plot_im(im):
    import matplotlib.pyplot as plt
    import numpy as np
    fig = plt.figure(figsize=(8, 6))
    fig.clf()
    ax = fig.gca()
    ax.imshow(np.array(im).astype(float))
    return 'Done'
plot_im(steve)
```

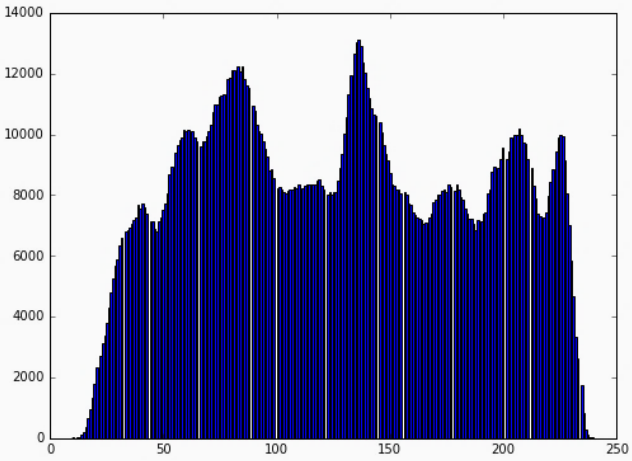


Histograms and Equalization

It is often the case that a raw image does not have the favorable statistical properties required for further analysis. For example, poor contrast in the image can make it difficult to detect features. Histogram equalization is a widely used method for improving the properties of an image.

Ideally, the histogram of the image should be close to a uniform distribution. that the Numpy **flatten** method is applied to the array. This method removes the dimension attribute, creating a 1-D array:

```
def hist_im(im, bins = 256):
    """ Display histogram of flattened image"""
    import matplotlib.pyplot as plt
    import numpy as np
    fig = plt.figure(figsize=(8, 6))
    fig.clf()
    ax = fig.gca()
    ax.hist(np.array(im).flatten(), bins = bins)
    return 'Done'
hist_im(steve)
```



Examining the **Histogram** illustrates a **non-uniform distribution** of the image above.

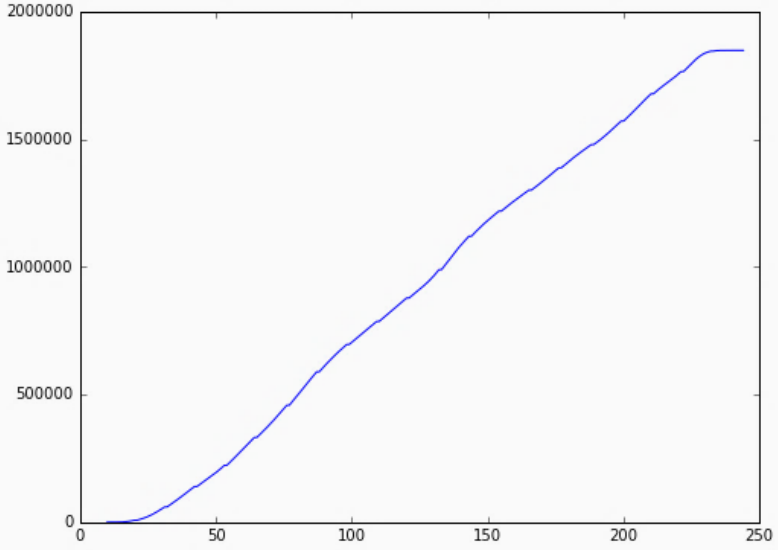
Another tool for visualizing image statistics is the **cumulative distribution function** (**CDF**) plot:

```
def cdf_im(im, bins = 256):
    """Display cumulative distirbution of flattened image"""
    import matplotlib.pyplot as plt
    import numpy as np

    y, x = np.histogram(np.array(im).flatten(), bins = bins)
    y = y.cumsum()

    fig = plt.figure(figsize=(8, 6))
    fig.clf()
    ax = fig.gca()
    ax.plot(x[:256], y)
    return 'Done'
cdf_im(steve)
```

The **CDF** of an image with uniformly distributed pixel value is a straight line; the above **CDF** is curved, particularly at the ends.



Histogram equalization is often used to improve the statistics of images. In simple terms, the histogram equalization algorithm attempts to adjust the pixel values in the image to create a more uniform distribution. The code below uses a simple **linear interpolation** method on the image histogram to equalize the image:

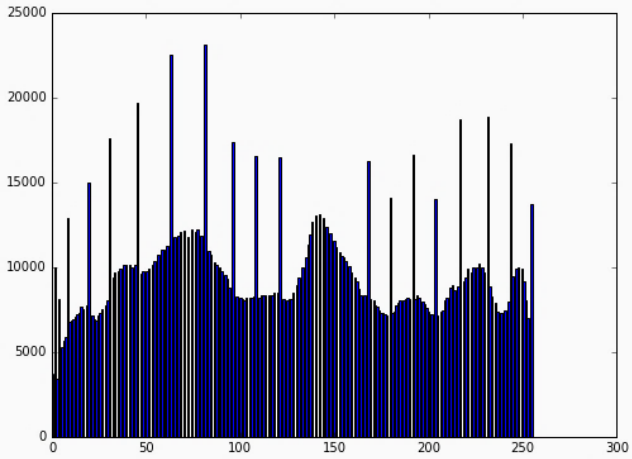
Note: The histogram will appear to have spikes, which are the result of the binning, and not a problem with the equalization.

```
def image_equalize(im, num_bins = 256):
    """Function to equalize the image"""
    import numpy as np
    ## Compute the histogram of flattened image
    imhist, bins = np.histogram(im.flatten(), num_bins, normed=True)

    cdf = imhist.cumsum() #cumulative distribution function
    cdf = 255 * cdf / cdf[-1]  # normalize

    ## Interpolate to equalize the image
    out = np.interp(im.flatten(), bins[:-1], cdf)
    return out.reshape(im.shape)

steve_eq = image_equalize(steve)
hist_im(steve_eq)
cdf_im(steve_eq)
```

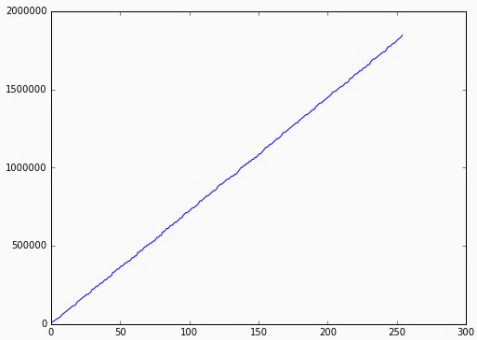


The histogram of the equalized image is more uniform in appearance.

Also displayed is the **CDF** of the equalized image:

The **CDF** is more linear than before. Both the histogram and CDF indicate that the statistics of the image have improved.

A comparison of the actual images are illustrated below between the **unequalized → equalized** image respectively:




```
def plot_im2(im1, im2):
    import matplotlib.pyplot as plt
    import numpy as np
    fig, ax = plt.subplots(1, 2, figsize = (6,12))
    ax[0].imshow(im1)
    ax[1].imshow(im2)
    return 'Done'
plot_im2(steve, steve_eq)
```

The **original image** is on the **left** and the **equalized image** is on the **right**. Notice the improved contrast in the equalized image.

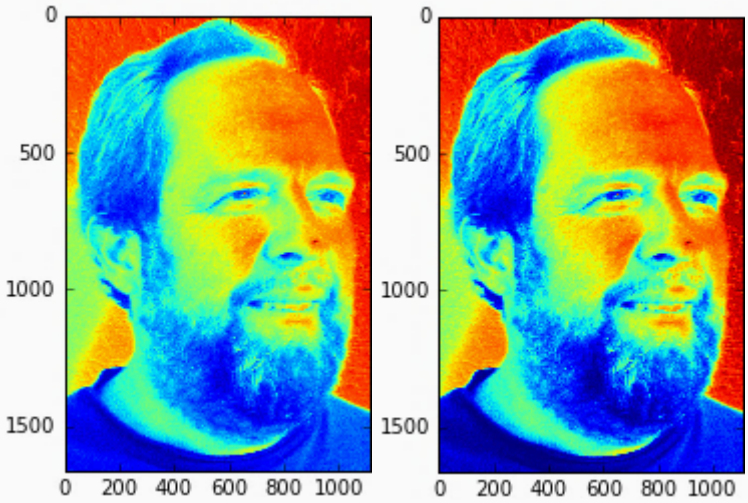
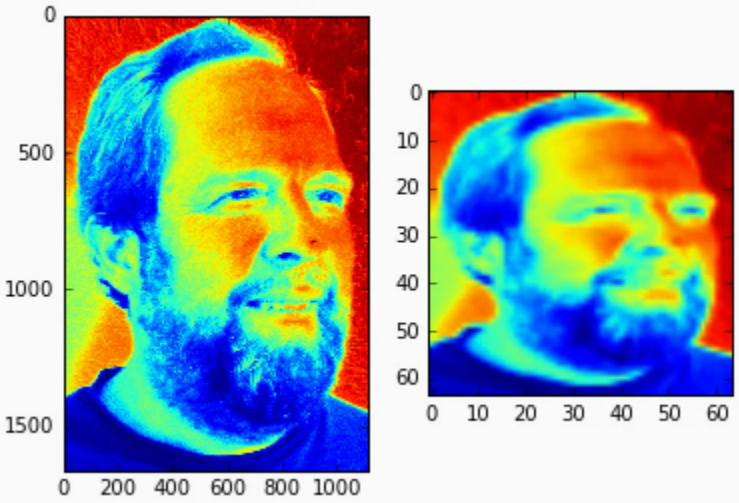


Image Manipulation

After examining the properties of images, basic image manipulation is applied where appropriate.

Resizing is a common form of image manipulation. Images are often resampled to a smaller size to reduce the amount of data which must be processed. The **scipy.misc.imresize** method provides a convenient way to resize an image:

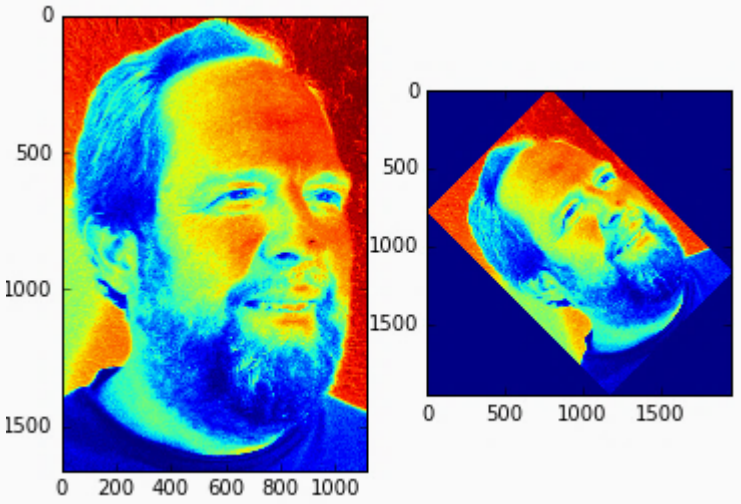
```
def resize(im, size = (64, 64)):
    import scipy.misc as mc
    return mc.imresize(im, size)
plot_im2(steve_eq, resize(steve_eq))
```



The reduced image output is more course and granular than the original. However, the integrity of the original image is maintained considering it was reduced by a factor of >500.

Rotation is performed by **pixel interpolation**.
The **scipy.ndimage.interpolation.rotate** method is to rotate the image by 45°:

```
def rotate(im, angle = 45):
    from scipy.ndimage import interpolation
    return interpolation.rotate(im, angle)
plot_im2(steve_eq, rotate(steve_eq))
```



The rotated image on the right is a duplicate of the image on the left. Notice that the area around the rotated image has been backfilled with zero values.

Working with a list of images

Typically multiple images are analyzed as a group. This group of images can be stored as a list object in Python:

```
import os
import urllib
import urllib2

baseUrl = "https://github.com/MicrosoftLearning/Applied-Machine-
Learning/raw/master/Labs/CarrotImages/Carrot"

image_list = []
for i in range(1,10):
    url = baseUrl + str(i) + ".JPG"
    fileobject = urllib2.urlopen(url)
    im = misc.imread(fileobject)
image_list.append(im)
```

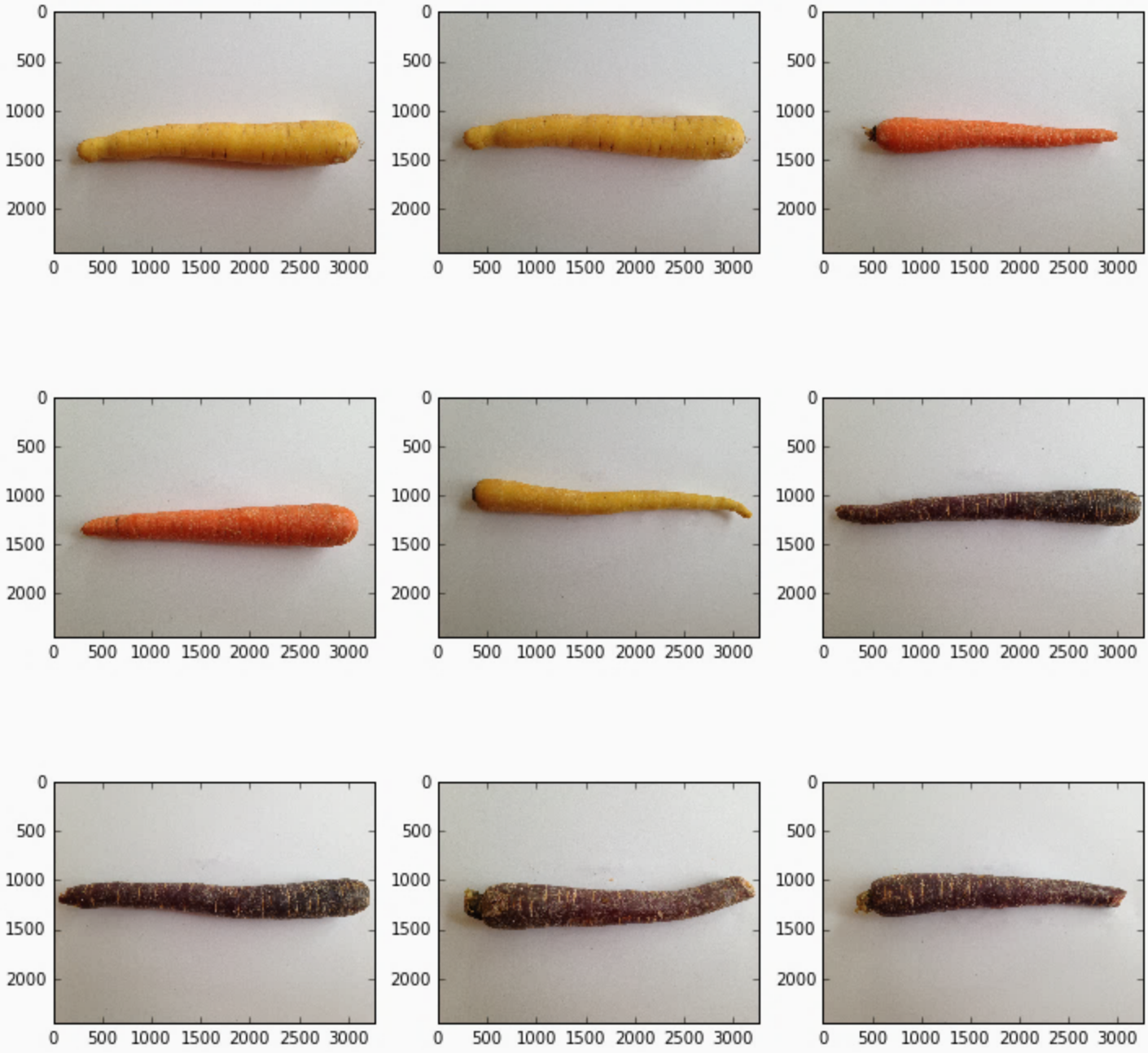
The list contains 9 images and the numpy image array has dimensions of (2448, 3264, 3):

```
len(image_list)
image_list[0].shape
```

The Numpy image array has three dimensions. These are color images, with red, green and blue layers, each stored as 2x2 arrays.

The code in the function below integrates over the images in the list and displays each one into an array of axes (images of carrots):

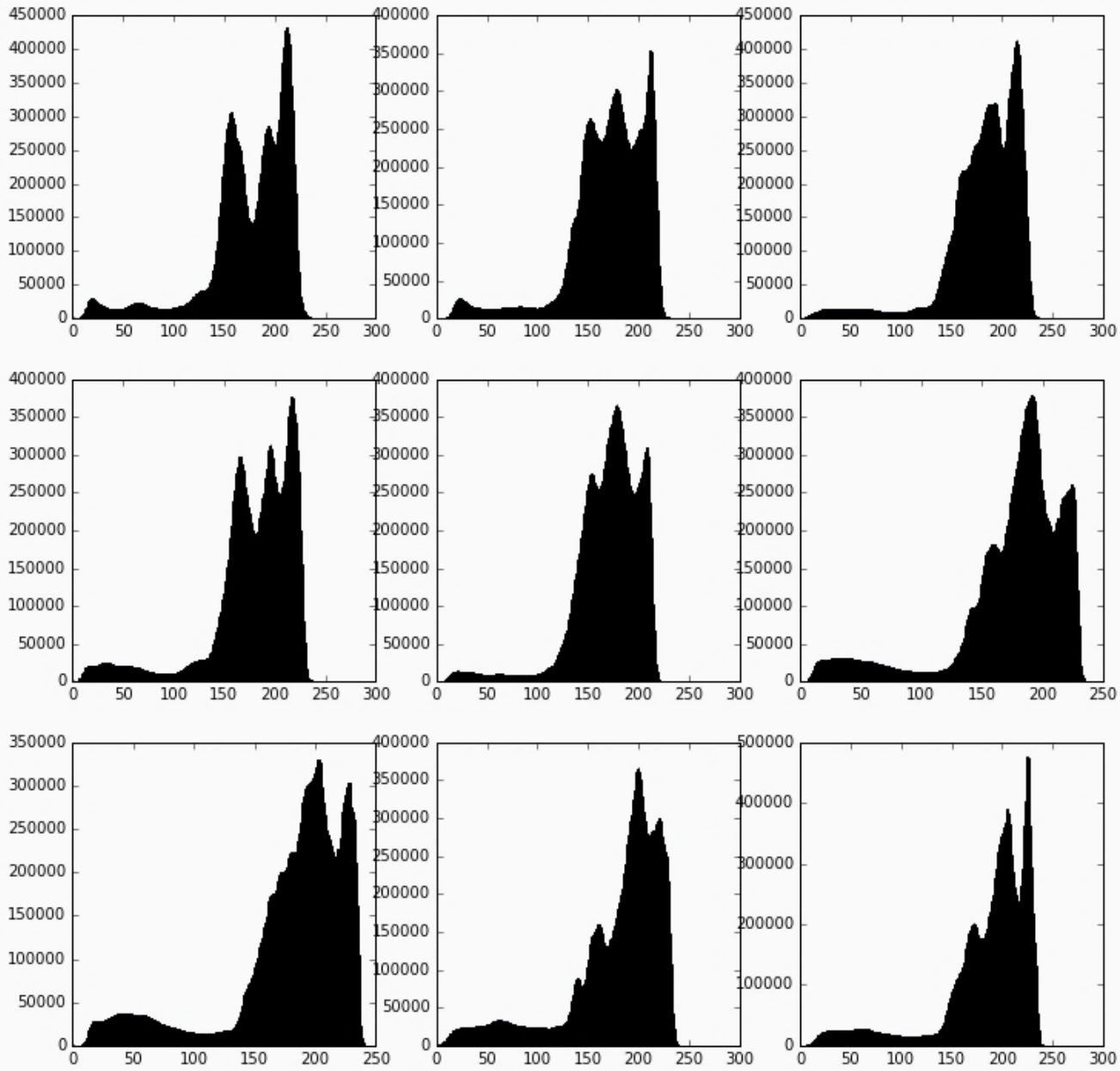
```
def plot_carrot(im_list):
    import matplotlib.pyplot as plt
    fig, ax = plt.subplots(3, 3, figsize = (12,12))
    j = -1
    for i, image in enumerate(im_list):
        k = i % 3
        if k == 0: j += 1
        ax[j,k].imshow(image)
    return 'Done'
plot_carrot(image_list)
```



The 9 images of carrots which have different shapes, several colors and different orientations.

The code below loops over the list of images and plots each histogram into an array of axes:

```
def hist_carrot(im_list, bins = 256):
    """ Display histogram of flattened image"""
    import matplotlib.pyplot as plt
    import numpy as np
    fig, ax = plt.subplots(3, 3, figsize = (12,12))
    j = -1
    for i, image in enumerate(im_list):
        k = i % 3
        if k == 0: j += 1
        ax[j,k].hist(np.array(image).flatten(), bins = bins)
    return 'Done'
hist_carrot(image_list)
```



Each image has a long left tail, arising from the background.

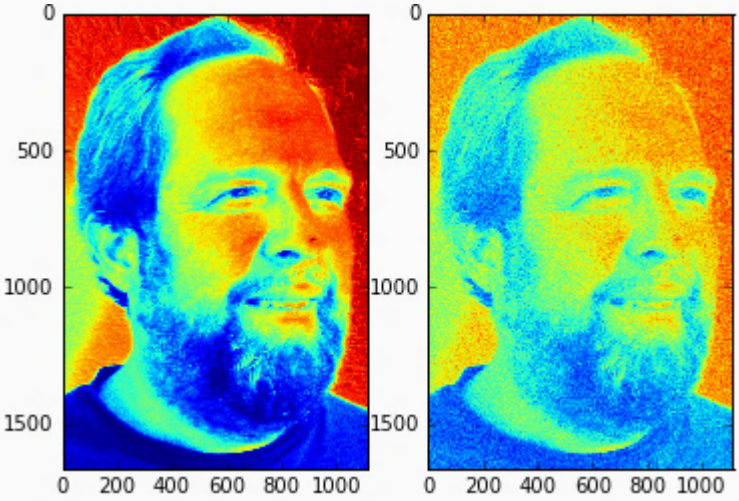
Image Filtering

Images have been loaded, properties explored, and basic processing performed. Next will apply specific filters to the images. **Filters** are applied to images in order to either enhance aspects of the image or to remove undesired properties of the image (such as noise).

Gaussian or white noise is added to the face image. The code below does the following:

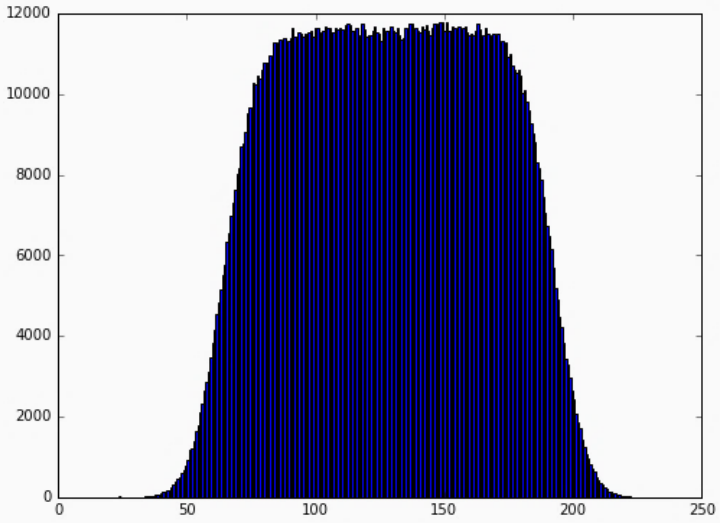
- Generate a one dimensional array of Gaussian noise.
- Shape the noise array to match the image.
- Add the noise array to the image.
- Ensure there are no image values less than zero.

```
def add_noise(im, mean = 128, sd = 20):
    """Adds Gausian noise to the image"""
    from numpy.random import normal
    import numpy as np
    im_a = np.array(im)
    shape = im_a.shape
ng = normal(loc = mean, scale = sd, size
= shape[0] * shape[1])
    ng.shape = shape
    ng = np.divide( np.add(ng, im_a), 2.0)
    ng[np.where( ng < 0 )] = 0
    return ng
steve_n = add_noise(steve_eq)
plot_im2(steve_eq, steve_n)
```



The original image in on the left and the noisy image is less distinct on the right.

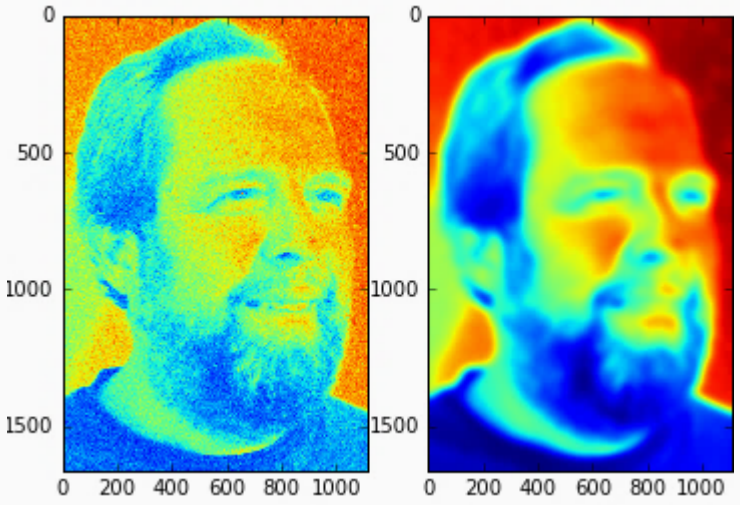
The Histogram (`hist_im(steve_n)`) of the noisy image is a smoother and more uniform. The range of pixel values is also limited. These changes are the result of adding white noise to the image, a process often referred to as 'pre-whitening':



Next applies a Gaussian filter. A **Gaussian filter** is a two dimensional filter using a Gaussian or bell-shaped curve kernel. In effect, the **Gaussian filter** is a smoothing filter. The span of the filter determines the degree of smoothing of the filter.

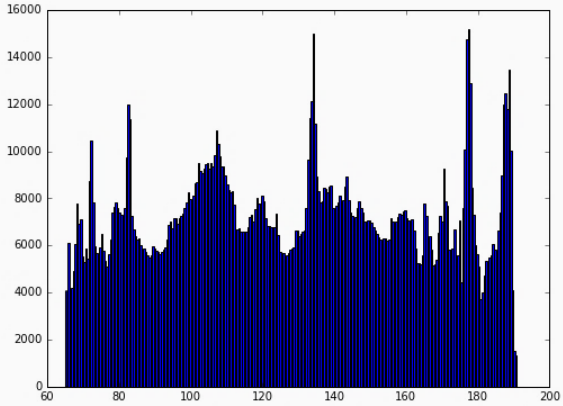
```
def gauss_filter(im, sigma = 16):
    from scipy.ndimage.filters import
gaussian_filter as gf
    import numpy as np
    im_a = np.array(im)
    return gf(im_a, sigma = sigma)
steve_g = gauss_filter(steve_n)
plot_im2(steve_n, steve_g)
```

The 2-d span of the filter is specified in pixels:



- The filtered image on the right has smoother or blurred features. For this reason, Gaussian filters are often called blurring filters.
- The image on the right does not exhibit the 'salt and pepper' noise of the image on the left.

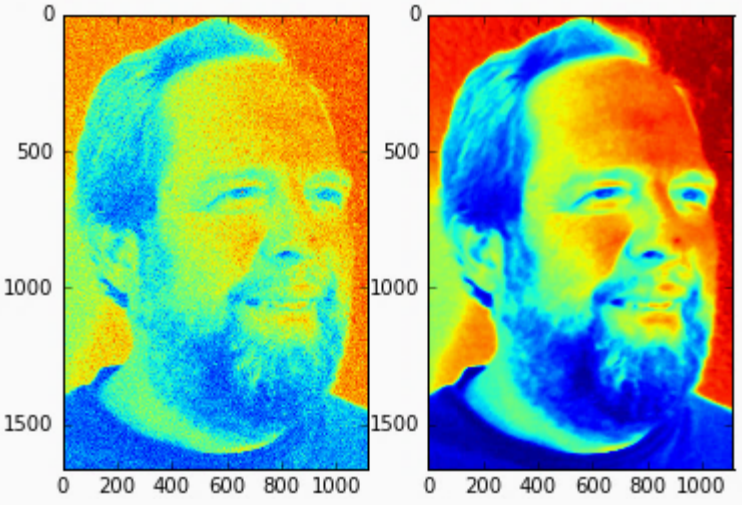
The Histogram (`hist_im(steve_n)`) of the filtered image is jagged when compared to the noisy image. This texture arises from removing the 'whitening' from the Gaussian noise.



Next applys a **median filter** on the image. Whereas, a Gaussian filter acts as a smoother, a median filter enhances edges and transitions.

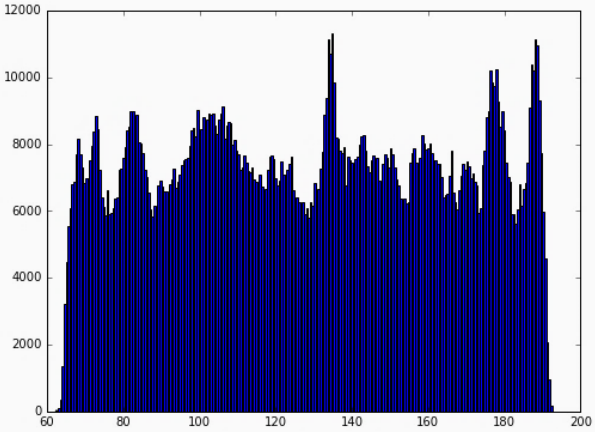
The code below applies a **median filter** to the image. The median filter kernel is a rectangular patch with a span specified as the filter size:

```
def med_filter(im, size = 16):
    from scipy.ndimage.filters import
median_filter as mf
    import numpy as np
    im_a = np.array(im)
    return mf(im_a, size = size)
steve_m = med_filter(steve_n)
plot_im2(steve_n, steve_m)
hist_im(steve_m)
```



Comparing the filtered image on the right to the original noisy image on the left and to the Gaussian filtered image on the right:

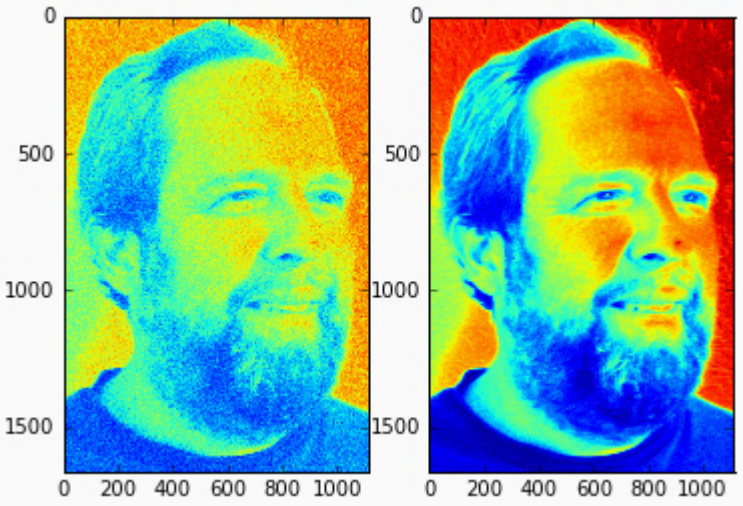
- The edges and transitions in the median filtered image are sharper and more distinct when compared to the noisy image or the Gaussian filtered image on the left.
- Regions between edges in the median filtered image are more uniform looking and do not have the 'salt and pepper' look of the noisy image on the right.
- The filtering has smoothed the histogram slightly.



Using a Different Span

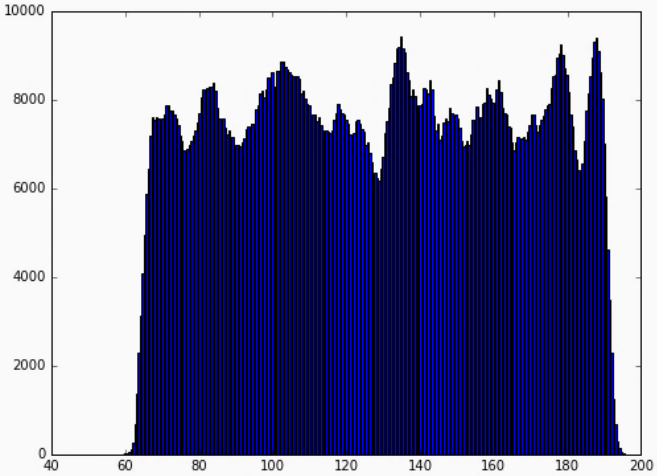
Applying a Gaussian filter to the noisy image with a different span (**sigma value = 2**) and visualizing the result:

```
def gauss_filter2(im, sigma = 2):
    from scipy.ndimage.filters import
gaussian_filter as gf
    import numpy as np
    im_a = np.array(im)
    return gf(im_a, sigma = sigma)
steve_g = gauss_filter2(steve_n)
plot_im2(steve_n, steve_g)
```



The original noisy image and the Gaussian filtered reveal:

- The filtered image shows less salt and pepper noise than the original.
- Compared to the first filtered image, the features in the new filtered image are less blurred.



Next will **pre-whiten** the carrot images by adding Gaussian noise. The process is the same as before.

The code below iterates over the images in the list and adds Gaussian noise to each image in turn:

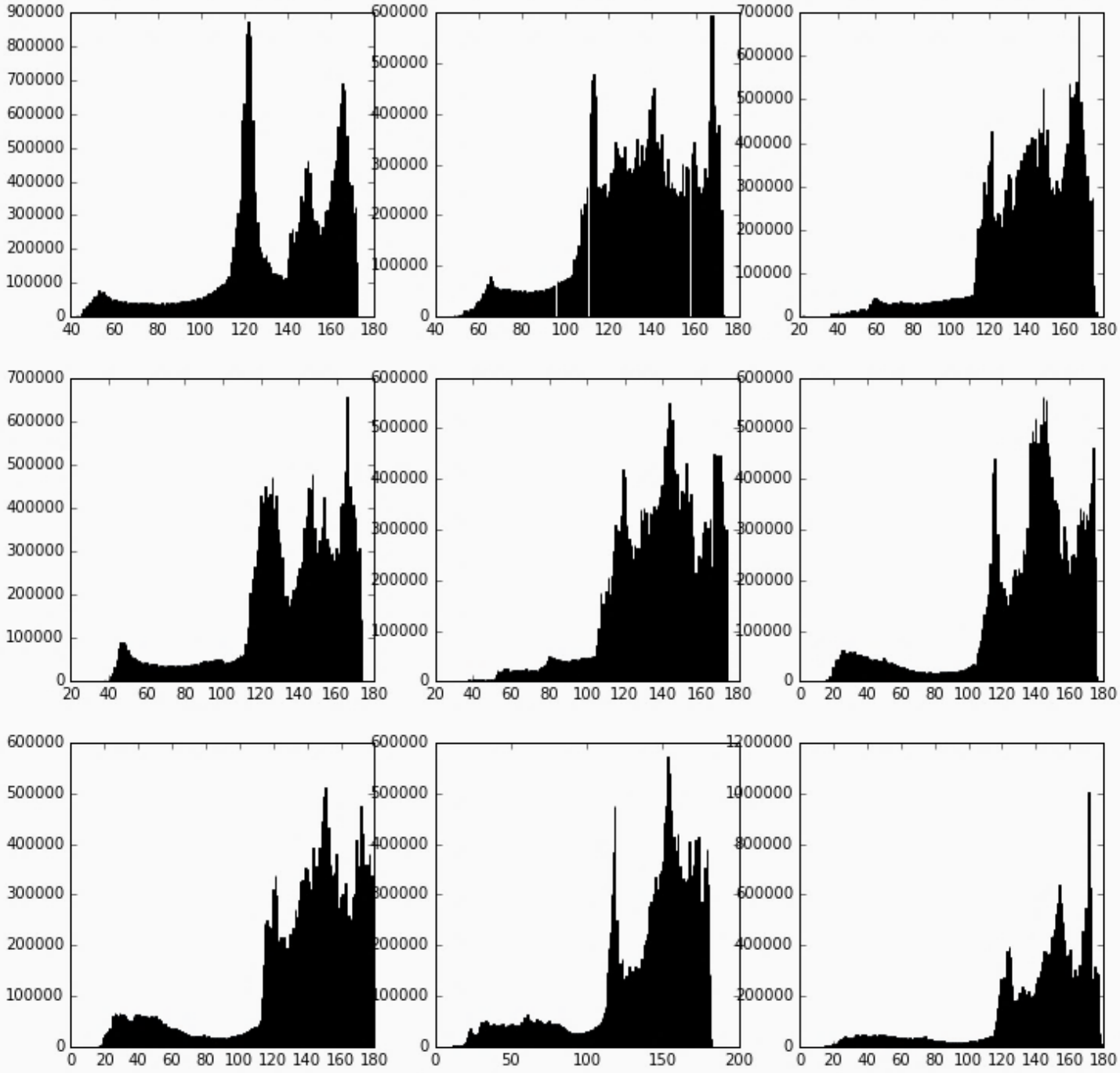
```
def pre_white(im_list, mean = 0, sd = 20):
    """Adds Gausian noise to the image"""
    from numpy.random import normal
    import numpy as np
    out = []
    for image in im_list:
        shape = image.shape
        ng = normal(loc = mean, scale = sd, size = shape[0] * shape[1] * shape[2])
        ng.shape = shape
        ng = np.add(ng, image)
        ng[np.where( ng < 0 )] = 0
        ng *= 255.0 / np.amax(ng)  # normalize
        ng = ng.astype(np.uint8)
        out.append(ng)
    return out
carrot_filter = pre_white(image_list)
```

Next applies a Gaussian filter to the images. The code below iterates over the list of images and applies the filter to each image:

```
def gauss_filter(im_list, sigma = 20):
    from scipy.ndimage.filters import gaussian_filter as gf
    out = []
    for image in im_list:
        out.append(gf(image, sigma = sigma))
    return out
carrot_filter = gauss_filter(carrot_filter)
```

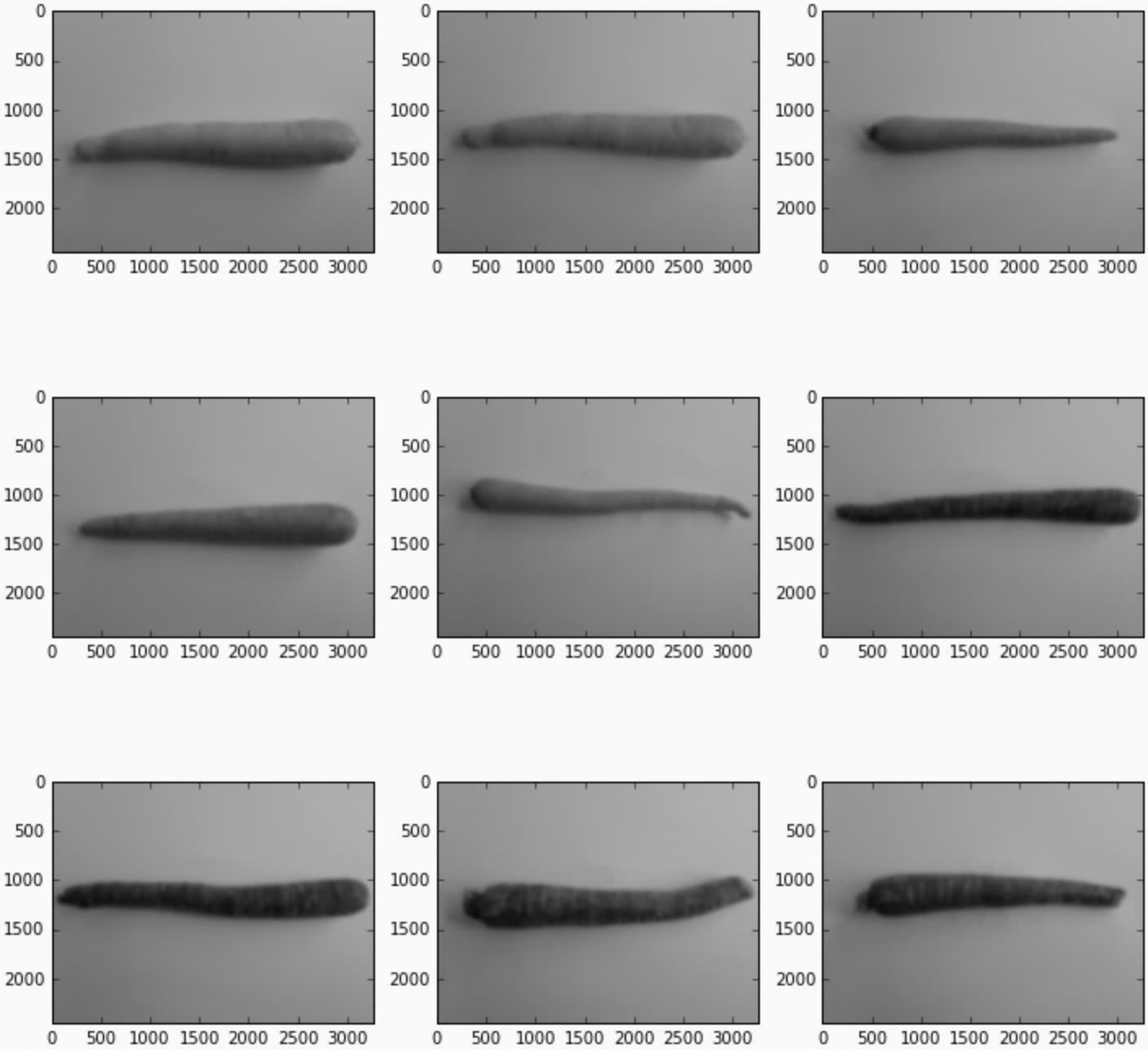
The **pre-whitened histograms** and images are displayed by executing the code below:

```
hist_carrot(carrot_filter)
plot_carrot(carrot_filter)
```



Comparing the histograms and images to the original histograms and images, note the following:

- The histograms of the pre-whitened images have a more jagged appearance.
- The pre-whitened images are blurred or softer and have lost considerable color contrast when compared to the original images.


0 500 1000 1500 2000 2500 3000

Feature Extraction

Images have now been explored, manipulated and filtered. Next extracts features from the processed images. **Feature extraction** is an indispensable step in preparing image data for machine learning.

First applies the **Sobel Edge Detection Algorithm**. The Sobel edge detection algorithm finds regions of the image with large gradient values in multiple directions. Regions with **high omnidirectional gradient** are likely to be **edges** or **transitions** in the pixel values.

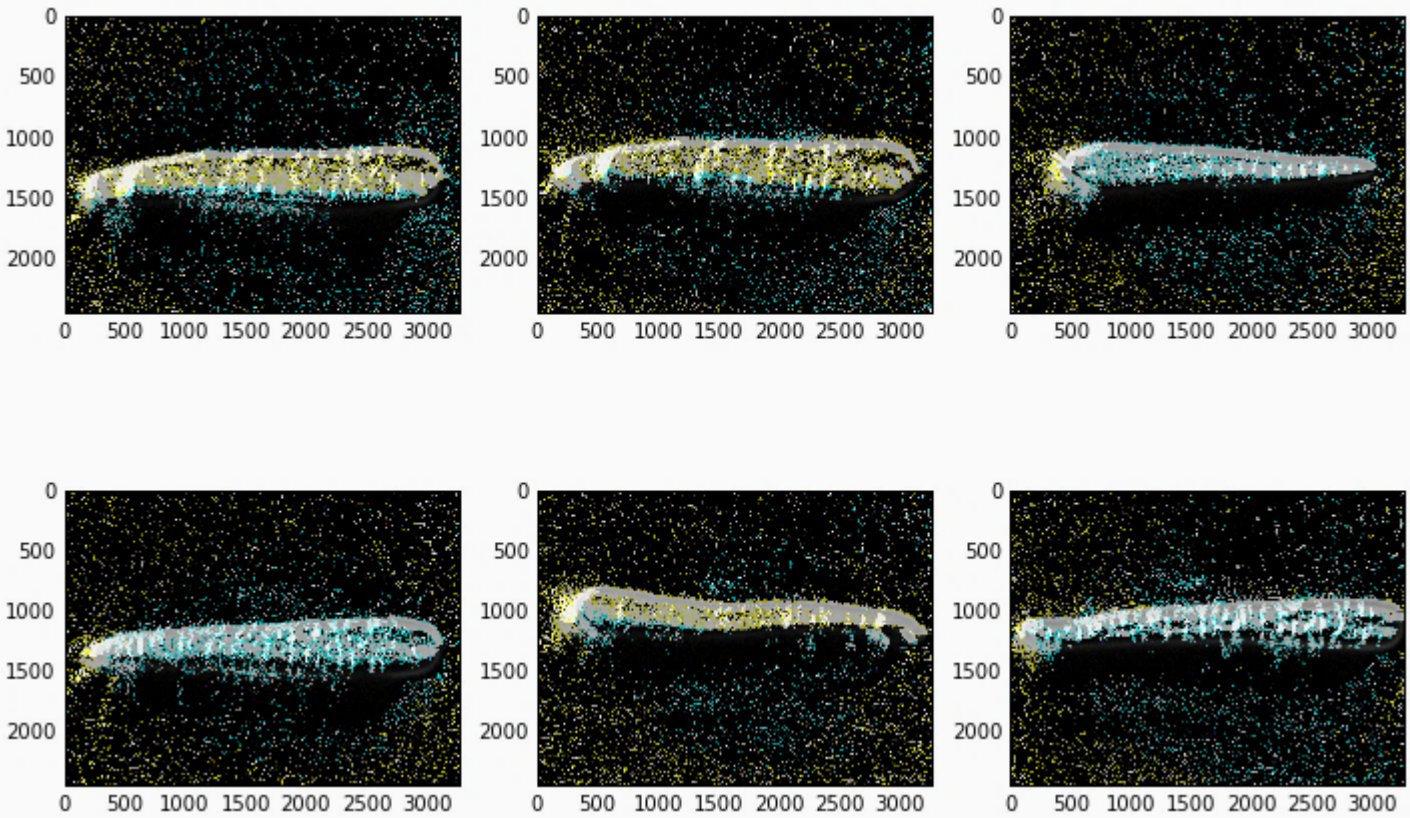
The code below applies the Sobel algorithm to a list of images, using these steps:

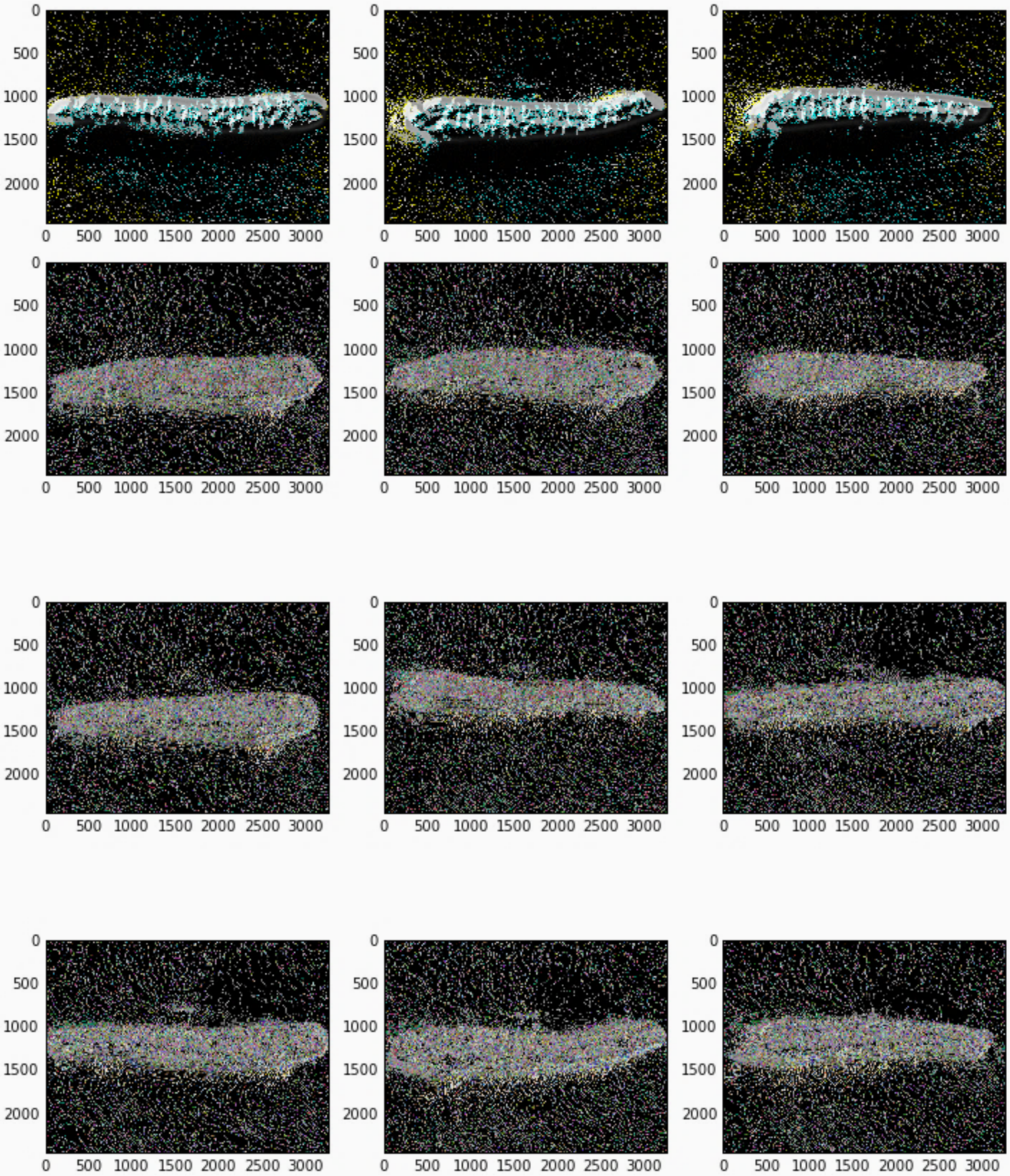
- The gradient in the x and y (horizontal and vertical) directions are computed.
- The magnitude of the gradient is computed.
- The gradient values are normalized.

```
def edge_sobel(im_list):
    from scipy import ndimage
    import numpy as np
    out = []
    for image in im_list:
        dx = ndimage.sobel(image, 1)  # horizontal derivative
        dy = ndimage.sobel(image, 0)  # vertical derivative
        mag = np.hypot(dx, dy)  # magnitude
        mag *= 255.0 / np.amax(mag)  # normalize (Q&D)
        mag = mag.astype(np.uint8)
        out.append(mag)
    return out
```

```
carrot_filter = edge_sobel(carrot_filter)#
```

```
plot_carrot(carrot_filter)
```

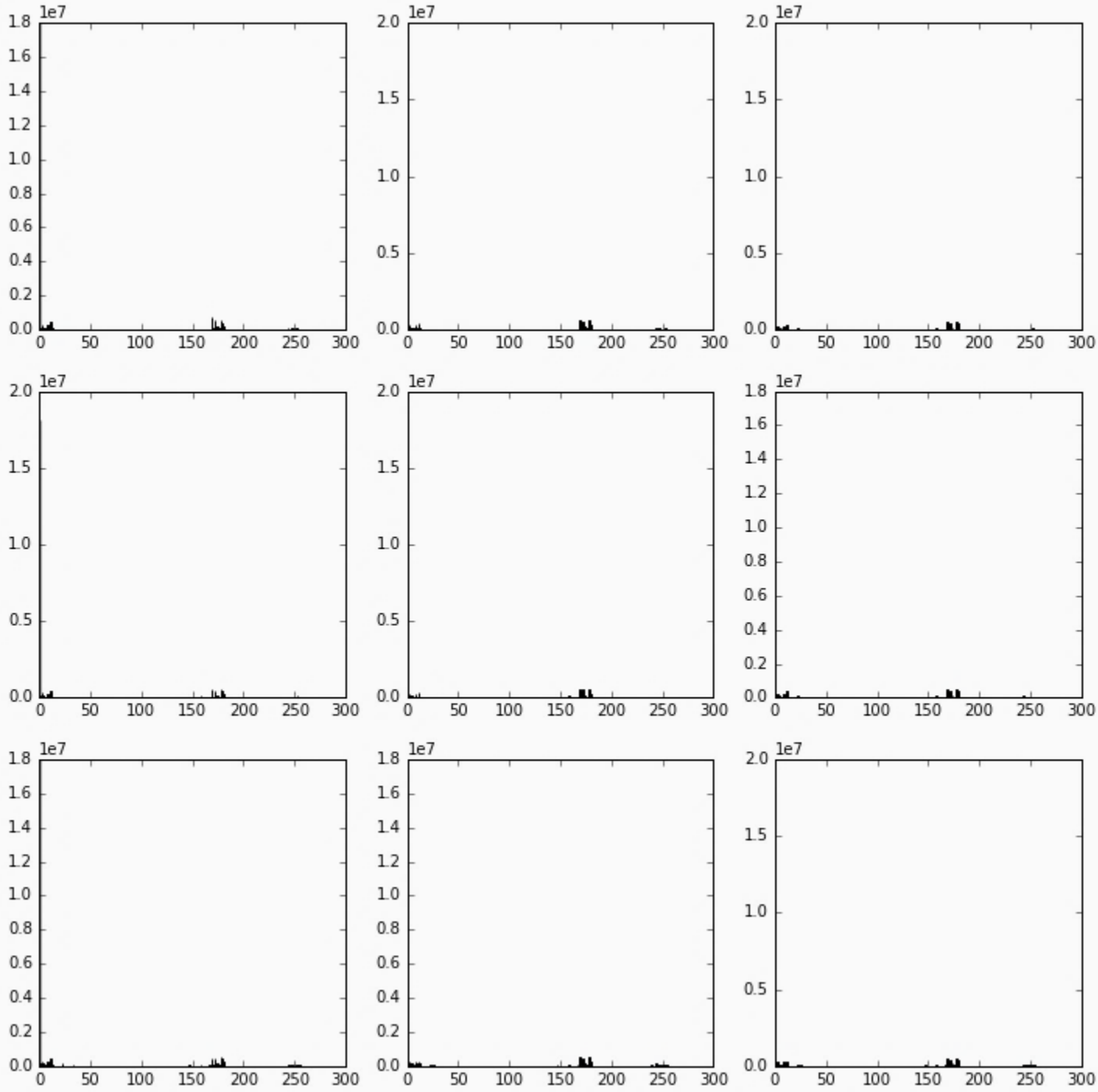




The carrots now appear in a skeletal form. The edges of the carrots are clear, along with the numerous cross marks on each carrot. These regions of the carrot image have high pixel gradient values.

The code in the cell below plots the histograms of the gradients of the images:

```
hist_carrot(carrot_filter)
```



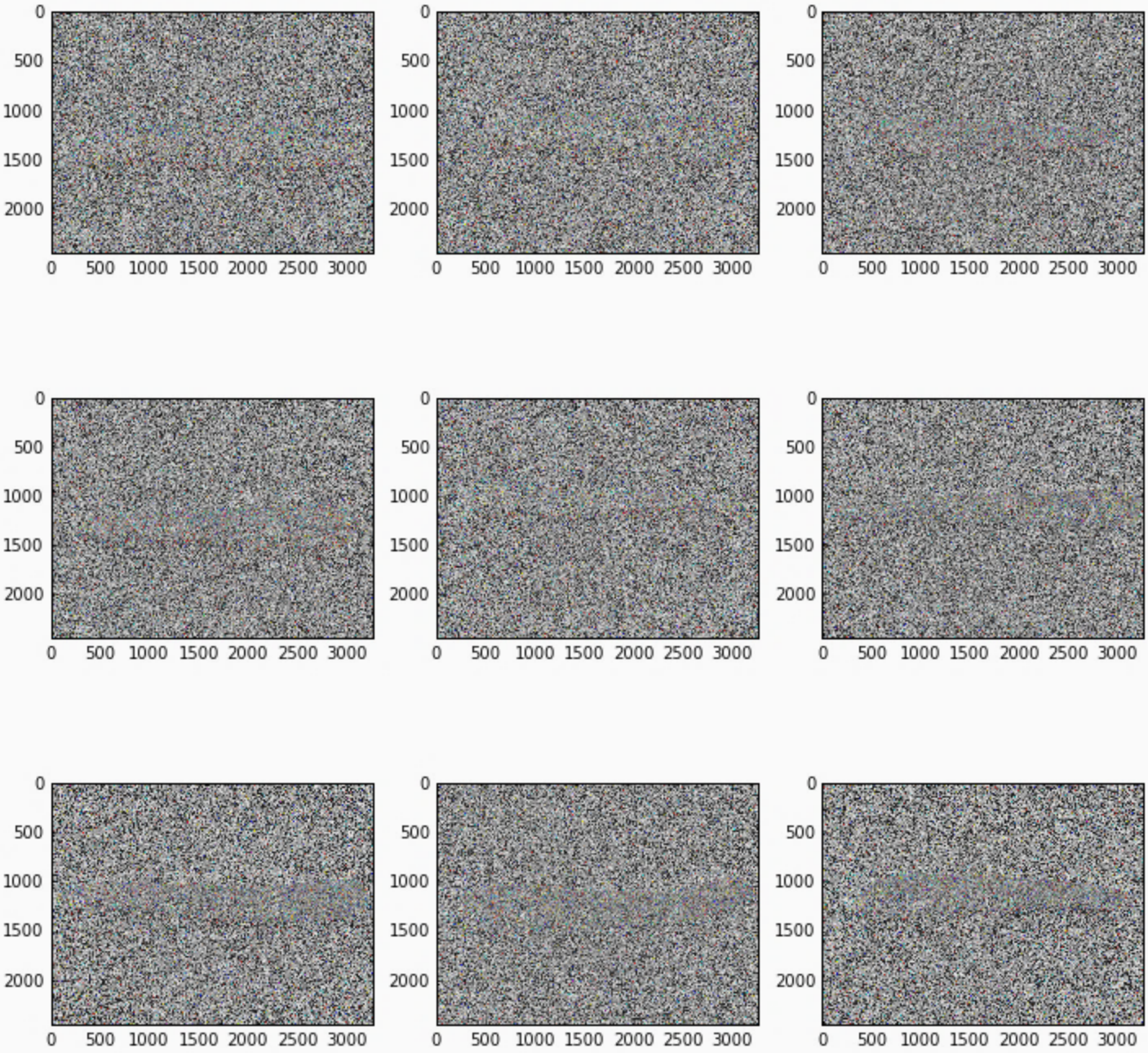
The pixel values in each histogram are in three groups:

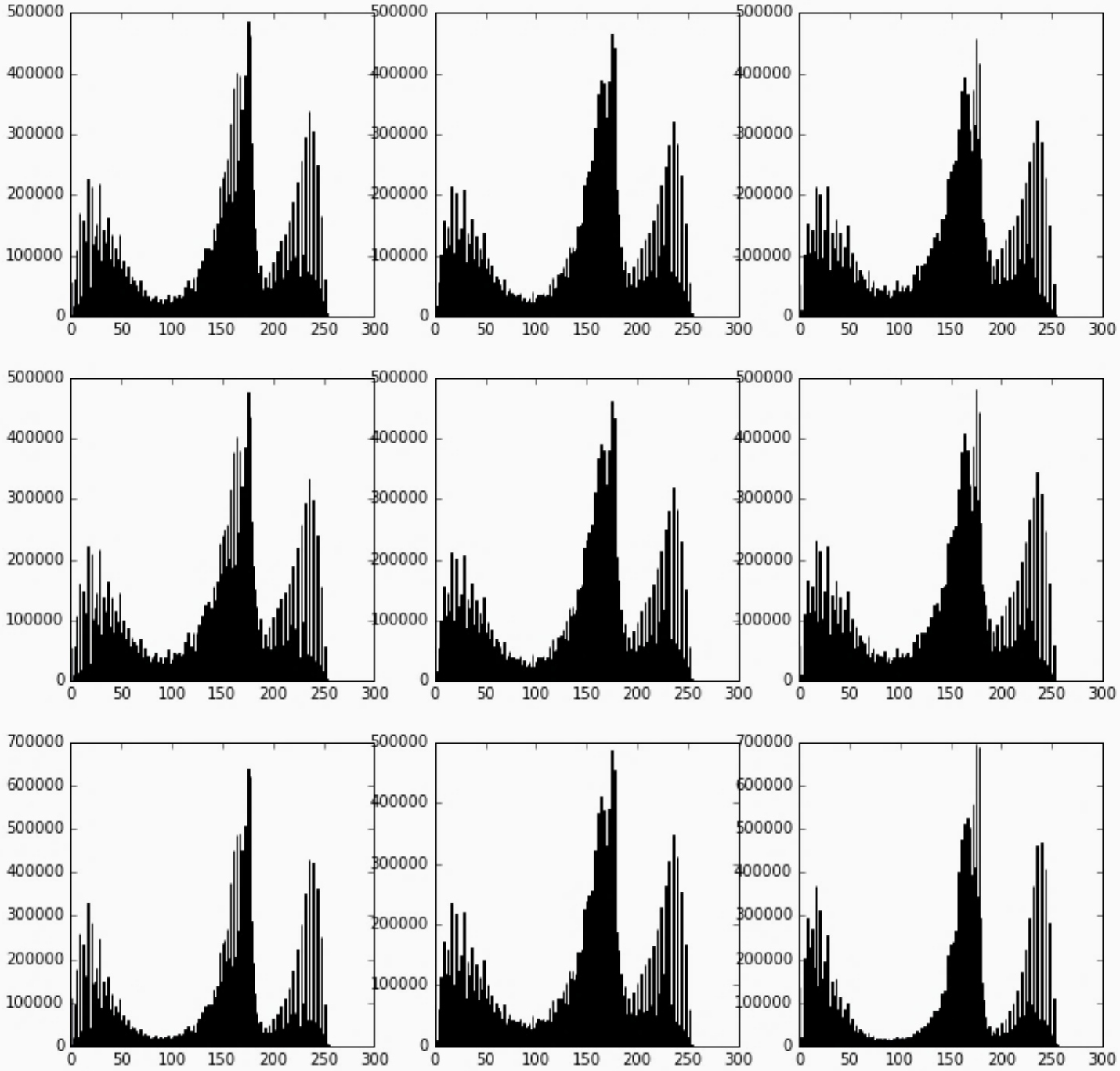
- The majority of the pixels have values at or near zero. These values represent regions of the image with little or no gradient.
- There is a group of pixels with values between 150 and 180, which likely represent the noise seen in the images of the gradient.
- The highest pixel values, above 220, are the edges of the image.

Applying the Sobel Edge Detection Algorithm

To illustrate this difference pre-processing has on the algorithm output values, the Sobel edge detection algorithm is applied to the original carrot images:

- When compared to the edges computed from the prepared images, the edges of the carrots from the raw images are less distinct.
- When compared to the edges computed from the prepared images the edges of the carrots from the raw images are more noisy.




500000
400000
300000
200000
100000
0
0 50 100 150 200 250 300
700000
600000
500000
400000
300000
200000
100000
0

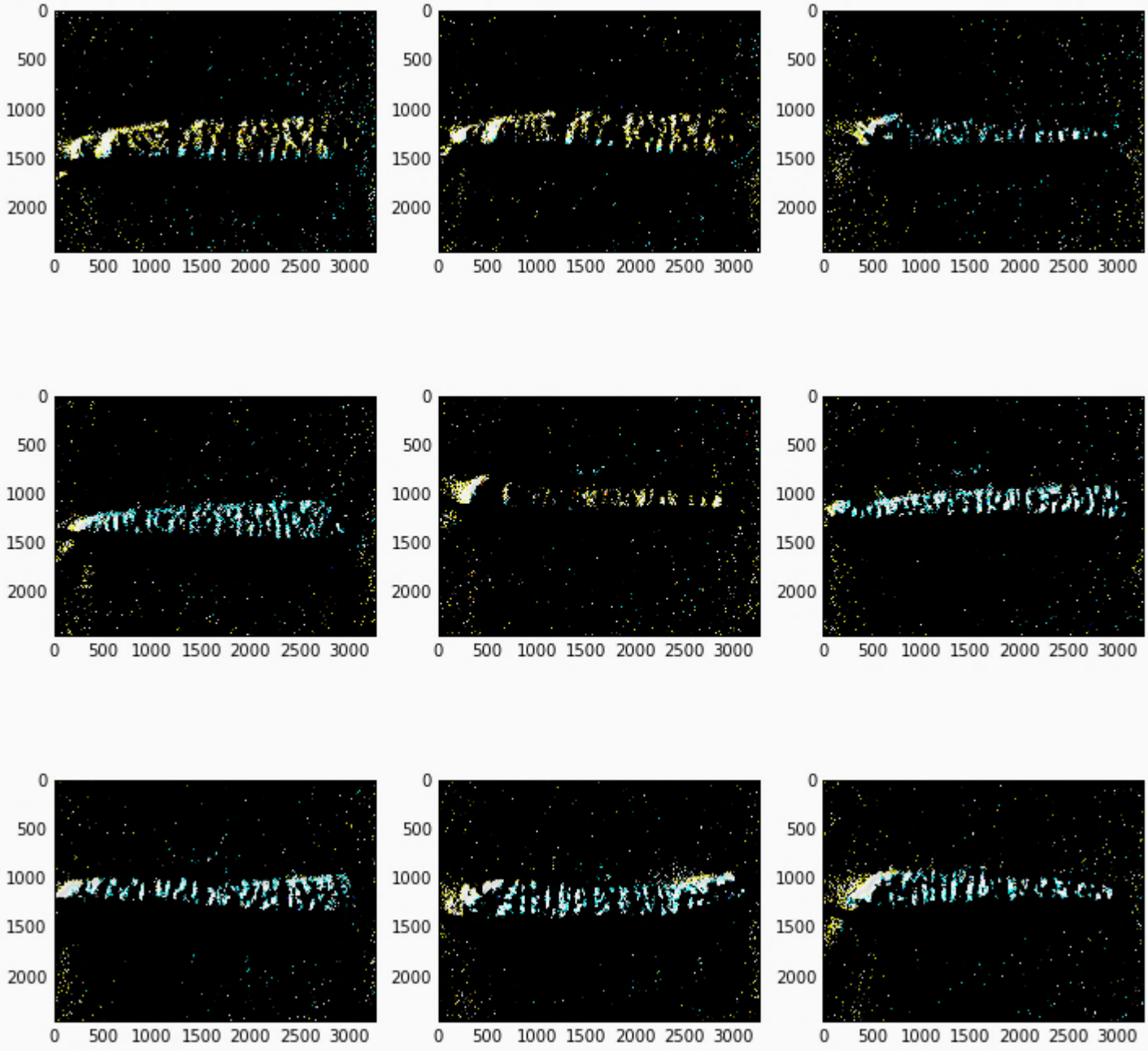
Segmentation

Image segmentation is a process of segregating certain regions or segments. **Thresholding** of pixel values on the image is a simple segmentation algorithm.

The code below thresholds the pixel values in each image in a list. All pixel values below the threshold are set to zero. The code executes an image threshold at a pixel value of 200.

```
def threshold(im_list, thresh = 200):
    import numpy as np
    out = []
    for image in im_list:
        image[np.where( image < thresh )] = 0
        out.append(image)
    return out
carrot_filter = threshold(carrot_filter)
plot_carrot(carrot_filter)
```

The edges of the carrots remain and are more distinctive than before **thresholding**. However, there are shadows in the image which have also been enhanced.

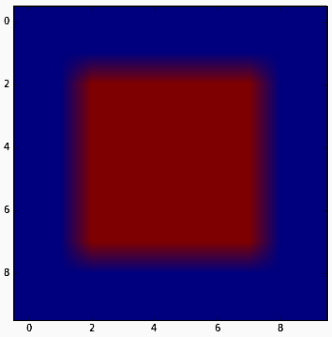


Corner Detection

Another example of a feature extraction algorithm is **corner detection**. In simple terms, the **Harris Corner Detection Algorithm** locates regions of the image with large changes in pixel values in all directions. These regions are said to be corners. The Harris corner detector is paired with the **corner_peaks** method. This operator filters the output of the Harris algorithm, over a patch of the image defined by the span of the filters, for the most likely corners.

As a simple example of corner detection, the algorithm is applied to a square shape. The code below creates and displays a matrix containing a square:

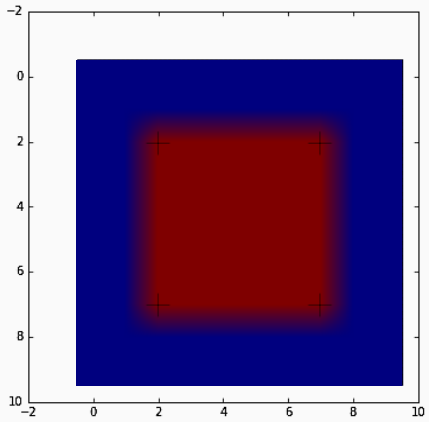
```
import numpy as np
square = np.zeros([10, 10])
square[2:8, 2:8] = 1
plot_im(square)
```



The code below, applies the Harris corner detector and the corner_peaks algorithm to the square:

```
from skimage.feature import corner_harris, corner_peaks
crn = corner_peaks(corner_harris(square), min_distance=1)
crn
```

```
Out[138]: array([[2, 2],
                 [2, 7],
                 [7, 2],
                 [7, 7]])
```



The corner detector has located four corners.

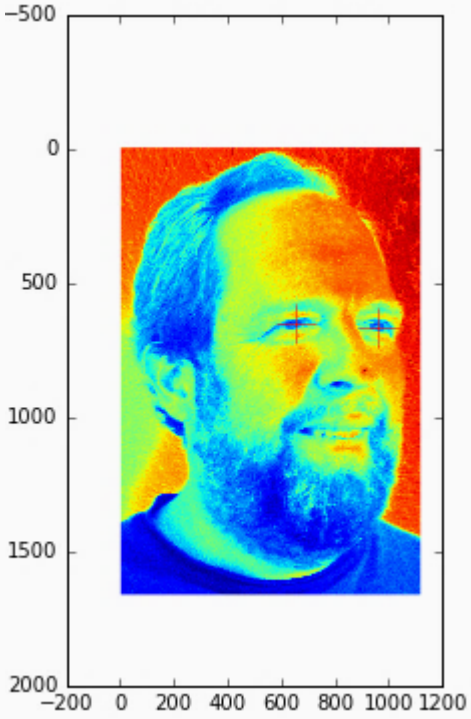
The code below plots the corners on top of the image:

```
def plot_harris(im, harris, markersize = 20, color = 'red'):
    import matplotlib.pyplot as plt
    import numpy as np
    fig = plt.figure(figsize=(8, 6))
    fig.clf()
    ax = fig.gca()
    ax.imshow(np.array(im).astype(float))
ax.plot(harris[:, 1], harris[:, 0], 'r+', color
 = color, markersize=markersize)
    return 'Done'
plot_harris(square, crn, color = 'black')
```

The four corners of the square have been correctly detected.

Next, applies the Harris corner detection algorithm to detect corners in the face image. Notice the span of the corner-peaks filter is now set to 4 pixels:

```
def corner_harr(im, min_distance = 4):
    from skimage.feature import corner_harris, corner_peaks
    mag = corner_harris(im)
    return corner_peaks(mag, min_distance = min_distance)
harris = corner_harr(steve, min_distance = 4)
plot_harris(steve, harris)
```



The corner detector has located the two eyes.

Image Morphology

Morphology methods are often used to enhance image features. Using these methods, a set of features can be cleaned and made more consistent.

Erosion

The first morphology method applied is erosion. **Erosion** is a fundamental operation and forms the basis of some other morphology operators. An erosion operator removes or erodes pixels at the edge of objects, by setting the values to zero.

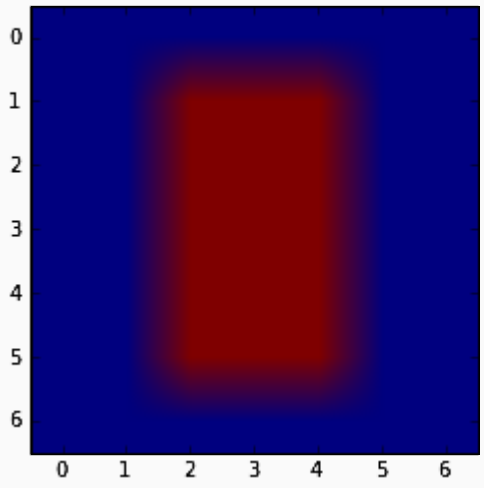
The code below creates a simple image containing a rectangle:

```
import numpy as np
from skimage import morphology
import matplotlib.pyplot as plt
a = np.zeros((7,7), dtype=np.int)
a[1:6, 2:5] = 1
a
```

```
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

```
plt.imshow(a)
```

```
<matplotlib.image.AxesImage at 0x7f774254c190>
```

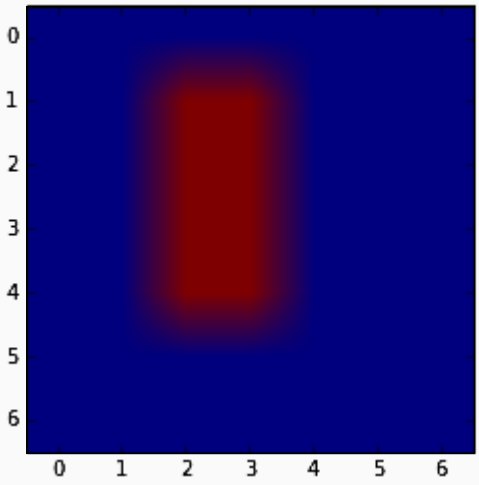


The code below erodes the image using a square shaped 2x2 operator:

```
a_erosion = morphology.binary_erosion(a, np.ones((2,2))).astype(np.uint8)
print(a_erosion)
plt.imshow(a_erosion)
```

```
[[0 0 0 0 0 0 0]
 [0 0 1 1 0 0 0]
 [0 0 1 1 0 0 0]
 [0 0 1 1 0 0 0]
 [0 0 1 1 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
```

```
<matplotlib.image.AxesImage at 0x7f77415d3f90>
```



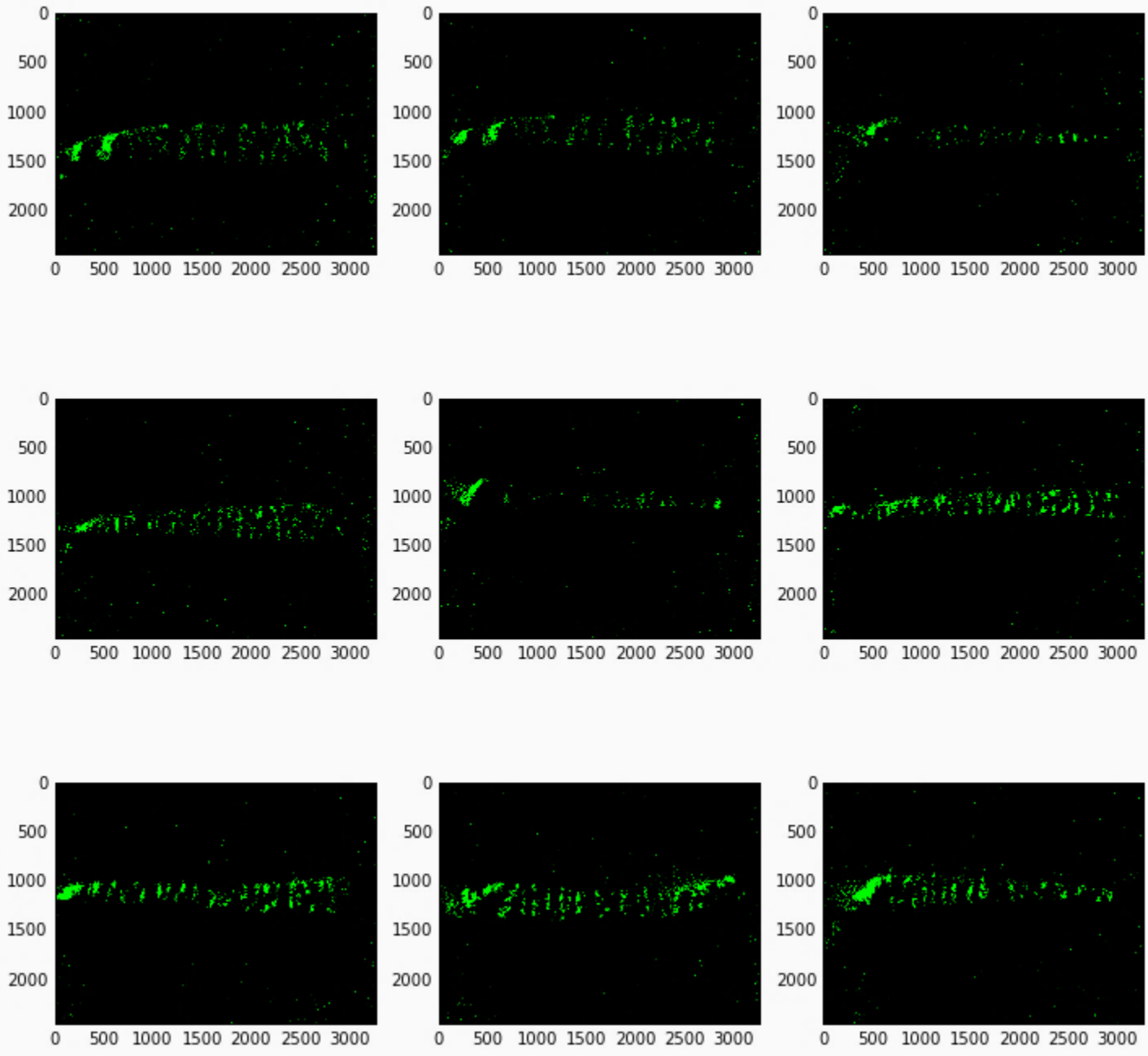
The erosion operator has reduced the size of the rectangle by one pixel in each dimension.

Now, run the erosion operator against the results of the edge detection applied to the carrot images. Since the carrot image has color, represented by a n x m x 3-d array, a 3-d operator is used. This is opposed to the gray scale face image which is represented by a n x m x 1-d array.

The code in the cell below iterates over the list of images, applying the erosion operator to each:

```
def im_erosion(im_list, structure = (4,4,3)):
    from scipy.ndimage import morphology
    import numpy as np
    out = []
    for image in im_list:
        out.append(morphology.binary_erosion(image,structure=np.ones(structure)))
    return out
carrot_erosion = im_erosion(carrot_filter, structure = (2,2,3))
plot_carrot(carrot_erosion)
```

Comparing images to the images resulting from applying edge detection and segmentation. The features of the carrots themselves are thinner. Additionally, the noise is less noticeable.



Dilation

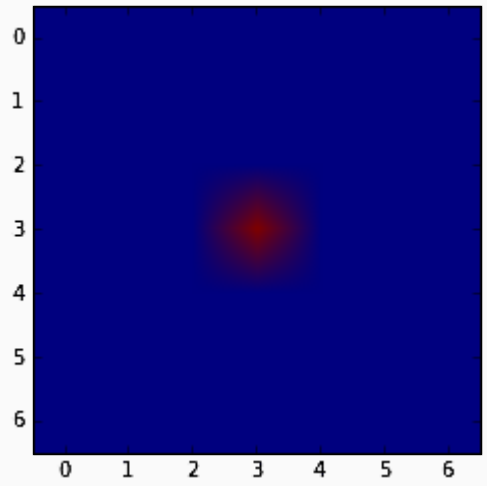
Dilation is another fundamental morphology operator. A **dilation** operator accretes pixels to image features. In effect, the image features are expanded or thickened.

The code below creates an image with a single positive pixel:

```
a = np.zeros((7,7), dtype=np.int)
a[3, 3] = 1
a
```

```
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

```
plt.imshow(a)
```

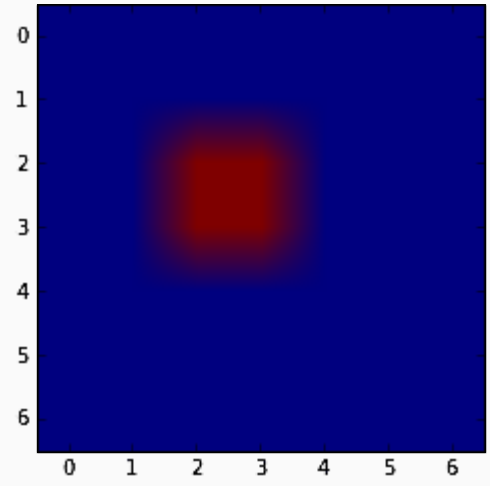


As before, a 2x2 square operator is used and applied to a diamond shaped dilation operator against the simple image created by executing the code below:

```
a_dilation = morphology.binary_dilation(a,
np.ones((2,2))).astype(np.uint8)
print(a_dilation)
plt.imshow(a_dilation)
```

```
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 1 1 0 0 0]
 [0 0 1 1 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
```

The image has accreted a row and column of non-zero pixels on the upper and left edges.



Opening

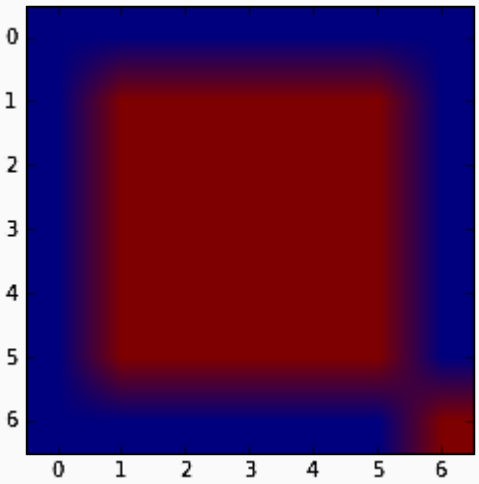
Opening is a morphological operation comprised of an erosion operator followed by a dilation operator. Opening is useful in cleaning certain types of noise from images. For example, opening tends to create better separated features in an image.

The code below creates an image with a square and a single non-zero pixel in the lower right corner:

```
a = np.zeros((7,7), dtype=np.int)
a[1:6, 1:6] = 1
a[6, 6] = 1
a
```

```
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 0, 0, 0, 0, 0, 1]])
```

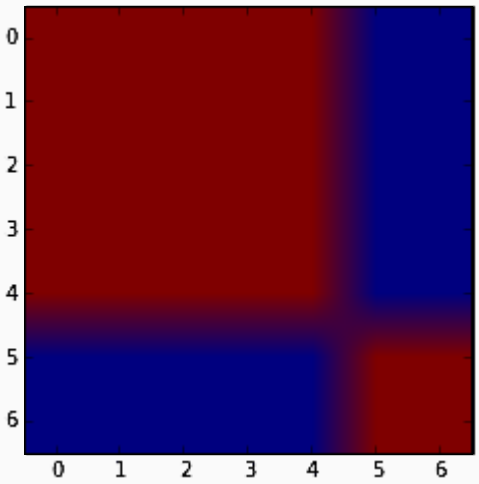
```
plt.imshow(a)
```



Apply the morphological opening operator to the image:

```
a_open = morphology.binary_opening(a, np.ones((2,2))).astype(np.uint8)
print(a_open)
plt.imshow(a_open)
```

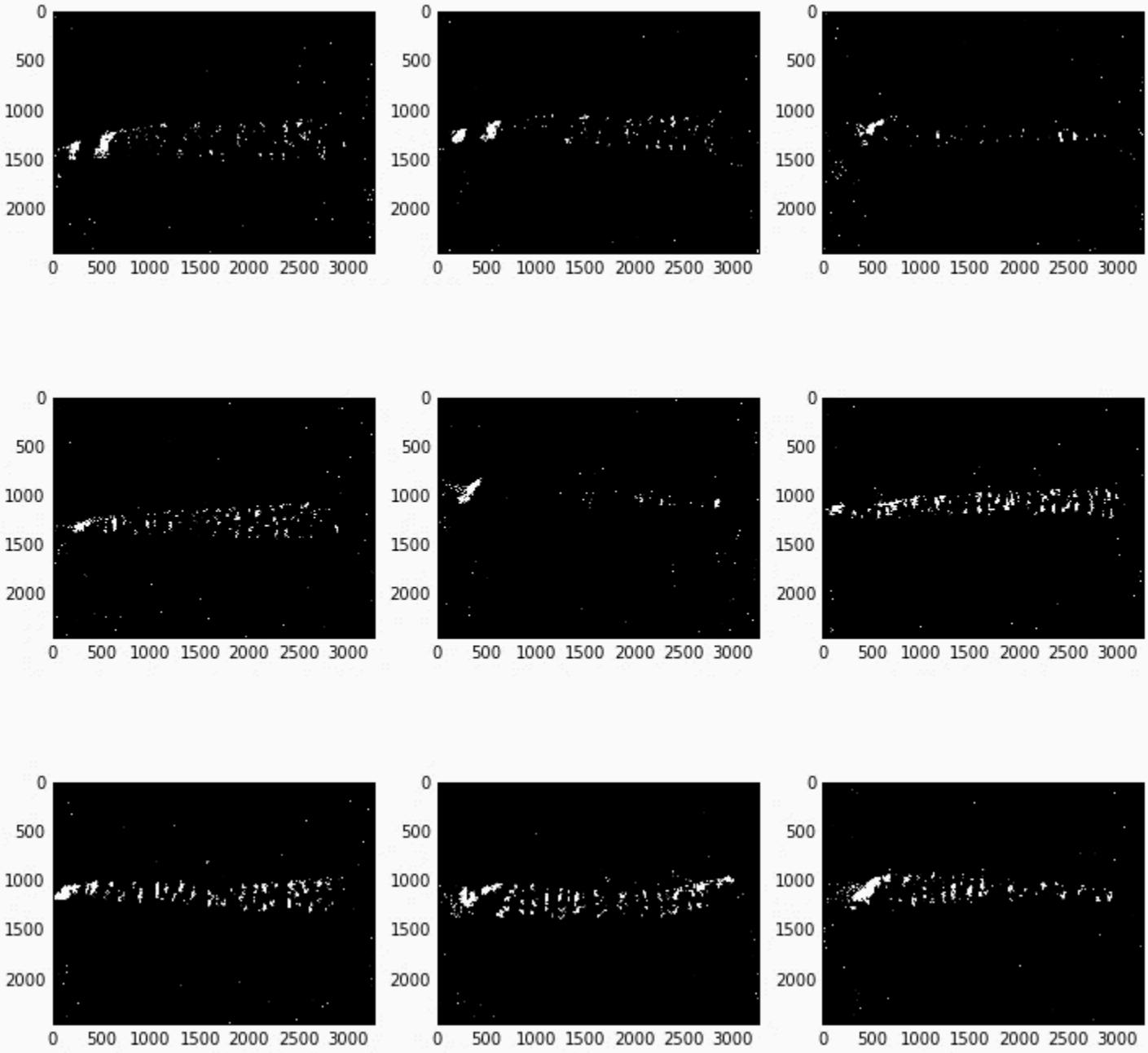
```
[[1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [0 0 0 0 0 1 1]
 [0 0 0 0 0 1 1]]
```



Compare the result to the original image. The pixel in the lower right has been set to zero. The image of the square retains its original dimensions. In effect, the non-zero pixel in the lower right corner has been filtered as though it was noise. Retaining the dimensions of the square is the result of following the opening operator with a dilation operator.

The code below applies the opening operator to the list of carrot edge images. The operator shape is asymmetric. Since the carrot edge features to be enhanced are elongated, mostly in the vertical (y) direction, an operator longer than it is wide is chosen. Apply the opening operator as follows:

```
def im_open(im_list, structure = (2,2,1)):
    from scipy.ndimage import morphology
    import numpy as np
    out = []
    for image in im_list:
        out.append(morphology.binary_opening(image,structure=np.ones(structure)))
    return out
carrot_open = im_open(carrot_filter, structure = (2,4,3))
plot_carrot(carrot_open)
```

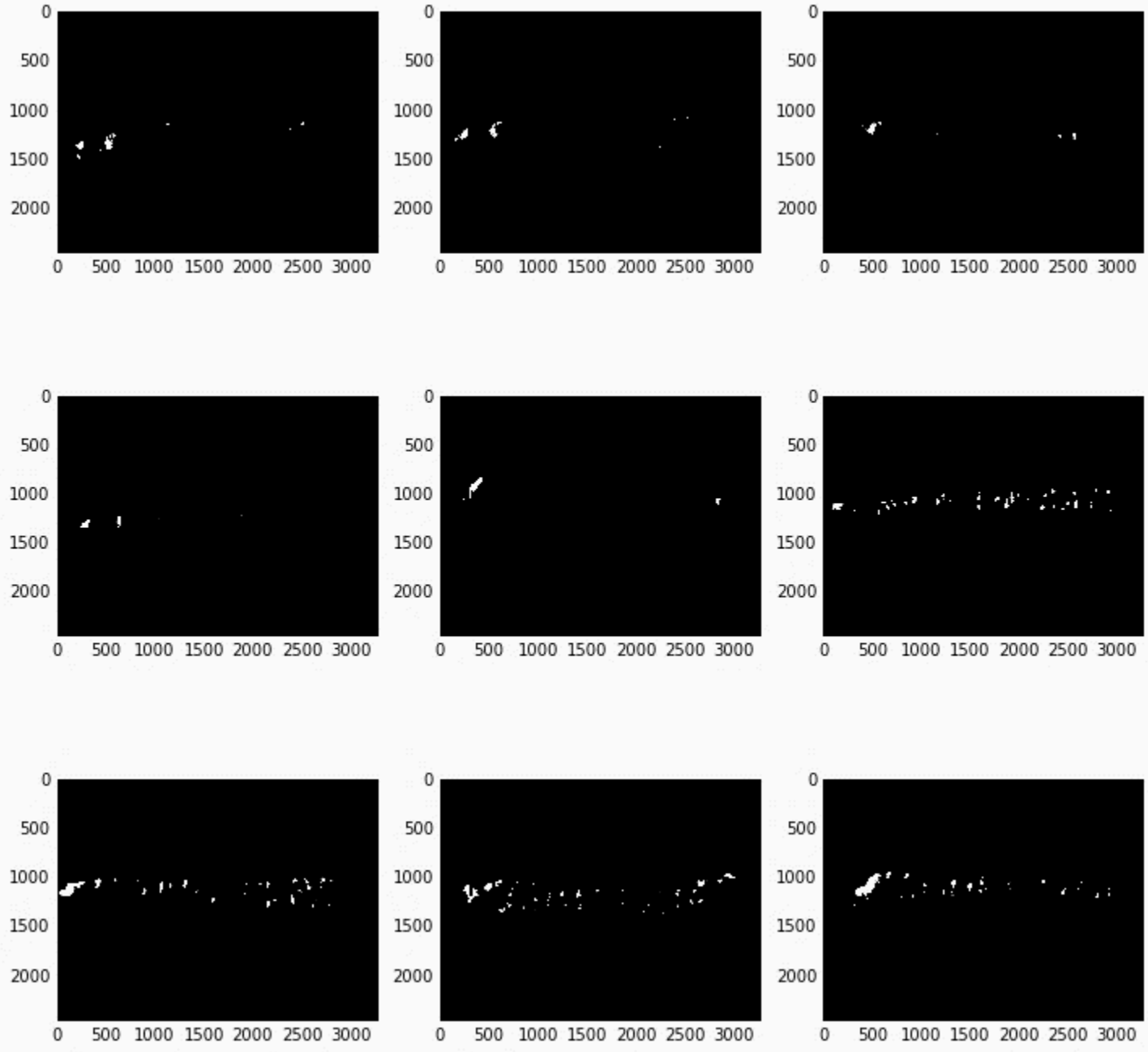


Notice much of the noise has been eliminated by applying the opening operator. Despite the reduction in noise, many essential features of the carrot edges have been retained. However, the features are thinner.

Changing the Operator Shape for Opening

To see this difference in operator effects, apply the opening operator with a shape defined by **(8,8,3)**. Compare the result to the carrot edge images created with an operator shape of **(2,4,3)**:

```
carrot_open = im_open(carrot_filter, structure = (8,8,3))
plot_carrot(carrot_open)
```



- The carrot edge features are better preserved by the smaller opening operator.
- The carrot edge feature created by the larger opening operator has less noise.

Closing

Closing is a morphological operation comprised of a dilation operator followed by an erosion operator, the opposite order of an opening operation. Like the opening operator, the closing operator can be useful in reducing noise when extracting features from images.

The code below creates a square with a hole in the middle:

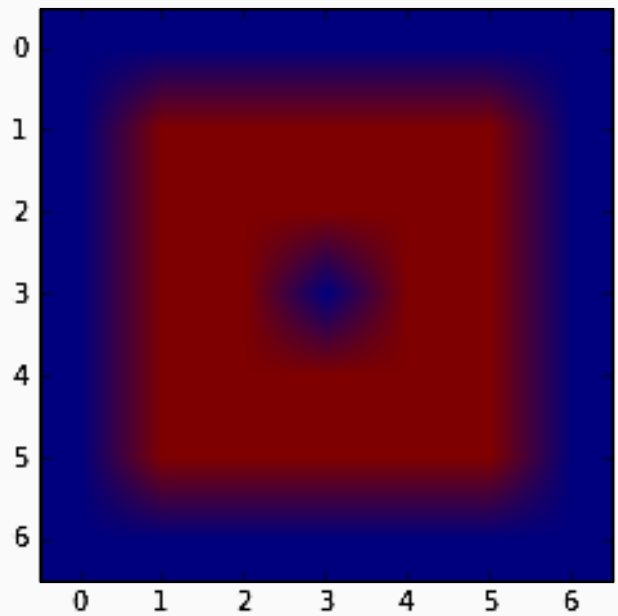
```
a = np.zeros((7,7), dtype=np.int)
a[1:6, 1:6] = 1
a[3,3] = 0
plt.imshow(a)
```

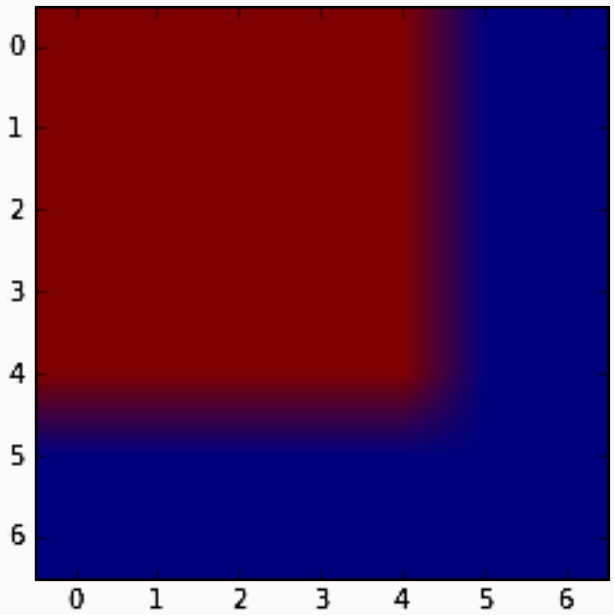
The code below applies the morphological closing operator to the image:

```
a_close = morphology.binary_closing(a, np.ones((2,2))).astype(np.uint8)
print(a_close)
plt.imshow(a_close)
```

```
[[1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
```

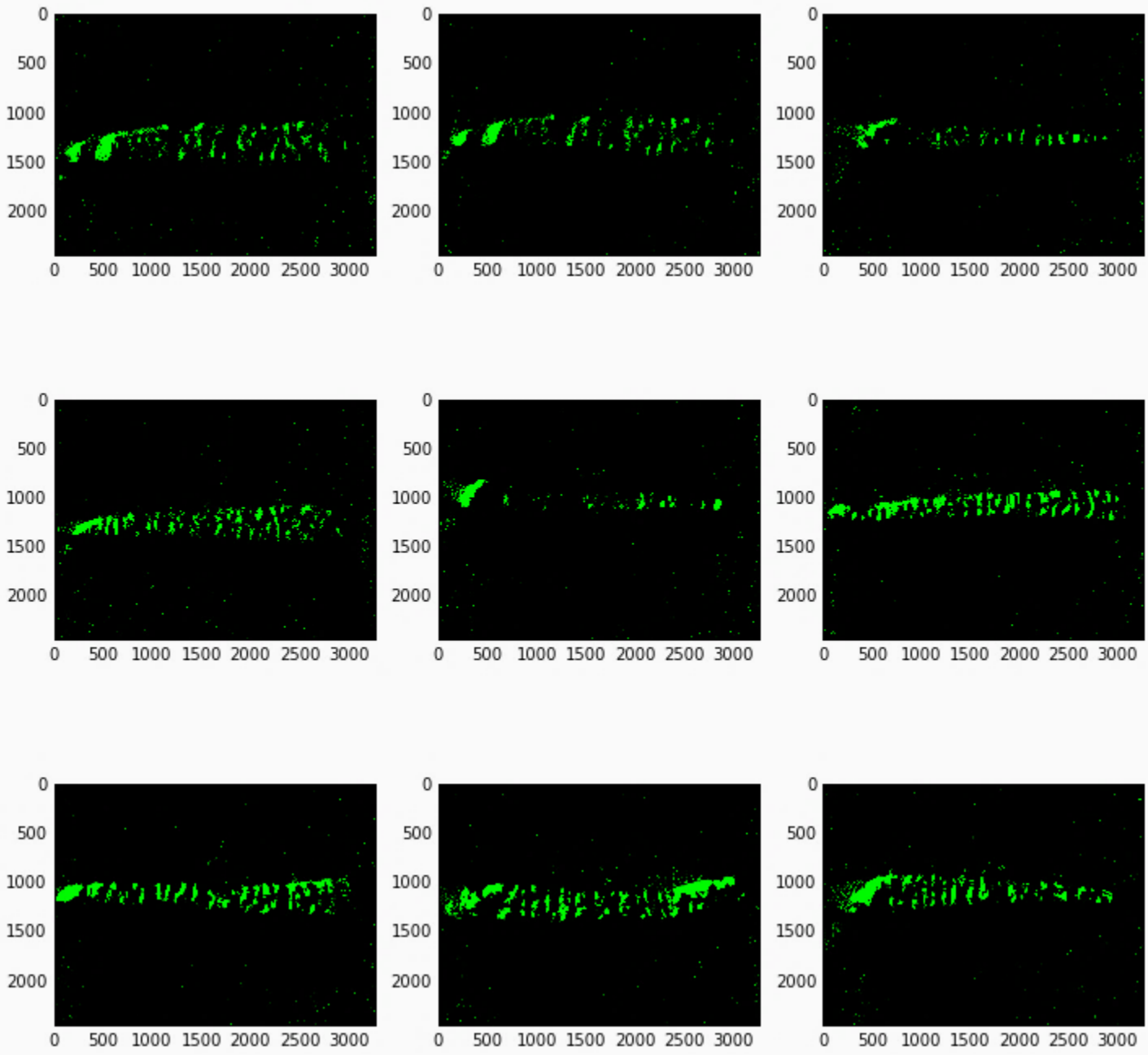
The hole in the square has been healed. The image of the square retains its original dimensions. Retaining the dimensions of the square results from the dilation operator with an erosion operator.





The code below applies the closing operator to the list of carrot images. The operator shape is asymmetric. Since the carrot edge features to be enhanced are elongated mostly in the vertical (y) direction, an operator longer than it is wide is chosen. The code applies the closing operator results:

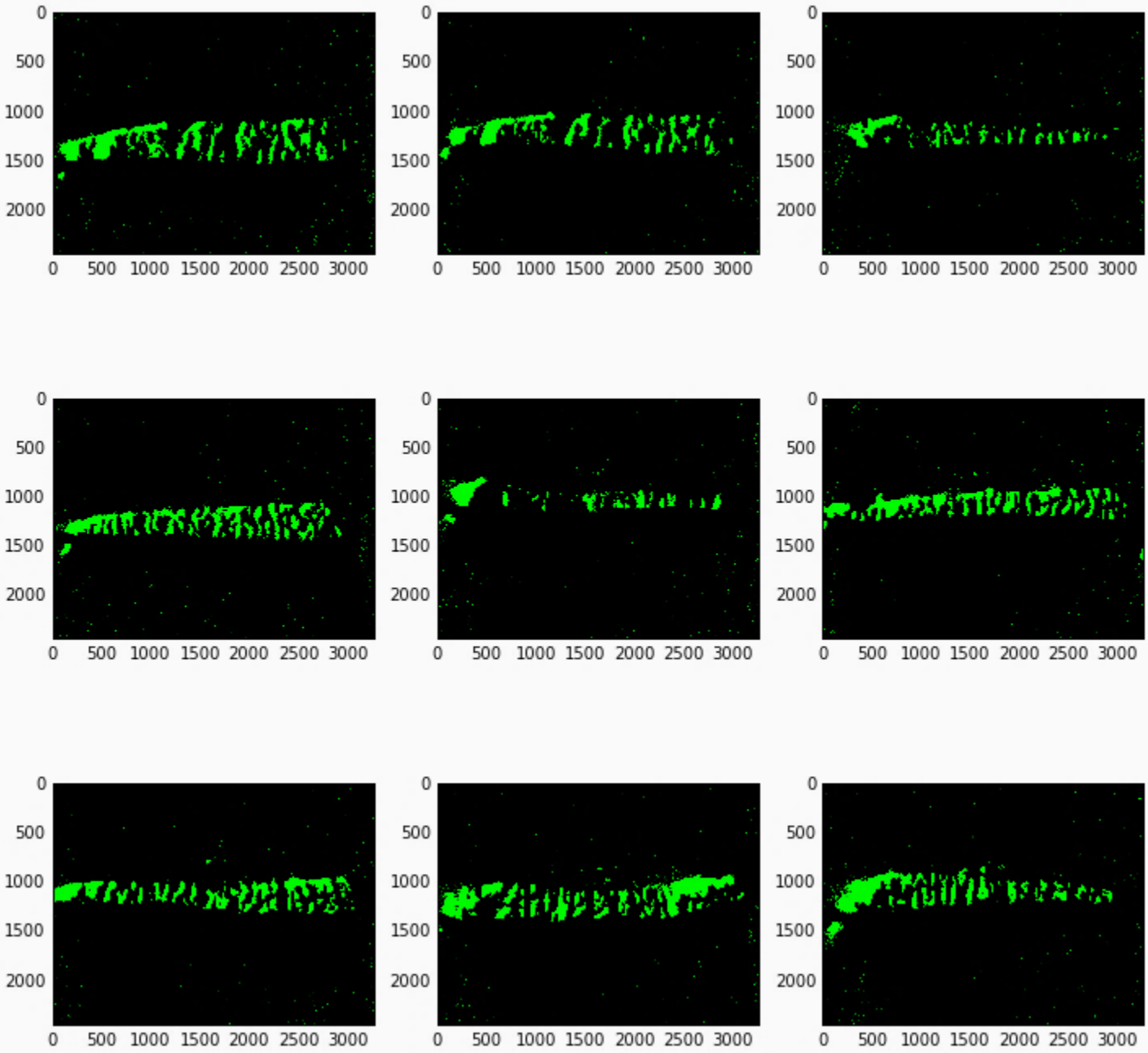
```
def im_close(im_list, structure = (2,2,3)):
    from scipy.ndimage import morphology
    import numpy as np
    out = []
    for image in im_list:
out.append(morphology.binary_closing(image,structure=np.ones(structure)))
    return out
carrot_close = im_close(carrot_erosion, structure = (2,6,3))
plot_carrot(carrot_close)
```



Comparing the images to the original images of the carrot edge features after segmentation. Much of the noise has been reduced by applying the closing operator. Despite the reduction in noise nearly all the essential features of the carrot edges are retained.

Change the Operator Shape for Closing

Changing the operator shape affects the results of the closing operation. To see this difference, the closing operator with a shape defined by **(6,12,3)** is applied to the carrot edge images. The result is compared to the result created with an operator shape of **(2,6,3)**:



- The carrot edge features better preserved by the larger closing operator.
- The carrot edge feature created by the larger closing operator has less noise.
- Of the four operators you have tried, a) **opening (2,4,3)**, b) **opening (8,8,3)**, c) **closing (2,6,3)** and **closing (6,12,3)**, it appears the first **opening (2,4,3)** did the best job reducing noise while preserving the carrot edge features.

Summary

This lab performed a steps to create images with features used in machine learning models:

- Loaded and explored the properties of the images.
- Equalized the image histograms.
- Plotted images.
- Added noise (pre-whitened) and filtered images.
- Extracted features from the images.
- Applied morphological operators to the image features.