Question 1

a) Write a program to show the minimum and the maximum pixel values of an 8 bits/pixel grayscale image. Convert grayscale image to a binary image using threshold (Tth) operation where `Tth = (minimum pixel value + maximum pixel value) / 2`. Mathematically, `G(x, y) = 0 if f(x, y) ≤ (minimum gray value + maximum gray value) / 2; 1, otherwise`.

b) Do the same thresholding operation considering `Tth = 128`. `G(x, y) = 0 if f(x, y) ≤ 128; 1 otherwise`.

Highlight the differences in the two images obtained.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'b256',
    'ba256',
    'f256',
    'l256',
    'n256',
    'o256',
    'p256',
    'pap256',
    'z256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of (filename, image) tuples for the images
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append((filename, image))

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = -(-len(filenames) // cols)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, (filename, image) in enumerate(images):
    # Set subplot title as '"filename" (rows, cols)'
    axs[int(idx // cols), idx % cols].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[int(idx // cols), idx % cols].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save original image as .bmp file
```

```
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```

In [5]:

```
# Store the threshold images
thres_imgs = []
```

Section (a)

Display the maximum and minimum pixel values of each image.

Convert grayscale image to a binary image using threshold (Tth) operation where

`Tth = (minimum pixel value + maximum pixel value) / 2`.

Mathematically, `G(x, y) = 0 if f(x, y) ≤ (minimum gray value + maximum gray value) / 2; 1, otherwise`.

In [6]:

```
# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)
fig.suptitle("Threshold: average(min, max)", fontsize=18)

# Iterate for all images
for idx, (filename, image) in enumerate(images):
    # print('Image: "{}"\n'.format(filename))

    min_pixel = min([min(i) for i in image])    # minimum pixel value
    max_pixel = max([max(i) for i in image])    # maximum pixel value

    # print('Mininum pixel value: {}'.format(min_pixel))
    # print('Maximum pixel value: {}'.format(max_pixel))

    threshold = (int(min_pixel) + int(max_pixel)) // 2  # threshold
    # print('Threshold: {}'.format(threshold))

    '''
    Threshold image.

    Create a binary matrix of same shape as image.
    Each value is whether corresponding pixel value is higher than threshold.
    '''
    thres_img = (image > threshold) * 1

    # Add dictionary of filename, min pixel, max pixel, threshold image into list
    thres_imgs.append({
        'filename': filename,
        'threshold': threshold,
        'thres_avg': thres_img
    })

    # Set subplot title
    axs[int(idx // cols), idx % cols].set_title(
        'image: "{}"\npixel range: [{} - {}]\nthreshold: {}'.format(
            filename,
            min_pixel,
            max_pixel,
            threshold
        )
    )
    # Add subplot to figure plot buffer
    axs[int(idx // cols), idx % cols].imshow(
        thres_img,
        cmap='gray',
        vmin=0,
        vmax=1
    )

    # Save threshold image as .bmp file
    plt.imsave(
        path_out_conv + ext_out[1:] + '/' + filename + '_thres_avg' + ext_out,
        thres_img,
        cmap='gray',
        vmin=0,
        vmax=1
    )

    # Save pixel values of threshold image as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_thres_avg' + ext_inp,
        thres_img,
        fmt=' %d',
        newline=' \n'
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Save and display the figure
plt.savefig('threshold_avg.jpg')
plt.show()
```

Section (b)

Do the same thresholding operation considering `Tth = 128` .

`G(x, y) = 0 if f(x, y) ≤ 128; 1 otherwise`

In [7]:

```
# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)
fig.suptitle("Threshold: 128", fontsize=18)

# Iterate for all images
for idx, (filename, image) in enumerate(images):
    threshold = 128

    '''
    Threshold image.

    Create a binary matrix of same shape as image.
    Each value is whether corresponding pixel value is higher than threshold.
```

```
'''
thres_img = (image > threshold) * 1

# Add threshold image to corresponding dictionary in list of threshold images
thres_imgs[idx]['thres_128'] = thres_img

# Set subplot title
axs[int(idx // cols), idx % cols].set_title(
    'image: "{}"\npixel range: [{} - {}]'.format(
        filename,
        min_pixel,
        max_pixel
    )
)
# Add subplot to figure plot buffer
axs[int(idx // cols), idx % cols].imshow(
    thres_img,
    cmap='gray',
    vmin=0,
    vmax=1
)

# Save threshold image as .bmp file
plt.imsave(
    path_out_conv + ext_out[1:] + '/' + filename + '_thres_128' + ext_out,
    thres_img,
    cmap='gray',
    vmin=0,
    vmax=1
)
```

```
    # Save pixel values of threshold image as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_thres_128' + ext_inp,
        thres_img,
        fmt=' %d',
        newline=' \n'
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Save and display the figure
plt.savefig('threshold_128.jpg')
plt.show()
```

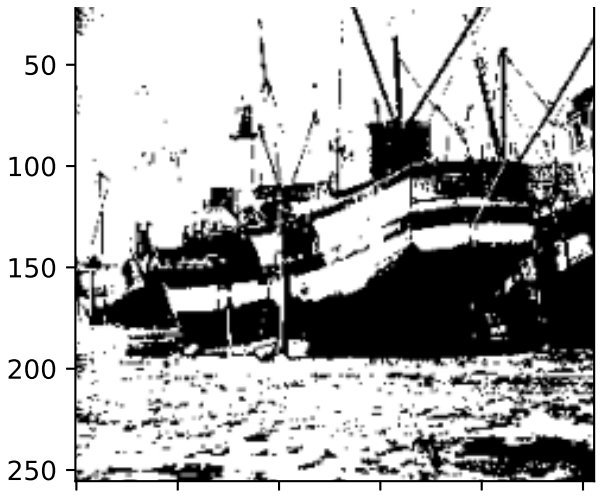
Threshold: 128

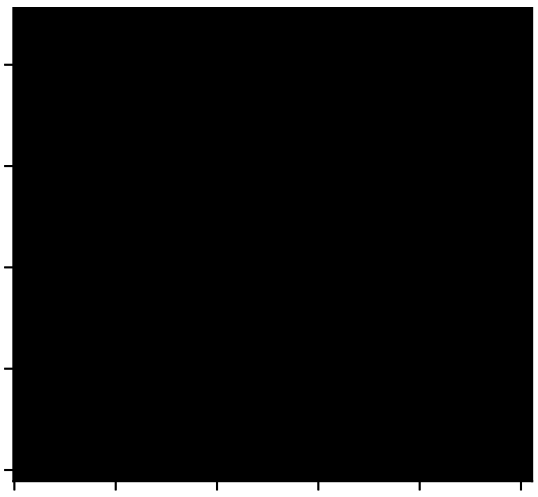
Compare images

In [8]:

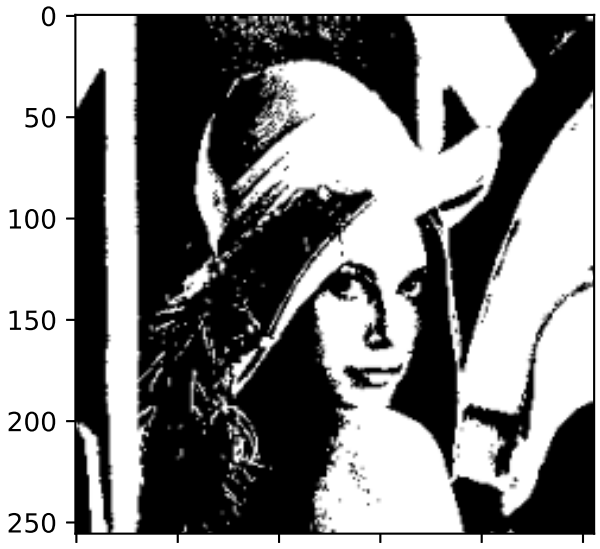
```
rows = len(thres_imgs)
cols = 3

# Create figure with len(thres_imgs) × 2 subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all threshold images
for idx, img_dict in enumerate(thres_imgs):
    # Generate binary difference matrix
    diff = abs(img_dict['thres_avg'] - img_dict['thres_128'])
    diff = 1 - ((diff == 0) * 1)

    # Set subplot title as 'threshold: $avg_value'
    axs[idx, 0].set_title('threshold: {}'.format(img_dict['threshold']))
    # Add subplot to figure plot buffer
    axs[idx, 0].imshow(
        img_dict['thres_avg'],
        cmap='gray',
```

```
        vmin=0,
        vmax=1
    )
    # Set subplot title as 'threshold: 128'
    axs[idx, 1].set_title('{}\nthreshold: 128'.format(
        '"{}"'.format(img_dict['filename'])
    ))
    # Add subplot to figure plot buffer
    axs[idx, 1].imshow(
        img_dict['thres_128'],
        cmap='gray',
        vmin=0,
        vmax=1
    )
    # Set subplot title as '"filename" (rows, cols)'
    axs[idx, 2].set_title('difference')
    # Add subplot to figure plot buffer
    axs[idx, 2].imshow(
        diff,
        cmap='gray',
        vmin=0,
        vmax=1
    )

    # Save difference image as .bmp file
    plt.imsave(
        path_out_conv + ext_out[1:] + '/' + img_dict['filename'] + '_diff' + ext_out,
        diff,
        cmap='gray',
        vmin=0,
        vmax=1
    )

    # Save pixel values of difference image as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + img_dict['filename'] + '_diff' + ext_inp,
        diff,
        fmt=' %d',
        newline=' \n'
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Save and display the figure
plt.savefig('difference.jpg')
plt.show()
```





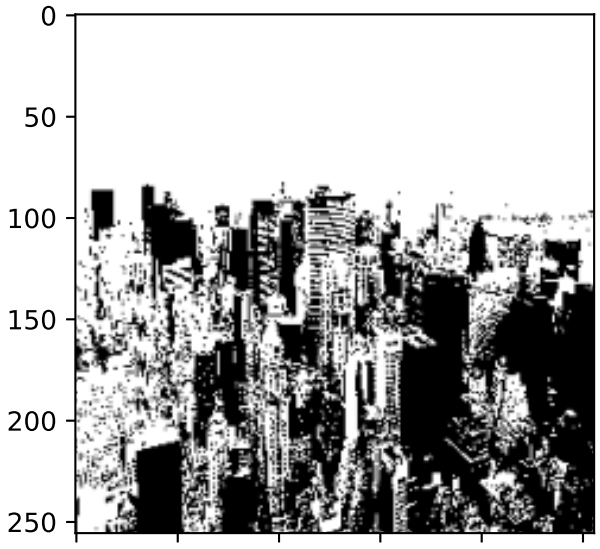
threshold: 133



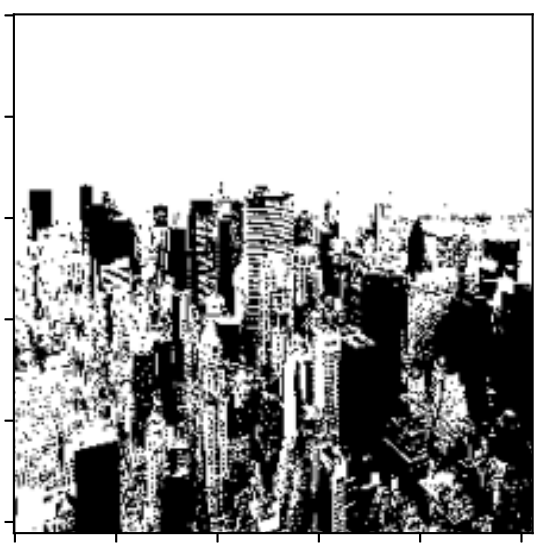
"l256"
threshold: 128

difference

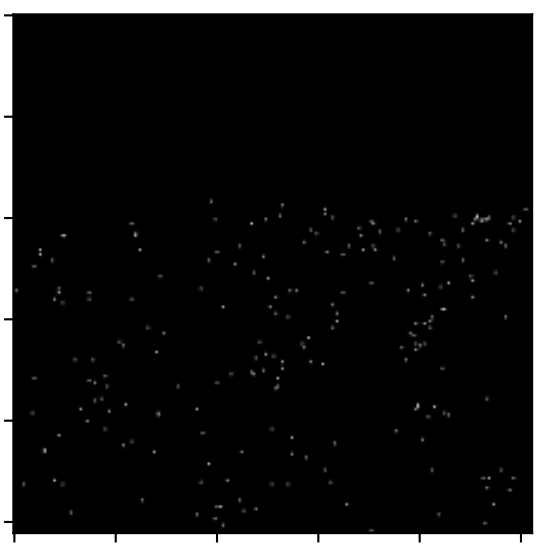
threshold: 127



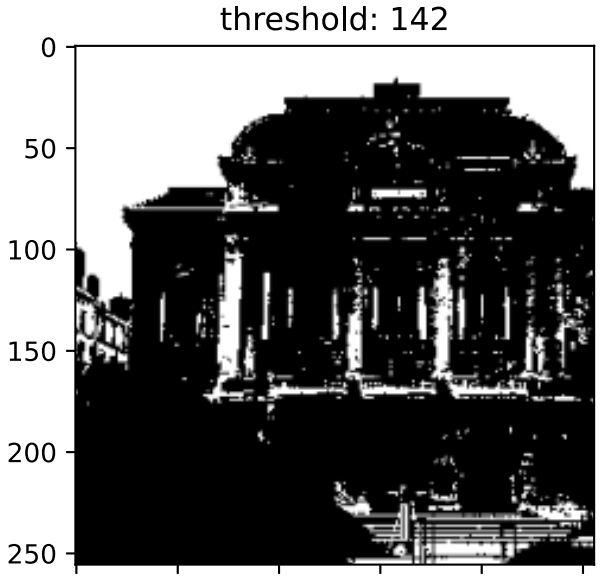
"n256"
threshold: 128



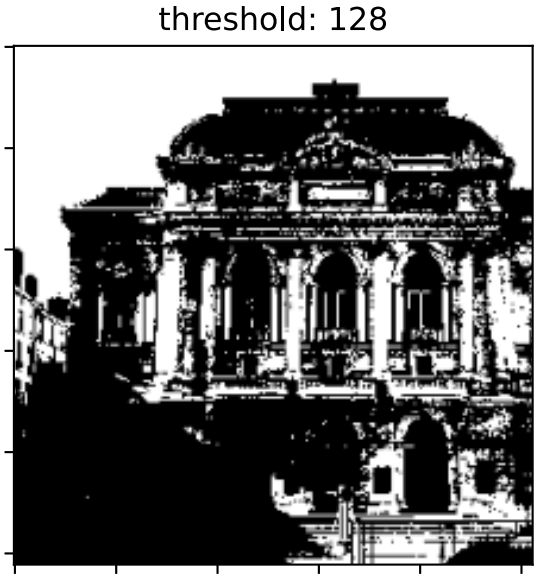
difference



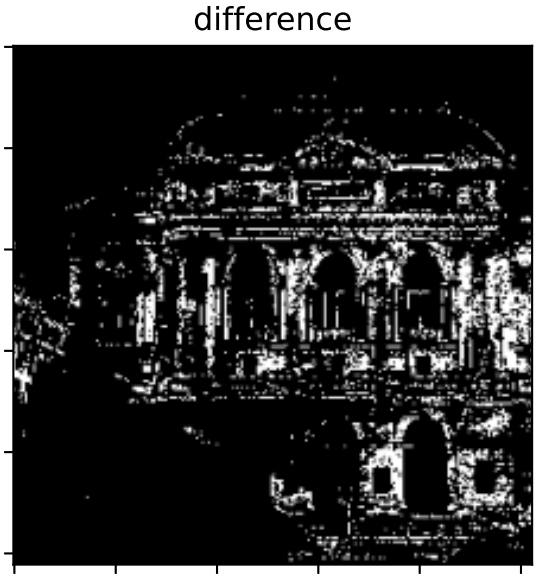
threshold: 142



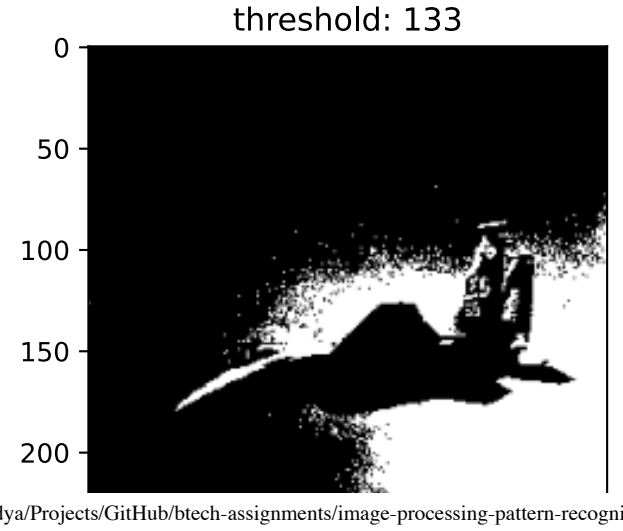
"o256"
threshold: 128



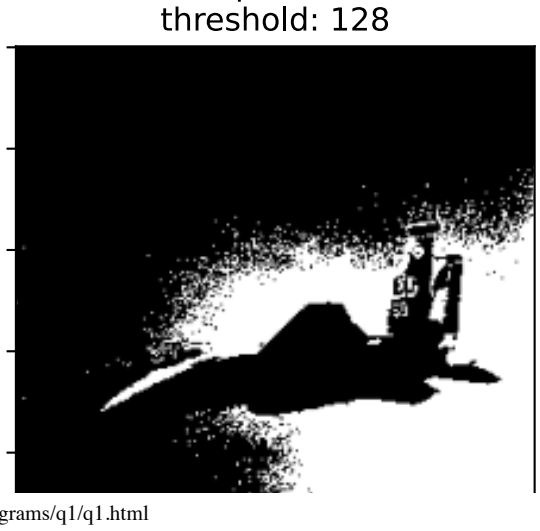
difference



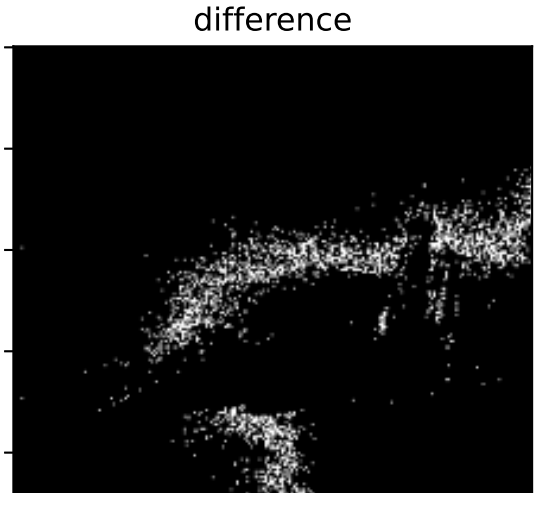
threshold: 133

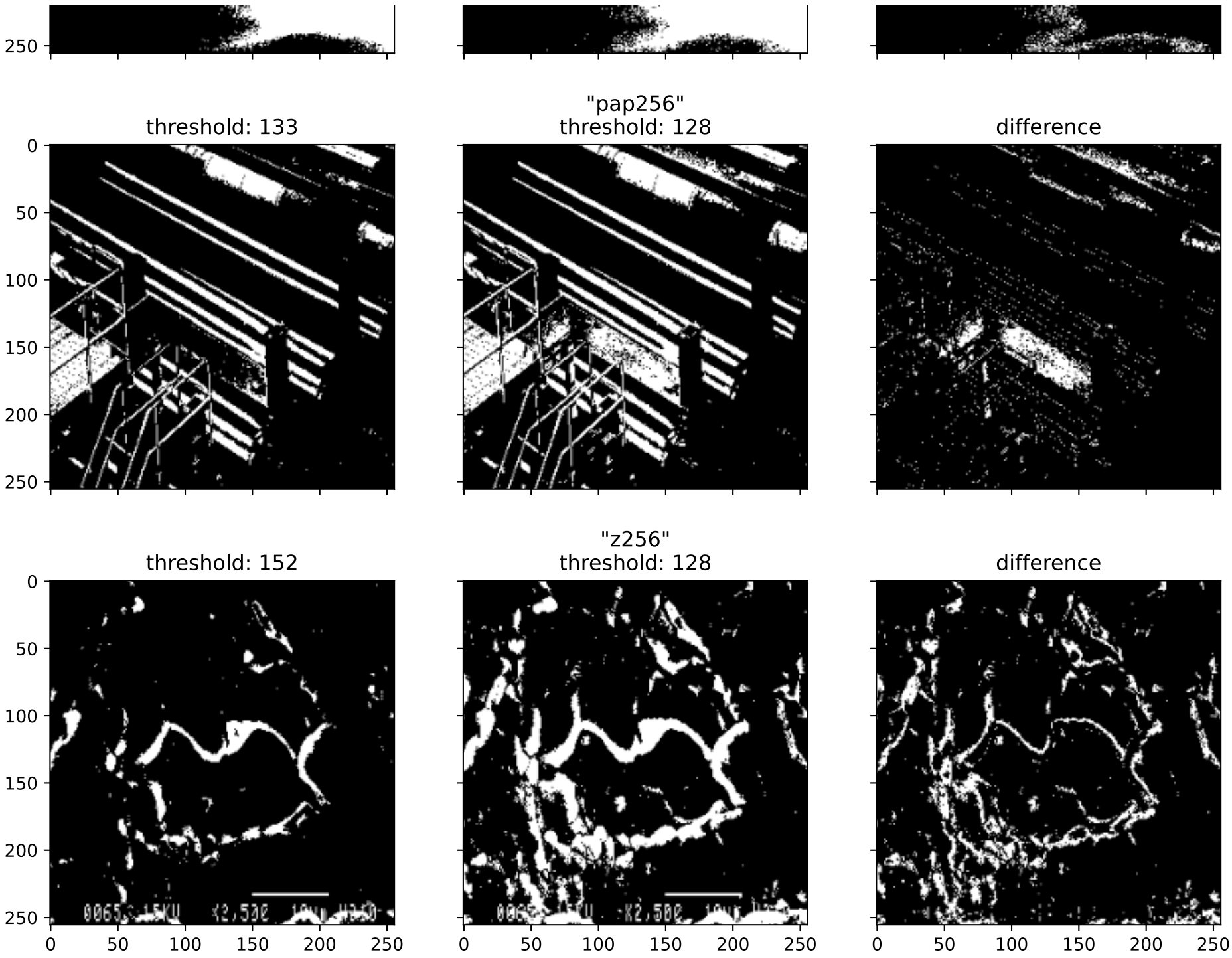


"p256"
threshold: 128



difference





Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 2

Write a program to implement down sampling of an image by a factor of 2 . Apply the same effect for 2 / 3 successive times and comment on visual content. Finally, write a program to upscale the down sampled image through interpolation and comment on visual quality of the image.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'f256',
    'l256',
    'o256',
    'p256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

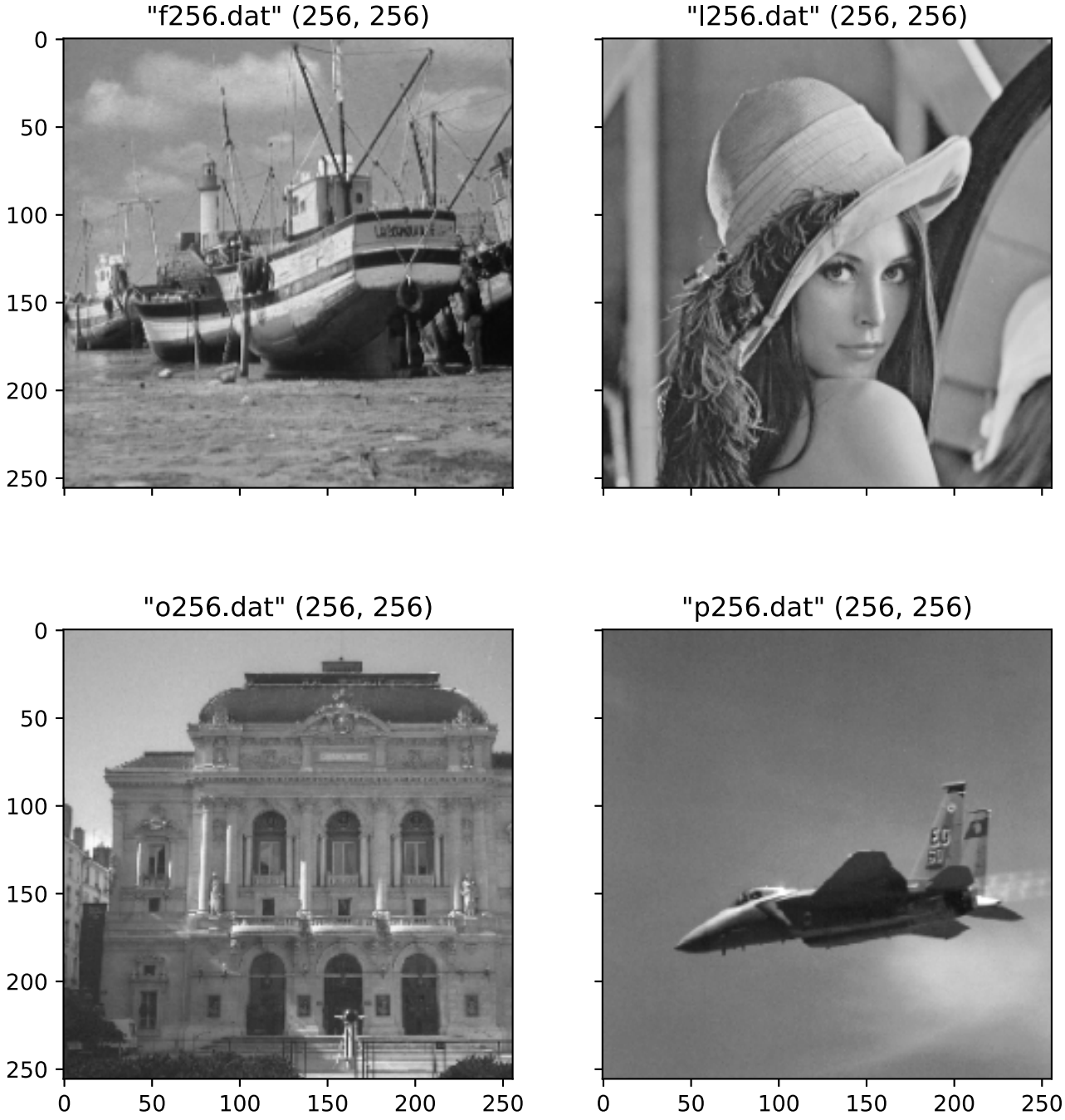
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 2
rows = -(-len(filenames) // cols)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[int(idx // cols), idx % cols].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[int(idx // cols), idx % cols].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
```

```
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



Downsampling

Implement down sampling by a factor of 2 .

Apply the same effect for 2 / 3 successive times and comment on visual content.

In [5]:
```
def downsample(image, multiplier):
    '''
    Downsamples an image.

    Averages pixel values in blocks of size = multiplier

    Parameters:
        image (array-like): Input image
        multiplier(int): Block size

    Returns:
        int: Downsampled image
    '''

    down_img = np.copy(image)
    height, width = down_img.shape

    for i in range(height)[::multiplier]:
        for j in range(width)[::multiplier]:
            step = multiplier // 2

            '''
            Average all pixel value in block
            '''
            sum = 0
            for r in range(i, i + multiplier):
                for c in range(j, j + multiplier):
                    sum = sum + int(down_img[r][c])
            avg_pix = sum // (multiplier ** 2)
```

```
            '''
            Assign all pixel values in block as average value
            '''
            for r in range(i, i + multiplier):
                for c in range(j, j + multiplier):
                    down_img[r][c] = avg_pix

    return down_img
```

In [6]:

```
def plot_downsampled(key, multiplier):
    '''
    Downsamples all images and plots them.

    Parameters:
        key (str): Key in image dict to access required input images
        multiplier (int): Block size

    Returns:
        None
    '''

    cols = 2
    rows = -(-len(filenames) // cols)

    # Create figure with rows × cols subplots
    fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
    fig.set_size_inches(4 * cols, 4.5 * rows)
    fig.suptitle('Downsampled: {} time ({}x)'.format(
        int(math.log(multiplier, 2)), multiplier),
        fontsize=18
    )

    # Iterate for all images
    for idx, image_dict in enumerate(images):
        filename = image_dict['filename']
        down_img = downsample(image_dict[key], multiplier)

        # Add image to dictionary
        images[idx]['down_{}x'.format(multiplier)] = down_img

        # Set subplot title
        axs[idx // cols, idx % cols].set_title('"{}"'.format(filename))
        # Add subplot to figure plot buffer
        axs[idx // cols, idx % cols].imshow(down_img, cmap='gray', vmin=0, vmax=255)

        # Save threshold image as .bmp file
        plt.imsave(
            path_out_conv + ext_out[1:] + '/' + filename + '_down_{}x'.format(multiplier) + ext_out,
            down_img,
            cmap='gray',
            vmin=0,
            vmax=255
        )

        # Save pixel values of threshold image as a 2D matrix in a .dat file
        np.savetxt(
            path_out_conv + ext_inp[1:] + '/' + filename + '_down_{}x'.format(multiplier) + ext_inp,
            down_img,
            fmt=' %d',
            newline=' \n'
        )

    # Hide x labels and tick labels for top plots and y ticks for right plots
    for ax in axs.flat:
        ax.label_outer()

    # Save and display the figure
    plt.savefig('downsampled_{}x.jpg'.format(multiplier))
    plt.show()
```

1st time (2x)

In [7]:

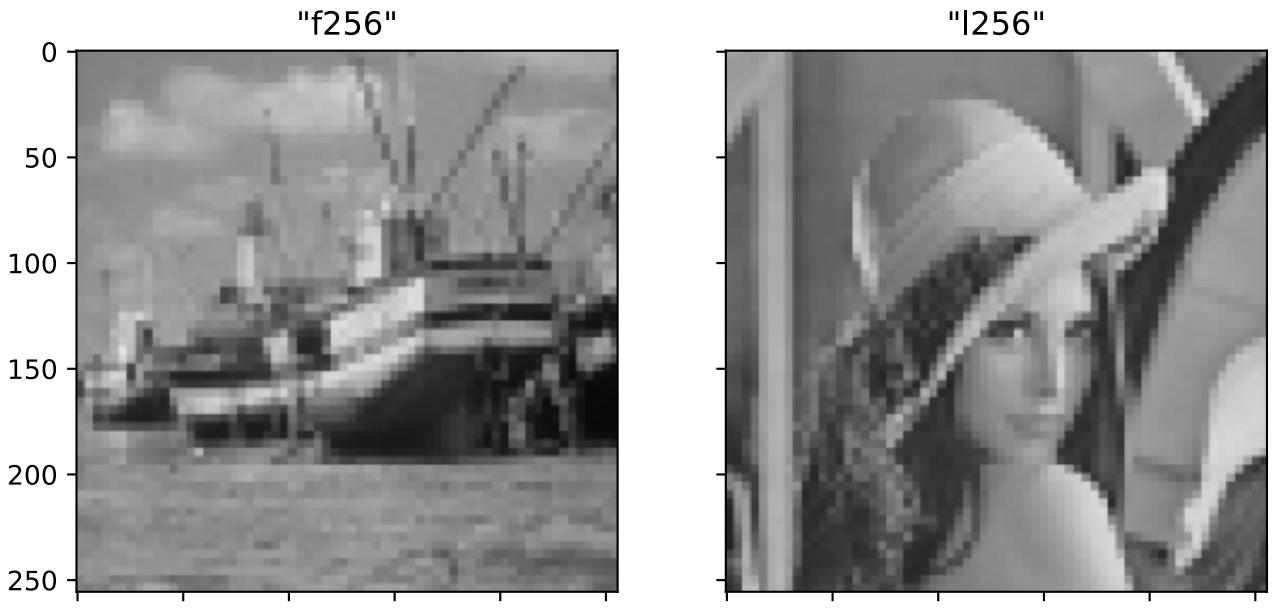
```
plot_downsampled('orig', 2)
```

Downsampled: 1 time (2x)

2nd time (4x)

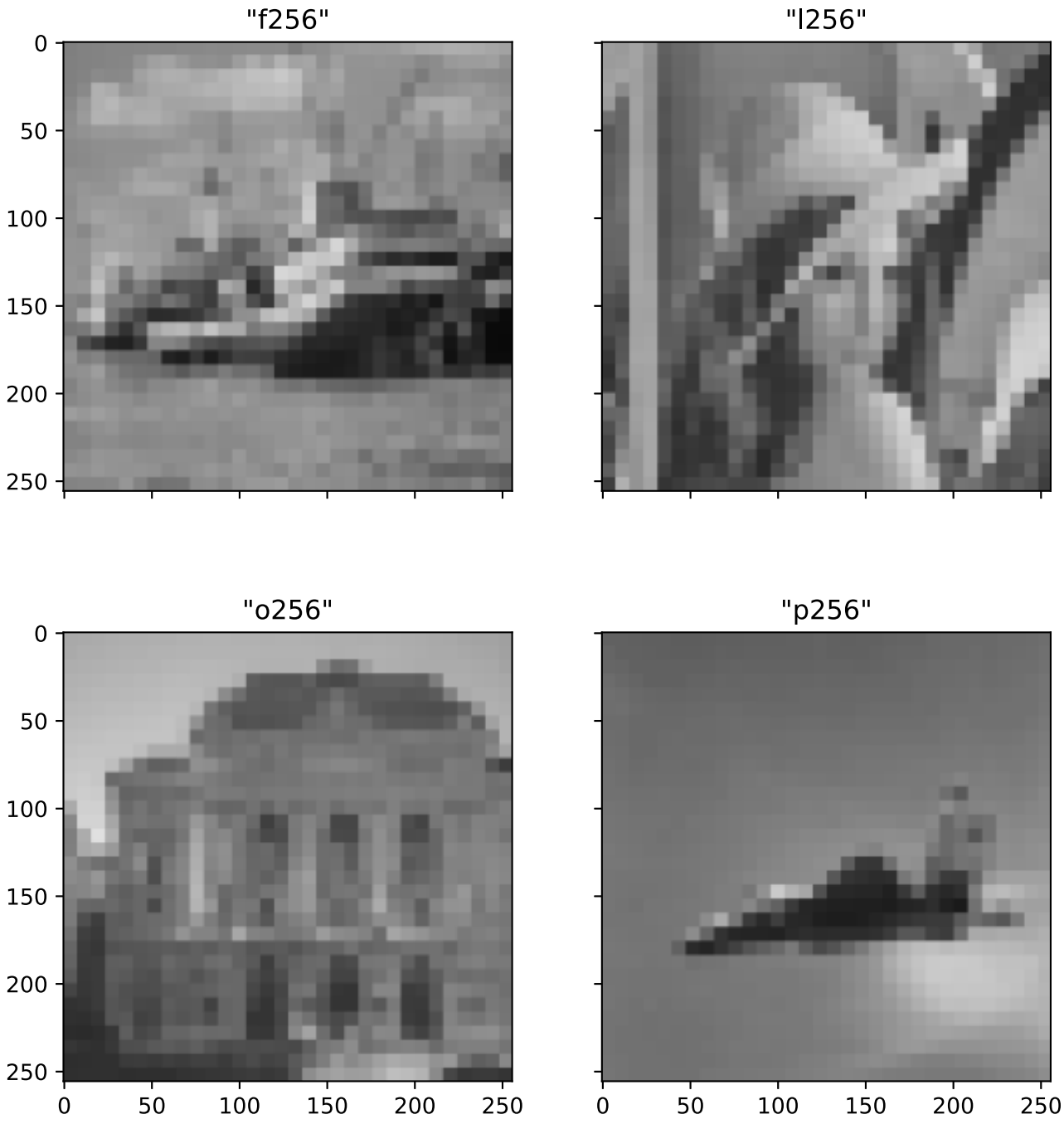
In [8]:
```
plot_downsampled('down_2x', 4)
```

Downsampled: 2 time (4x)



3rd time (8x)

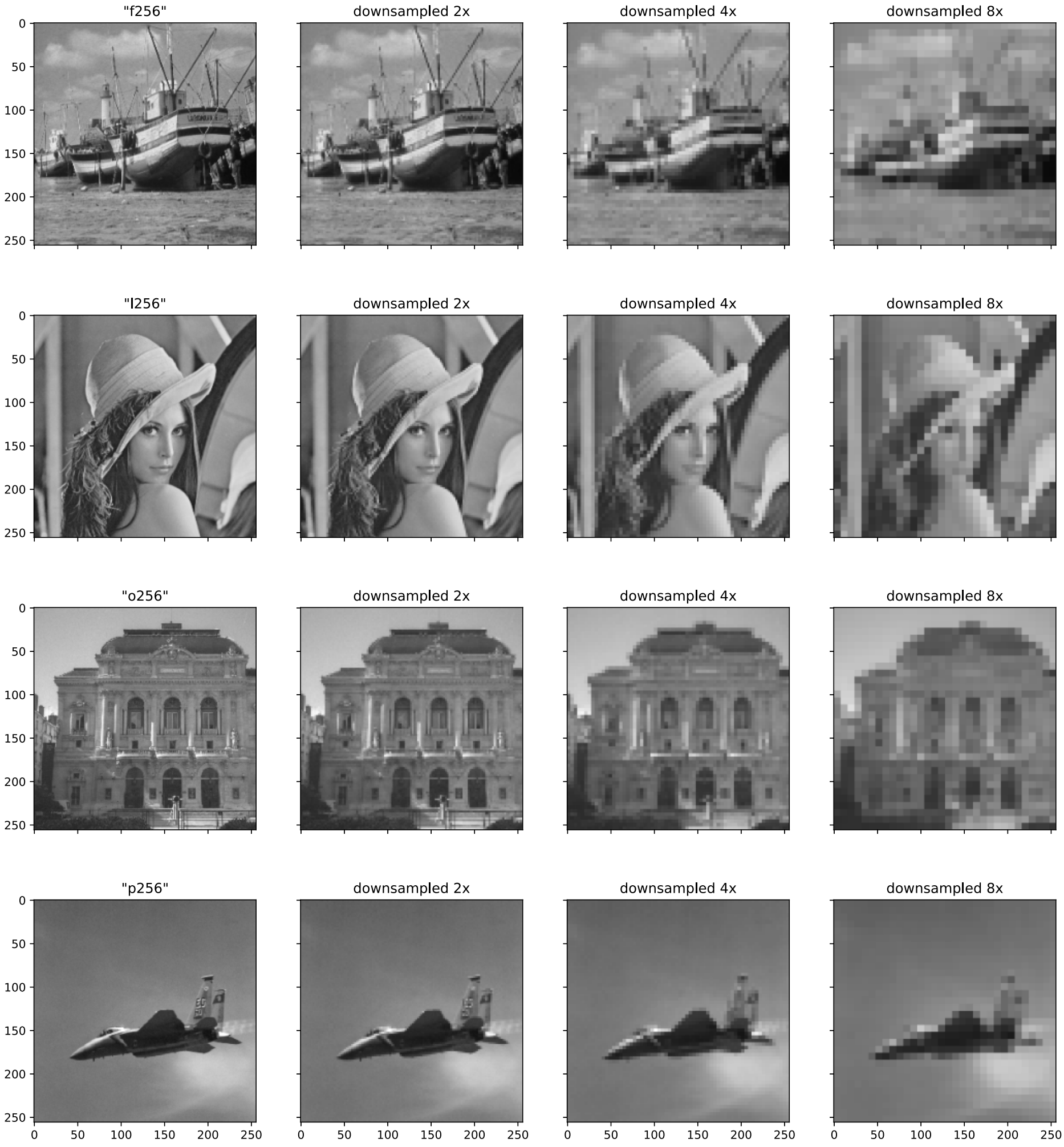
In [9]:
```
plot_downsampled('down_4x', 8)
```



Compare images

In [10]:

```
rows, cols = len(images), 4

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)
fig.suptitle("Comparing", fontsize=18)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    orig = image_dict['orig']
    down_2x = image_dict['down_2x']
    down_4x = image_dict['down_4x']
    down_8x = image_dict['down_8x']

    axs[idx, 0].set_title('"{}"'.format(filename))
    axs[idx, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[idx, 1].set_title('downsampled 2x')
    axs[idx, 1].imshow(down_2x, cmap='gray', vmin=0, vmax=255)

    axs[idx, 2].set_title('downsampled 4x')
    axs[idx, 2].imshow(down_4x, cmap='gray', vmin=0, vmax=255)

    axs[idx, 3].set_title('downsampled 8x')
    axs[idx, 3].imshow(down_8x, cmap='gray', vmin=0, vmax=255)

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Save and display the figure
plt.savefig('downsampled_comp.jpg')
plt.show()
```



Upsampling

Upscale the down sampled images through interpolation and comment on visual quality of the image.

In [11]:

```
def upsample(image, multiplier):
    '''
    Upsamples an image.

    Averages pixel values in blocks of size = multiplier

    Parameters:
        image (array-like): Input image
        multiplier(int): Block size

    Returns:
        int: Upsampled image
    '''

    up_img = np.copy(image)
    height, width = up_img.shape
```

```
    for i in range(height - multiplier)[::multiplier]:
        for j in range(width - multiplier)[::multiplier]:
            step = multiplier

            '''
            Average between current block and next block in all three directions.
            '''
            pix_r = (int(up_img[i][j]) + int(up_img[i][j + multiplier])) // 2
            pix_b = (int(up_img[i][j]) + int(up_img[i + multiplier][j])) // 2
            pix_d = (int(up_img[i][j]) + int(up_img[i + multiplier][j + multiplier])) // 2

            '''
            Assign right half of current block as average value between its left and
            right blocks
            '''
            for r in range(i, i + (multiplier // 2)):
                for c in range(j + (multiplier // 2), j + multiplier):
                    up_img[r][c] = pix_r

            '''
            Assign bottom half of current block as average value between its top and
            bottom blocks
            '''
            for r in range(i + (multiplier // 2), i + multiplier):
                for c in range(j, j + (multiplier // 2)):
                    up_img[r][c] = pix_b

            '''
            Assign corner half of current block as average value between its left and
            right diagonal blocks
            '''
            for r in range(i + (multiplier // 2), i + multiplier):
                for c in range(j + (multiplier // 2), j + multiplier):
                    up_img[r][c] = pix_d

    return up_img
```

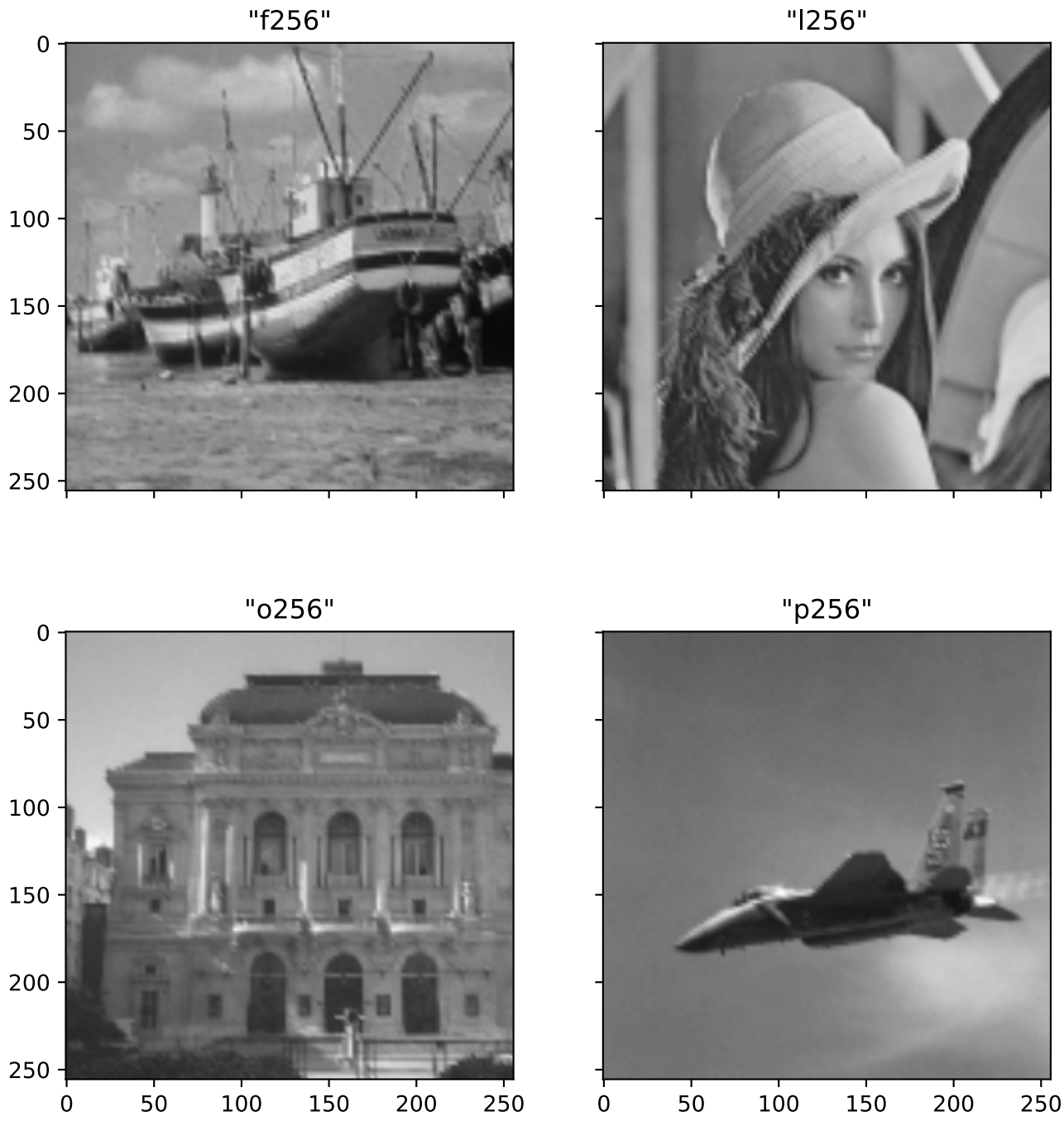
In [12]:

```
def plot_upsampled(key, multiplier):
    '''
    Upsamples all images and plots them.

    Parameters:
        key (str): Key in image dict to access required input images
        multiplier (int): Block size

    Returns:
        None
    '''

    cols = 2
    rows = -(-len(filenames) // cols)

    # Create figure with rows × cols subplots
    fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
    fig.set_size_inches(4 * cols, 4.5 * rows)
    fig.suptitle('Upsampled: {} time ({}x)'.format(
        int(math.log(multiplier, 2)), multiplier),
        fontsize=18
    )

    # Iterate for all images
    for idx, image_dict in enumerate(images):
        filename = image_dict['filename']

        for i in range(int(math.log(multiplier, 2))):
            up_img = upsample(image_dict[key], multiplier)
            # Add image to dictionary
            images[idx]['up_{}x'.format(multiplier)] = up_img

        # Set subplot title
        axs[idx // cols, idx % cols].set_title('"{}"'.format(filename))
        # Add subplot to figure plot buffer
        axs[idx // cols, idx % cols].imshow(up_img, cmap='gray', vmin=0, vmax=255)

        # Save threshold image as .bmp file
        plt.imsave(
            path_out_conv + ext_out[1:] + '/' + filename + '_up_{}x'.format(multiplier) + ext_out,
            up_img,
            cmap='gray',
            vmin=0,
            vmax=255
        )

        # Save pixel values of threshold image as a 2D matrix in a .dat file
        np.savetxt(
            path_out_conv + ext_inp[1:] + '/' + filename + '_up_{}x'.format(multiplier) + ext_inp,
            up_img,
            fmt=' %d',
            newline=' \n'
        )

    # Hide x labels and tick labels for top plots and y ticks for right plots
```

```
    for ax in axs.flat:
        ax.label_outer()

    # Save and display the figure
    plt.savefig('upsampled_{}x.jpg'.format(multiplier))
    plt.show()
```

1 time (2x)

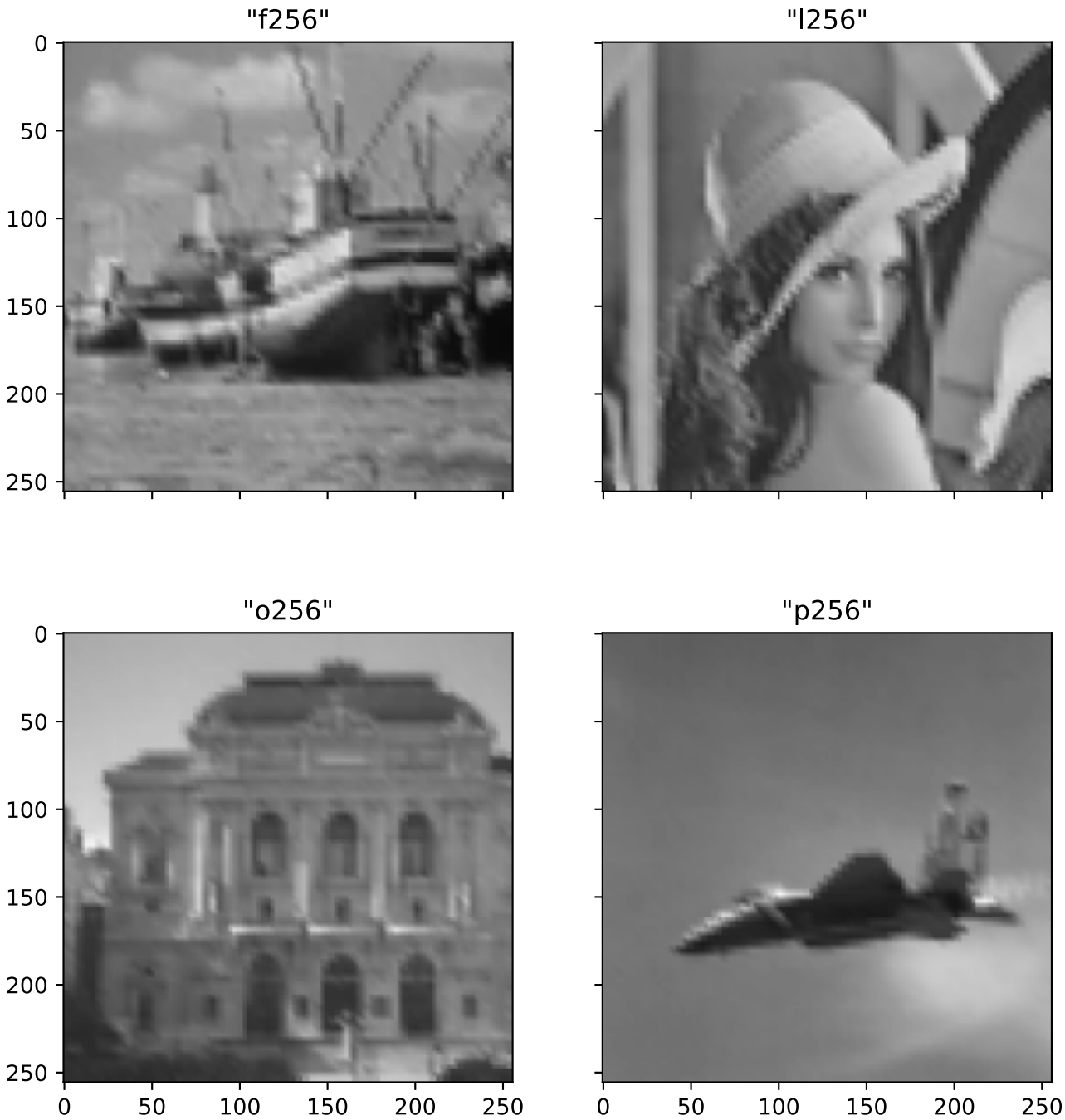
In [13]:
```
plot_upsampled('down_2x', 2)
```



2 times (4x)

In [14]:
```
plot_upsampled('down_4x', 4)
```

Upsampled: 2 time (4x)



3 times (8x)

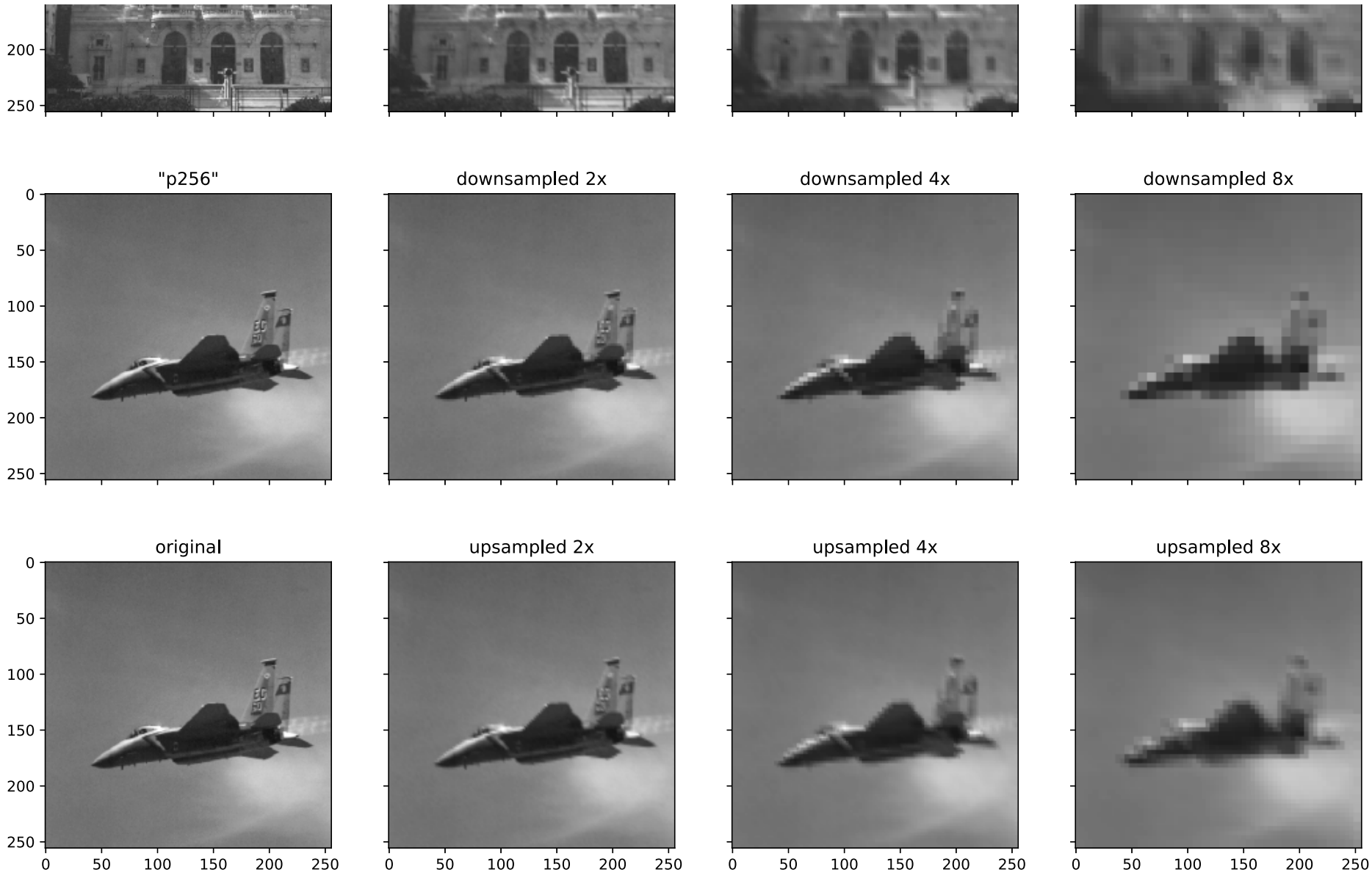
In [15]:
```
plot_upsampled('down_8x', 8)
```

Compare images

In [16]:

```
rows, cols = len(images) * 2, 4

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    orig = image_dict['orig']
    down_2x = image_dict['down_2x']
    down_4x = image_dict['down_4x']
    down_8x = image_dict['down_8x']
    up_2x = image_dict['up_2x']
    up_4x = image_dict['up_4x']
    up_8x = image_dict['up_8x']

    axs[2 * idx, 0].set_title('"{}"'.format(filename))
    axs[2 * idx, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[2 * idx, 1].set_title('downsampled 2x')
    axs[2 * idx, 1].imshow(down_2x, cmap='gray', vmin=0, vmax=255)

    axs[2 * idx, 2].set_title('downsampled 4x')
    axs[2 * idx, 2].imshow(down_4x, cmap='gray', vmin=0, vmax=255)

    axs[2 * idx, 3].set_title('downsampled 8x')
    axs[2 * idx, 3].imshow(down_8x, cmap='gray', vmin=0, vmax=255)

    axs[2 * idx + 1, 0].set_title('original')
    axs[2 * idx + 1, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[2 * idx + 1, 1].set_title('upsampled 2x')
    axs[2 * idx + 1, 1].imshow(up_2x, cmap='gray', vmin=0, vmax=255)

    axs[2 * idx + 1, 2].set_title('upsampled 4x')
    axs[2 * idx + 1, 2].imshow(up_4x, cmap='gray', vmin=0, vmax=255)

    axs[2 * idx + 1, 3].set_title('upsampled 8x')
    axs[2 * idx + 1, 3].imshow(up_8x, cmap='gray', vmin=0, vmax=255)

# Hide x labels and tick labels for top plots and y ticks for right plots
```

```
for ax in axs.flat:
    ax.label_outer()

# Save and display the figure
plt.savefig('upsampled_comp.jpg')
plt.show()
```



Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 3

Write a program to implement image negation operation.

$S = L - 1 - R$,

where R : pixel value of input image; S : pixel value of output image; L : maximum gray value

See the effect of the image negation operation for enhancing white or gray detail embedded in dark regions of an image dominant in size.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'a256',
    'ba256',
    'n256',
    'o256',
    'p256',
    'z256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

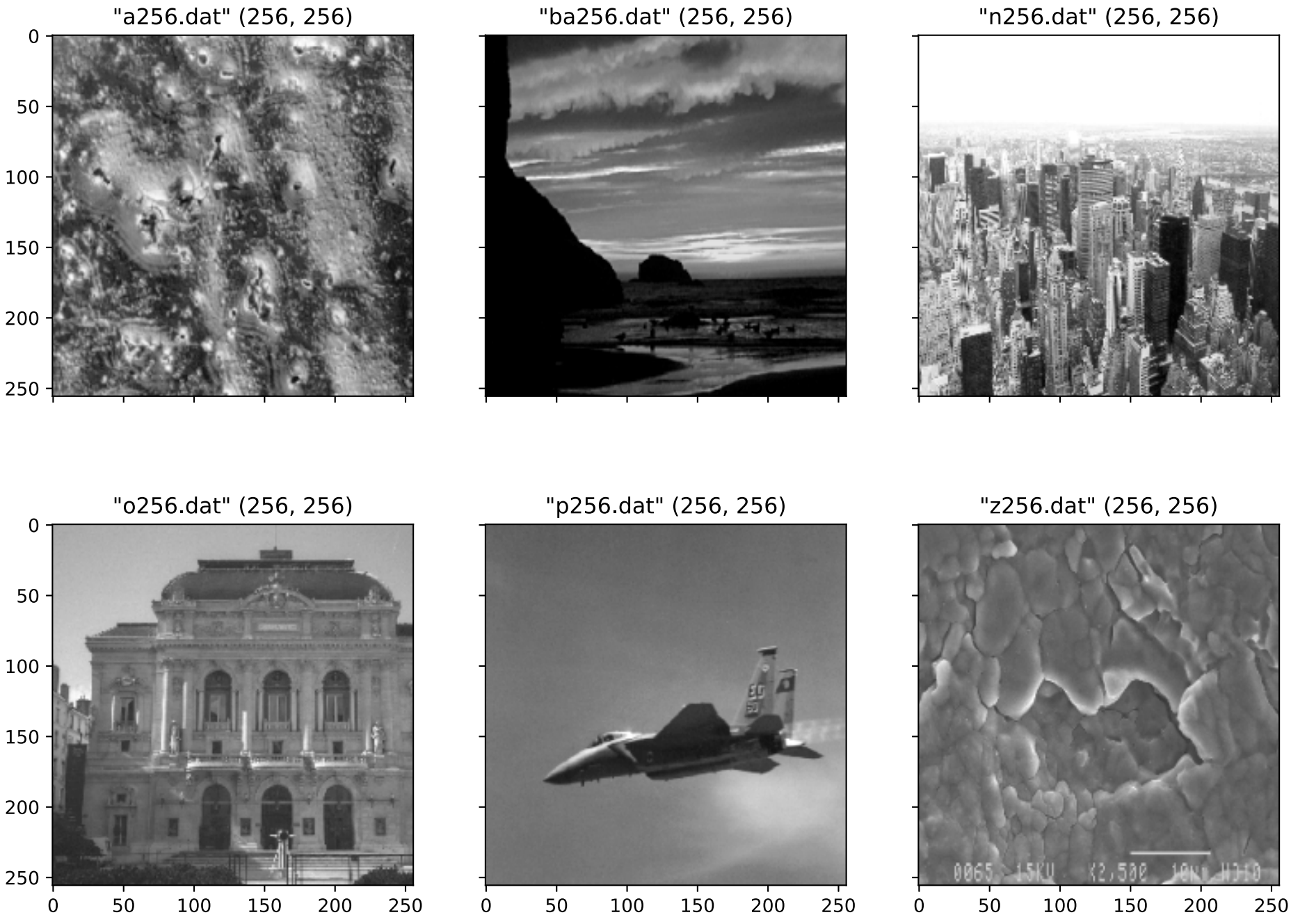
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [5]:
```
# Matrix dimensions
cols = 3
rows = -(-len(filenames) // cols)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[int(idx // cols), idx % cols].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[int(idx // cols), idx % cols].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
```

```
    )

    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



Negation

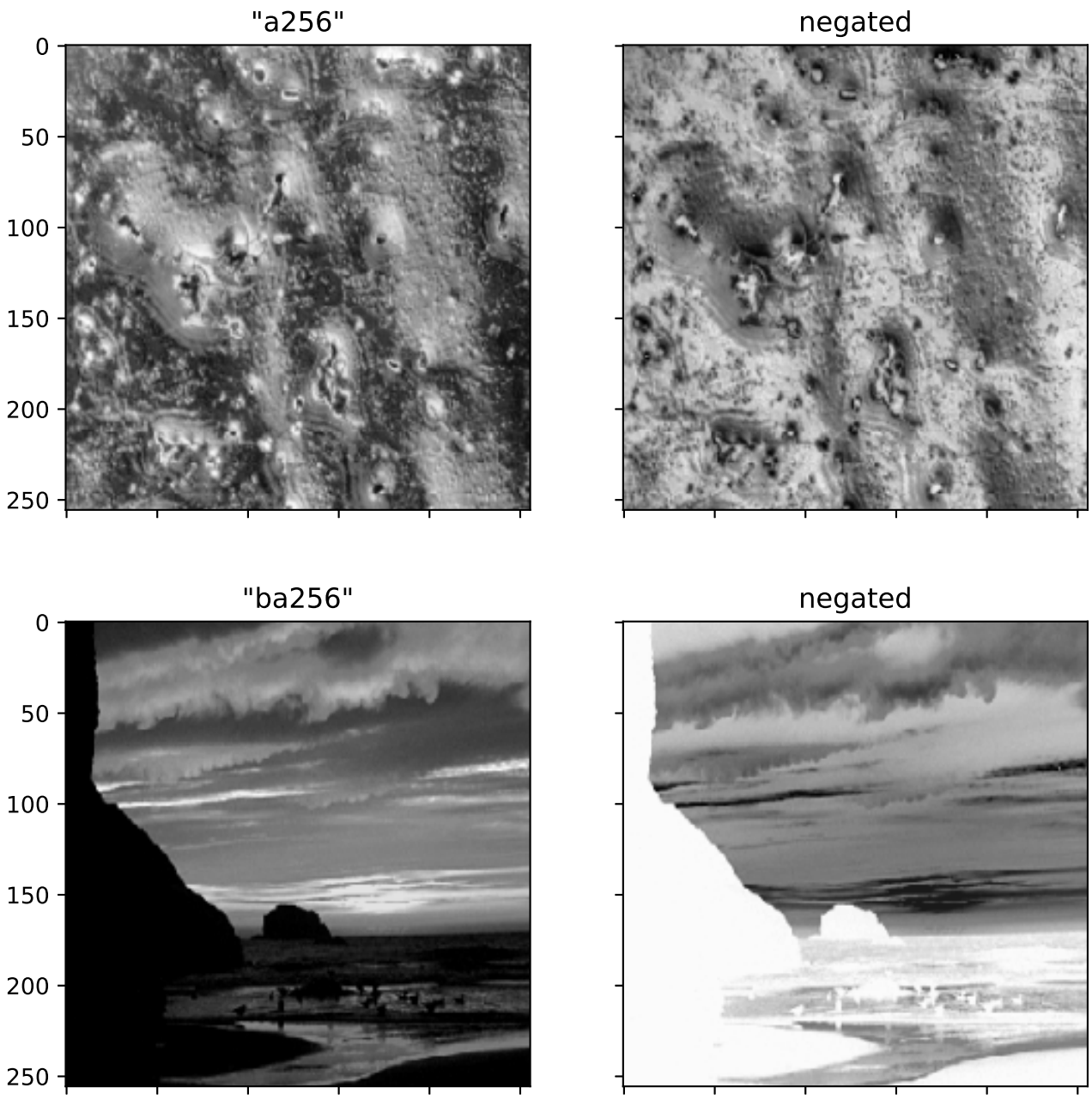
In [6]:

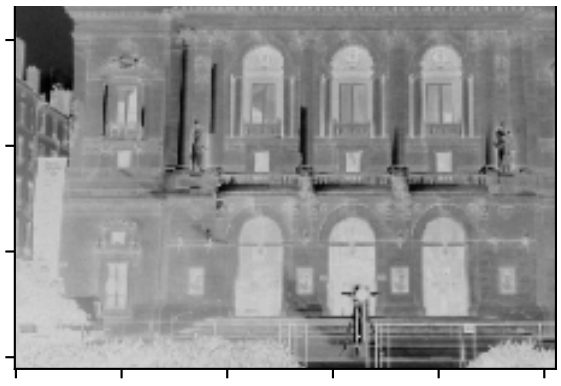
```
def negate(image):
    min_pixel = min([min(i) for i in image])
    max_pixel = max([max(i) for i in image])

    image = max_pixel - 1 - image

    return image
```

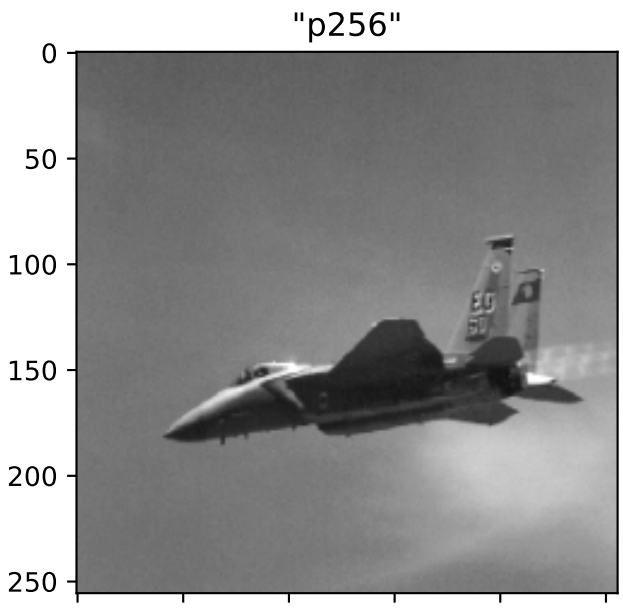
In [8]:

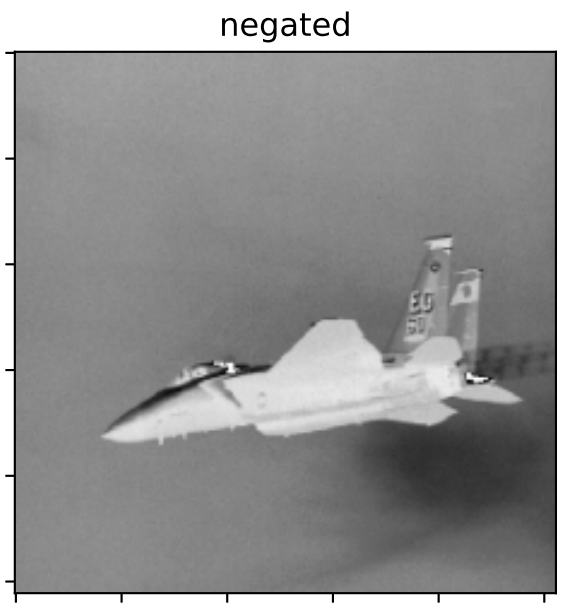
```
rows, cols = len(images), 2

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    orig = image_dict['orig']

    neg_img = negate(orig)

    axs[idx, 0].set_title('"{}"'.format(filename))
    axs[idx, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[idx, 1].set_title('negated')
    axs[idx, 1].imshow(neg_img, cmap='gray', vmin=0, vmax=255)

    # Save threshold image as .bmp file
```

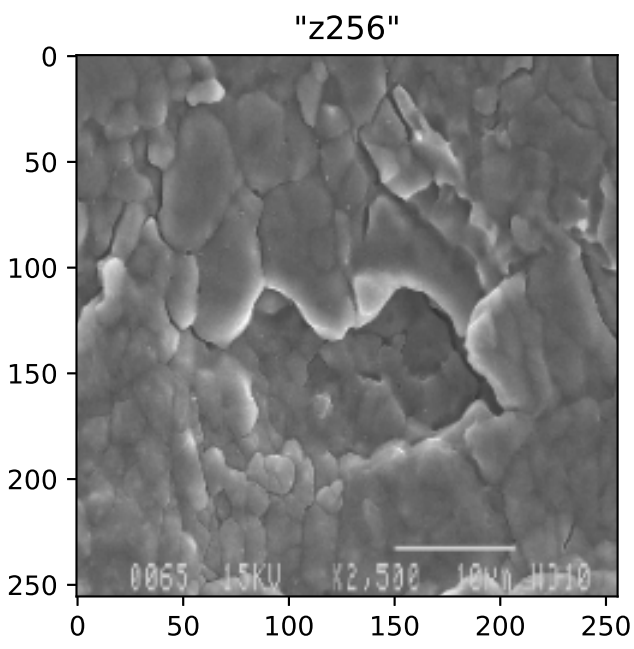
```
    plt.imsave(
        path_out_conv + ext_out[1:] + '/' + filename + '_neg' + ext_out,
        neg_img,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save pixel values of threshold image as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_neg' + ext_inp,
        neg_img,
        fmt=' %d',
        newline=' \n'
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Save and display the figure
plt.savefig('negated_comp.jpg')
plt.show()
```

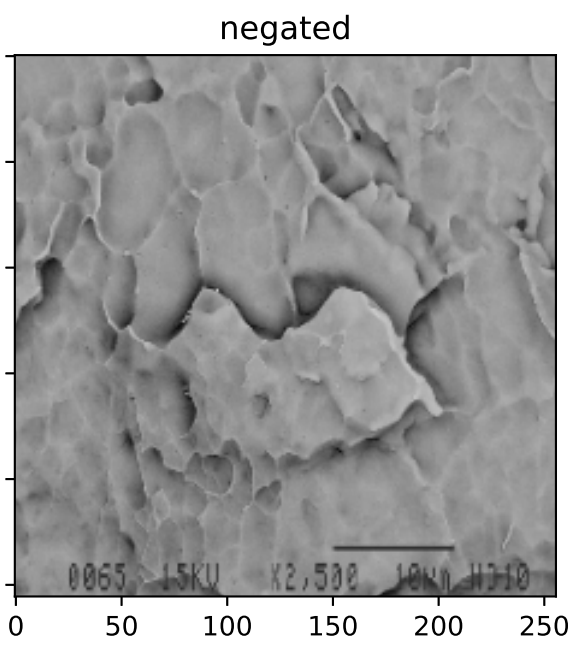












Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 4

Write a program to implement change in dynamic range of an image from `[a, b]` to `[c, d]`. `a` and `c` are the minimum pixel values of input and output images respectively, while `b` and `d` are the maximum for the two. Comment on visual quality of the image after the operation.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'l256',
    'o256',
    'p256',
    'z256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

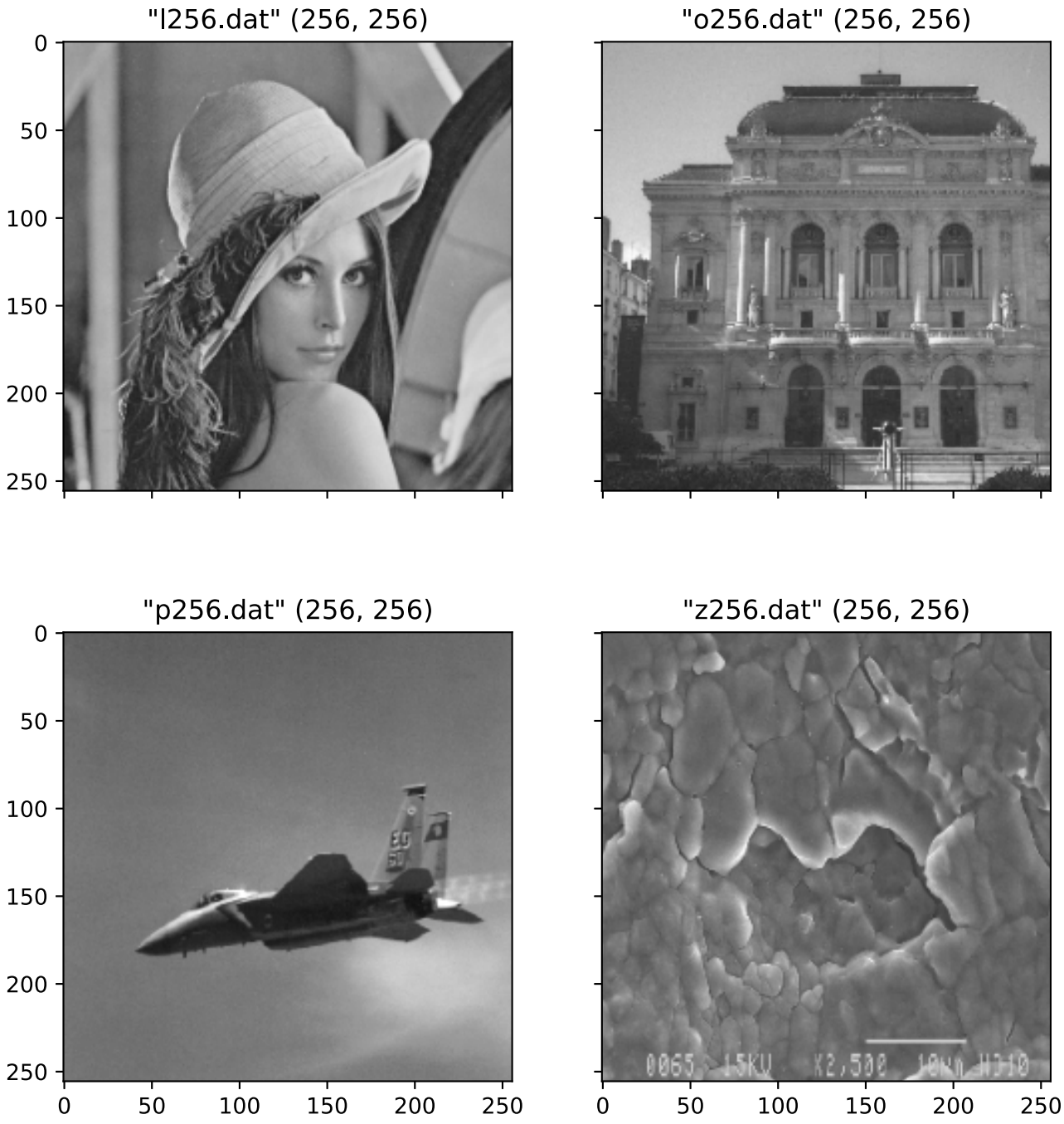
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 2
rows = -(-len(filenames) // cols)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[int(idx // cols), idx % cols].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[int(idx // cols), idx % cols].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
```

```
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



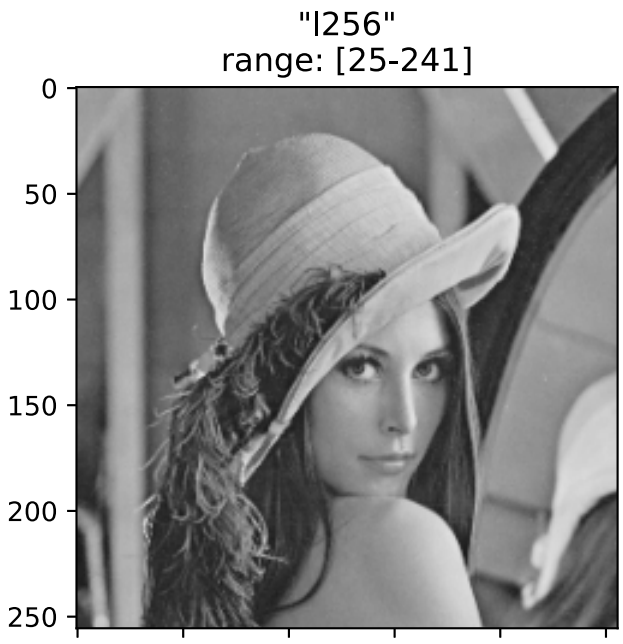
Dynamic Range Shift

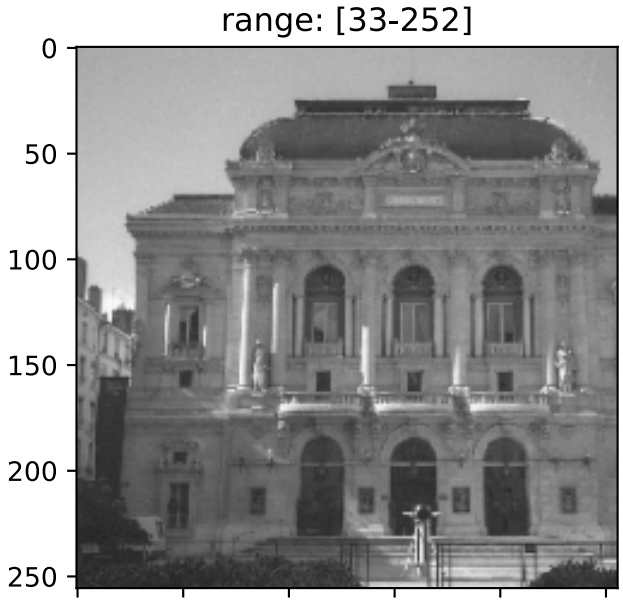
In [5]:
```
def shift_dyn_range(image, range):
    min_pixel = min([min(i) for i in image])
    max_pixel = max([max(i) for i in image])

    min_range, max_range = range

    image = \
        min_range + \
        ((image - min_pixel) / (max_pixel - min_pixel)) * (max_range - min_range)

    return np.array(image, dtype='uint8')
```

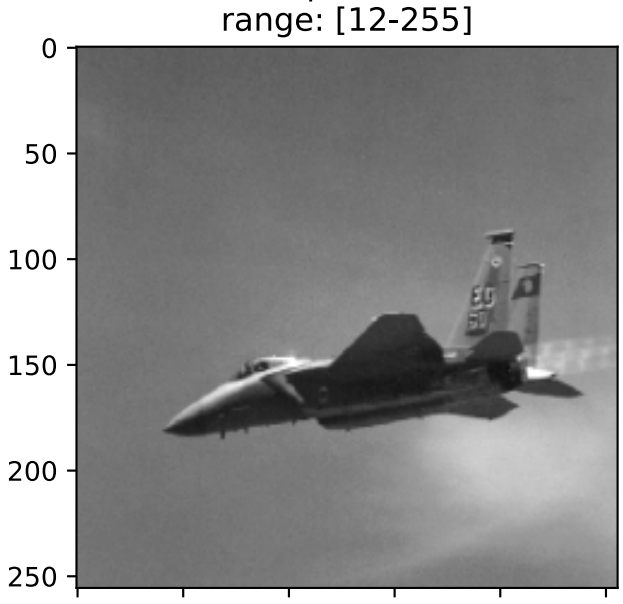
In [6]:
```
rows, cols = len(images), 2

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    orig = image_dict['orig']
    shift_img = shift_dyn_range(orig, [0, 255])

    min_orig = min([min(i) for i in orig])
    max_orig = max([max(i) for i in orig])

    axs[idx, 0].set_title('"{}"\nrange: [{}-{}]'.format(
        filename,
        min_orig,
        max_orig
    ))
    axs[idx, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)
```

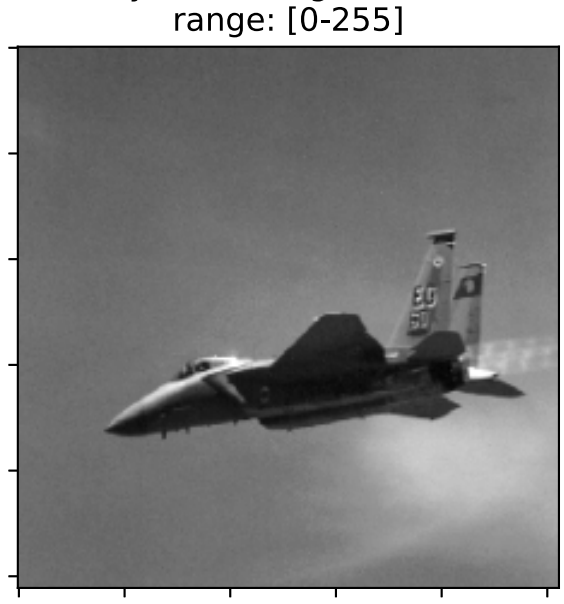
```
    min_shift = min([min(i) for i in shift_img])
    max_shift = max([max(i) for i in shift_img])

    axs[idx, 1].set_title('dynamic range shifted\nrange: [{}-{}]'.format(
        min_shift,
        max_shift
    ))
    axs[idx, 1].imshow(shift_img, cmap='gray', vmin=0, vmax=255)

    # Save threshold image as .bmp file
    plt.imsave(
        path_out_conv + ext_out[1:] + '/' + filename + '_dyn_shift' + ext_out,
        shift_img,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save pixel values of threshold image as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_dyn_shift' + ext_inp,
        shift_img,
        fmt=' %d',
        newline=' \n'
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()
```

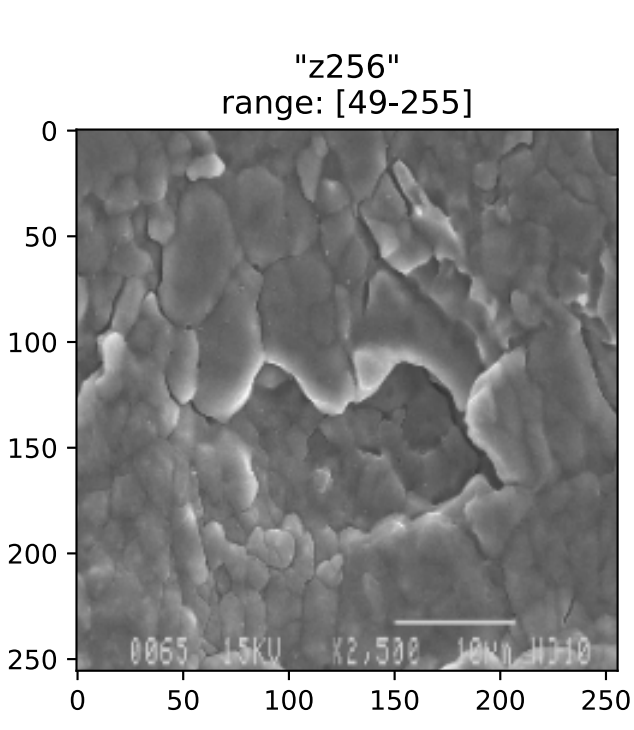
```
# Save and display the figure
plt.savefig('dynamic_range_shift_comp.jpg')
plt.show()
```

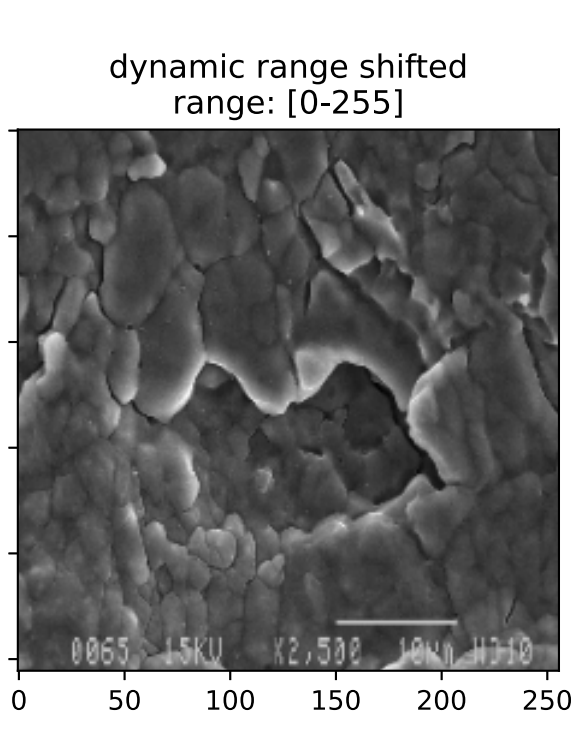












Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 5

Implement image negation operation using logical NOT operation and verify results with using linear transformation function.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import math
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'a256',
    'ba256',
    'n256',
    'o256',
    'p256',
    'z256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

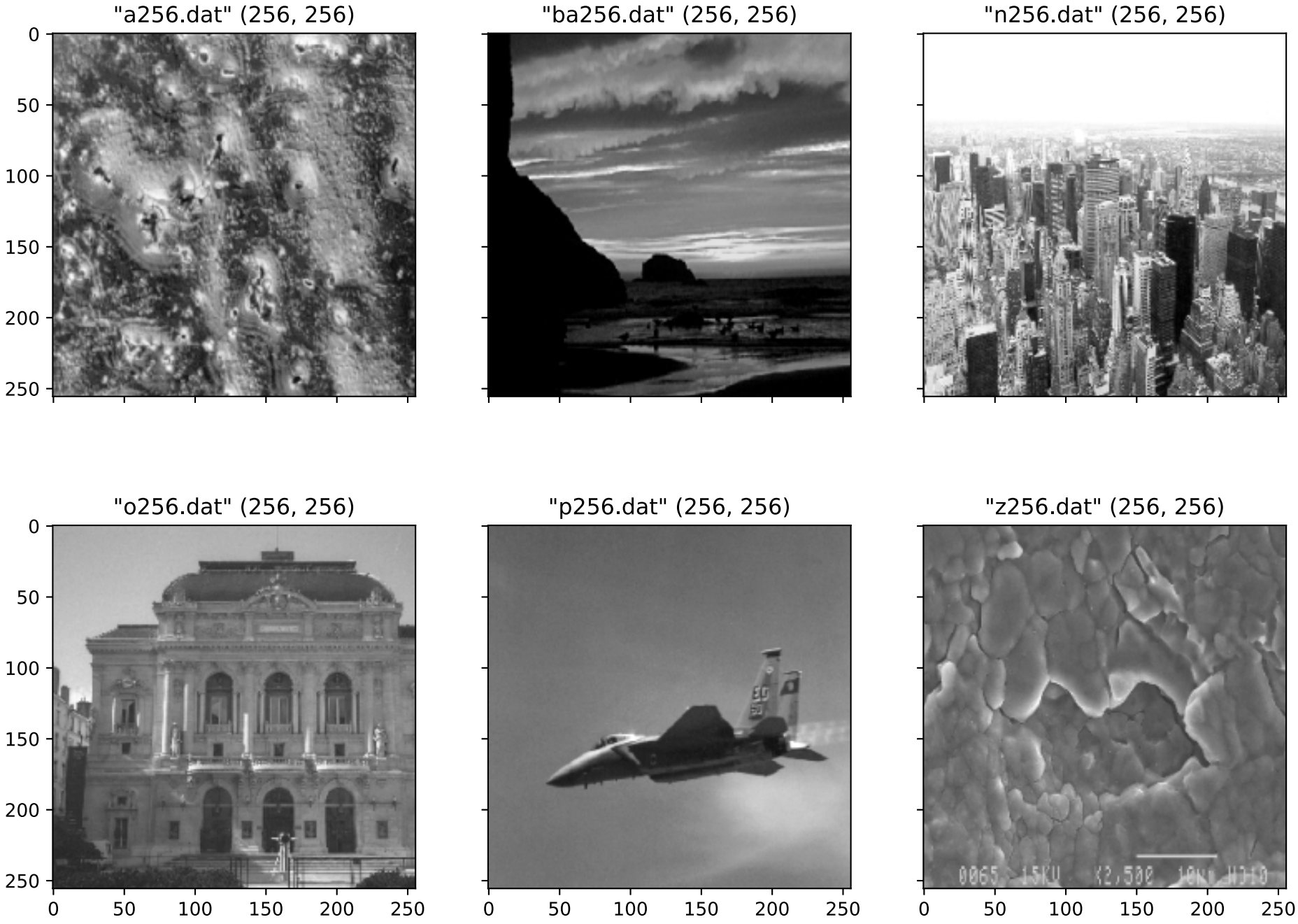
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = -(-len(filenames) // cols)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[int(idx // cols), idx % cols].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[int(idx // cols), idx % cols].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
```

```
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



Negation

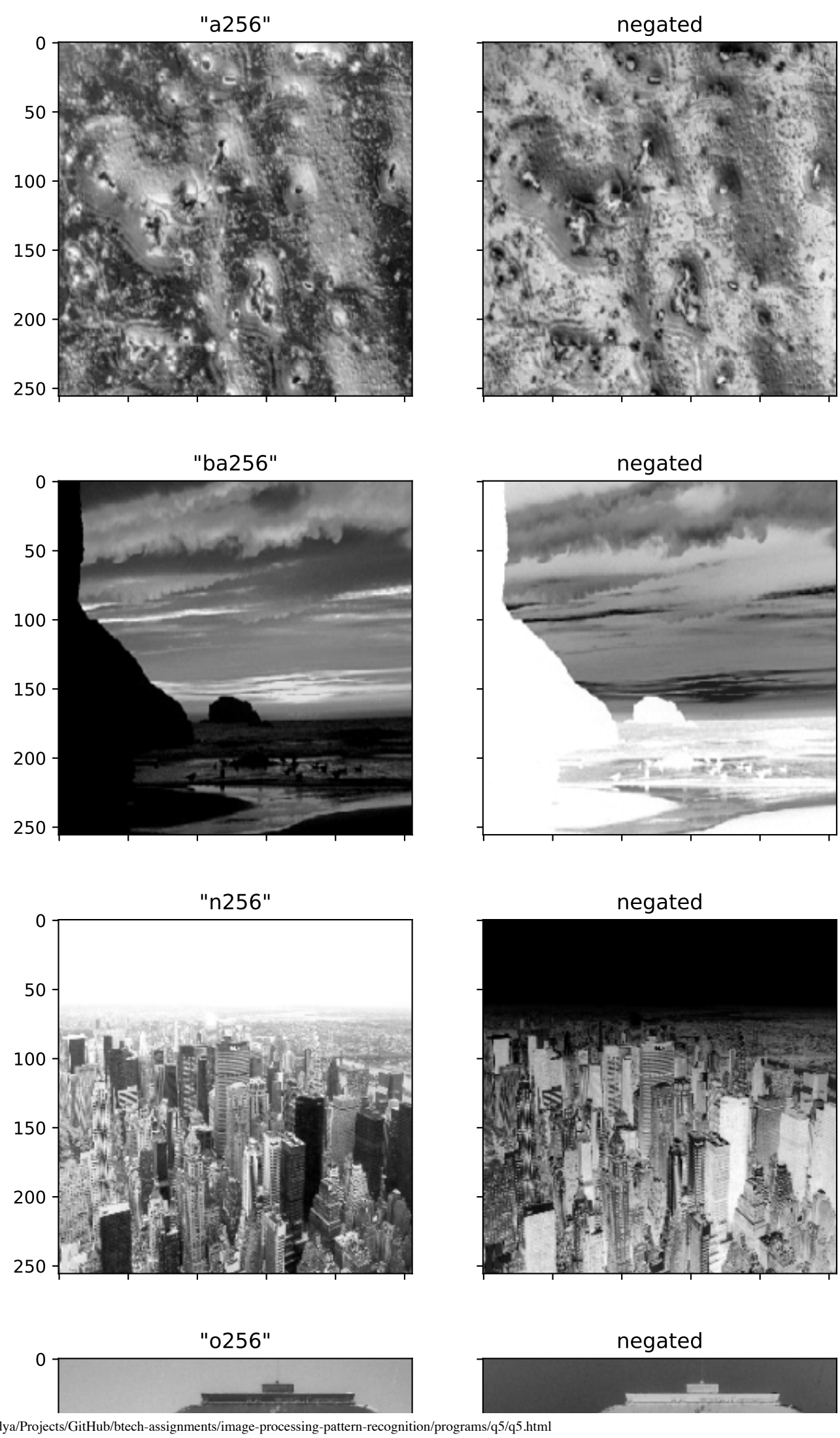
In [5]:

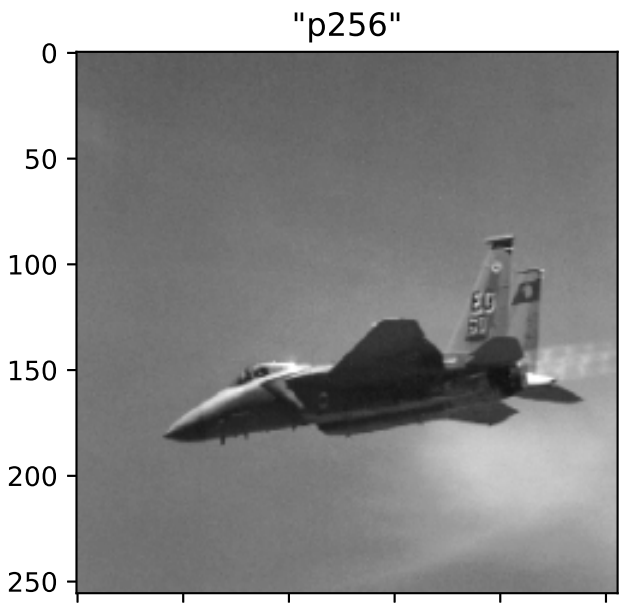
```
def negate(image):
    height, width = image.shape

    neg_img = np.array(np.zeros(image.shape), dtype='uint8')
    for i in range(height):
        for j in range(width):
            pix = image[i][j]
            mask = 1
            neg = 0
            for b in range(8):
                if not pix & mask:
                    neg |= 1 << b
                mask = mask << 1
            neg_img[i][j] = neg

    return neg_img
```

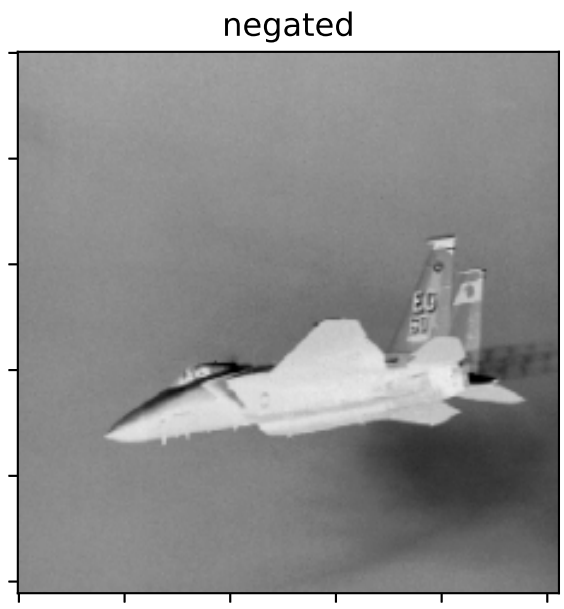
In [6]:

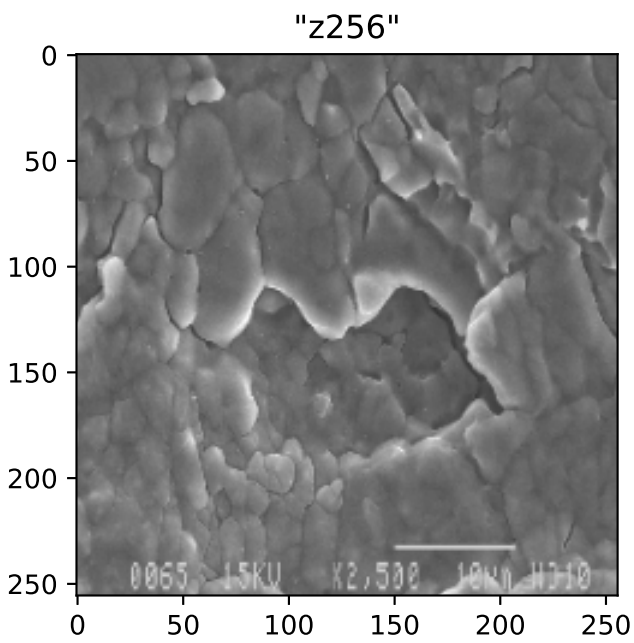
```
rows, cols = len(images), 2

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    orig = image_dict['orig']

    neg_img = negate(orig)

    axs[idx, 0].set_title('"{}"'.format(filename))
    axs[idx, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[idx, 1].set_title('negated')
```

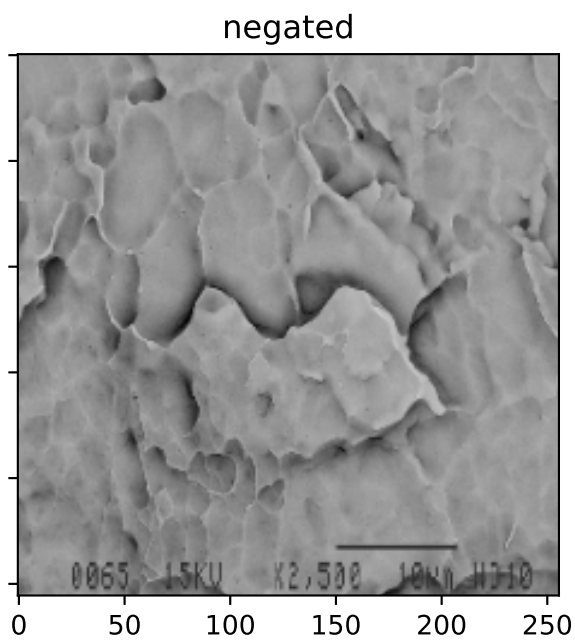
```
    axs[idx, 1].imshow(neg_img, cmap='gray', vmin=0, vmax=255)

    # Save threshold image as .bmp file
    plt.imsave(
        path_out_conv + ext_out[1:] + '/' + filename + '_neg' + ext_out,
        neg_img,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save pixel values of threshold image as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_neg' + ext_inp,
        neg_img,
        fmt=' %d',
        newline=' \n'
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Save and display the figure
plt.savefig('negated_comp.jpg')
plt.show()
```











Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 6

Write a program to develop histogram of an image and show it through display. The size of the image and pixel values is made flexible. Show and comment about the effect on the histogram of the image if

1. lower-order bit planes, and
2. higher order bit plane are set to zero.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'a256',
    'ba256',
    'n256',
    'o256',
    'p256',
    'z256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

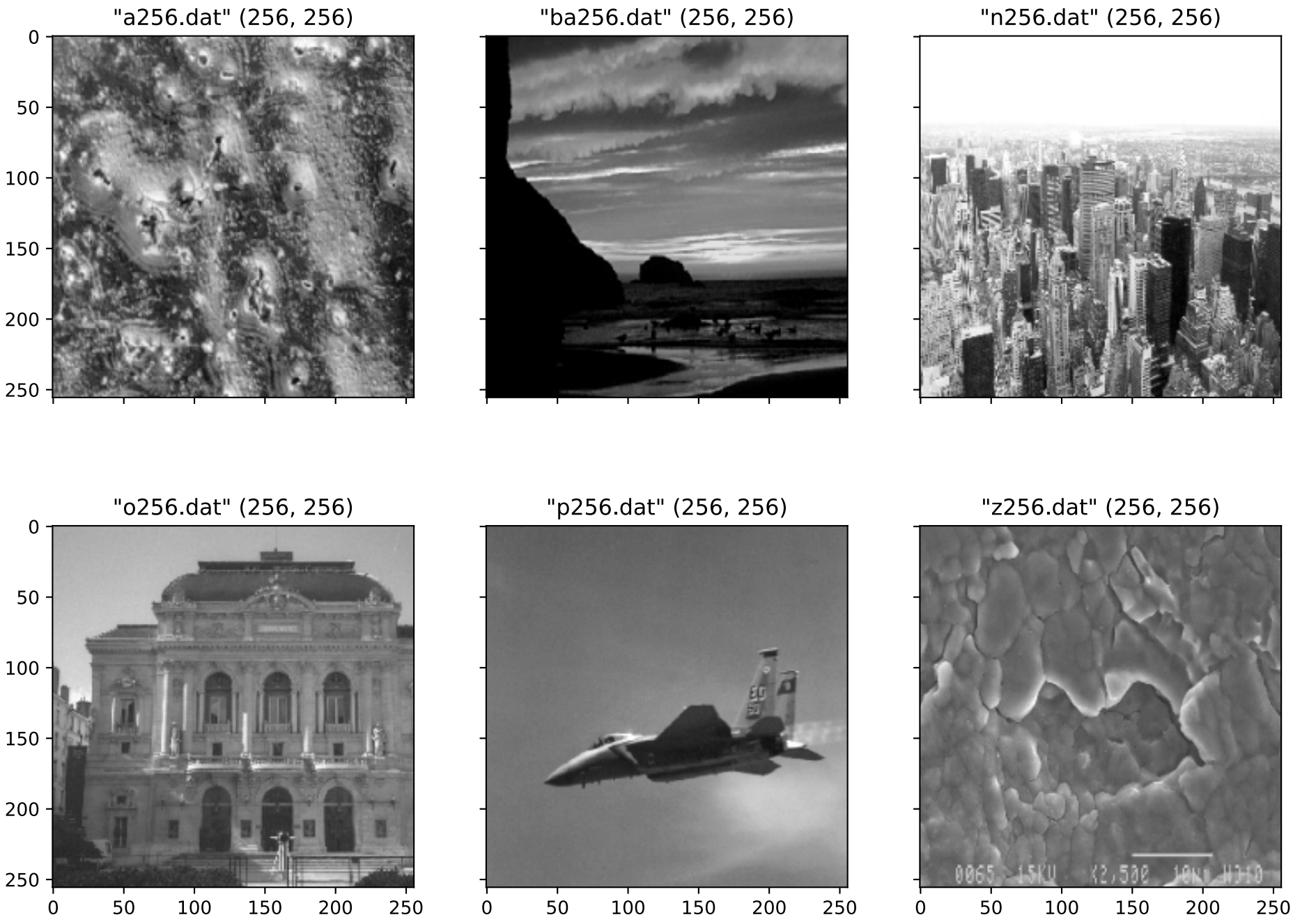
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = -(-len(filenames) // cols)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[int(idx // cols), idx % cols].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[int(idx // cols), idx % cols].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )
```

```
    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



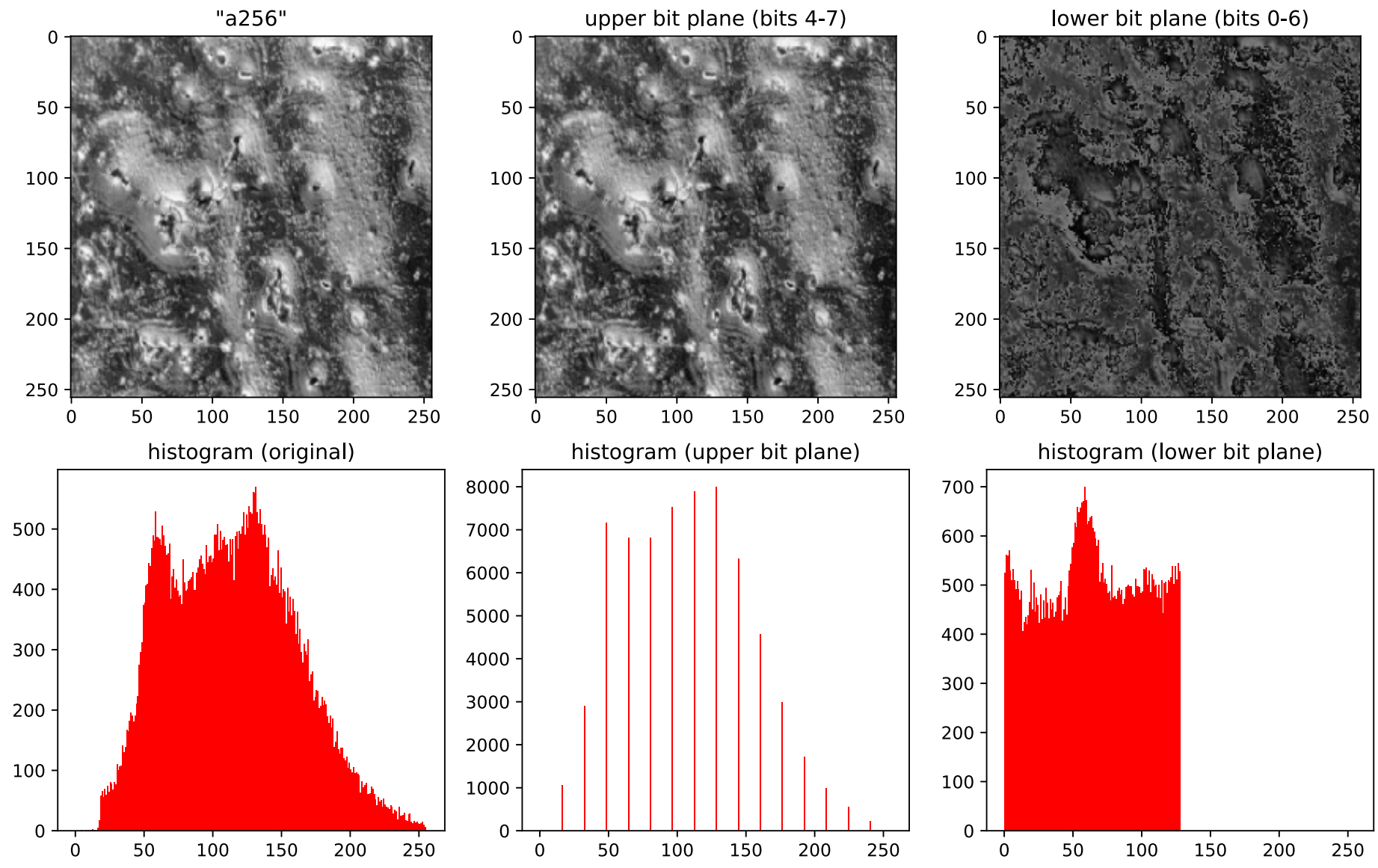
Histogram

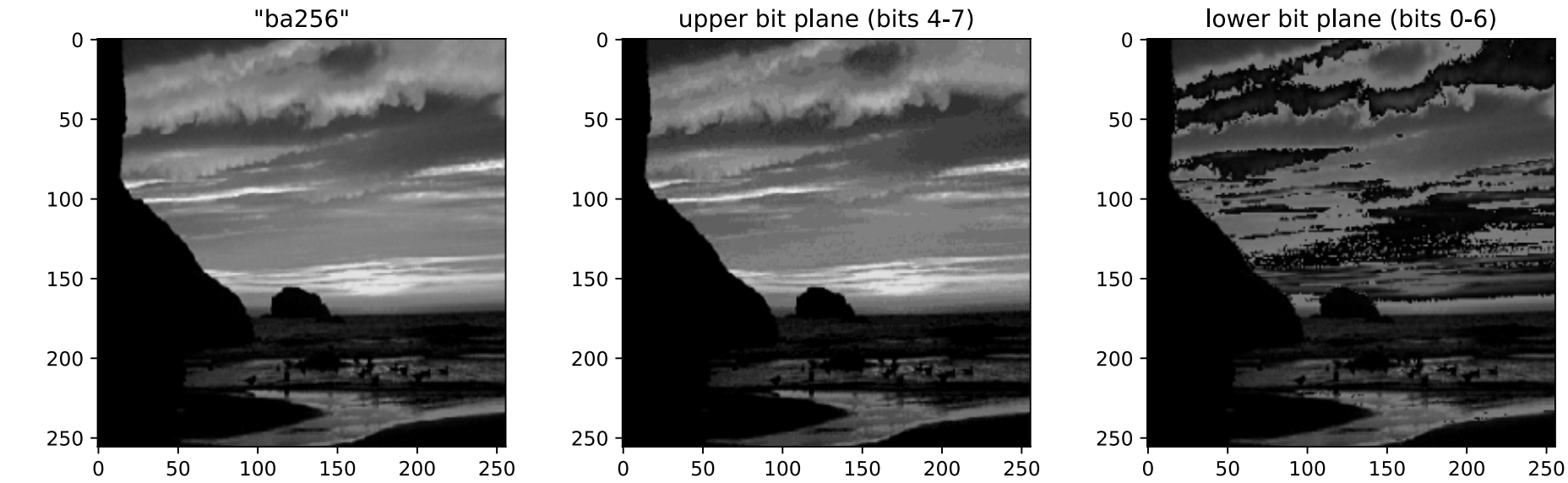
In [5]:
```
def gen_histogram(image):
    histogram = np.zeros(256)

    height, width = image.shape
    for i in range(height):
        for j in range(width):
            histogram[image[i][j]] += 1

    return histogram
```

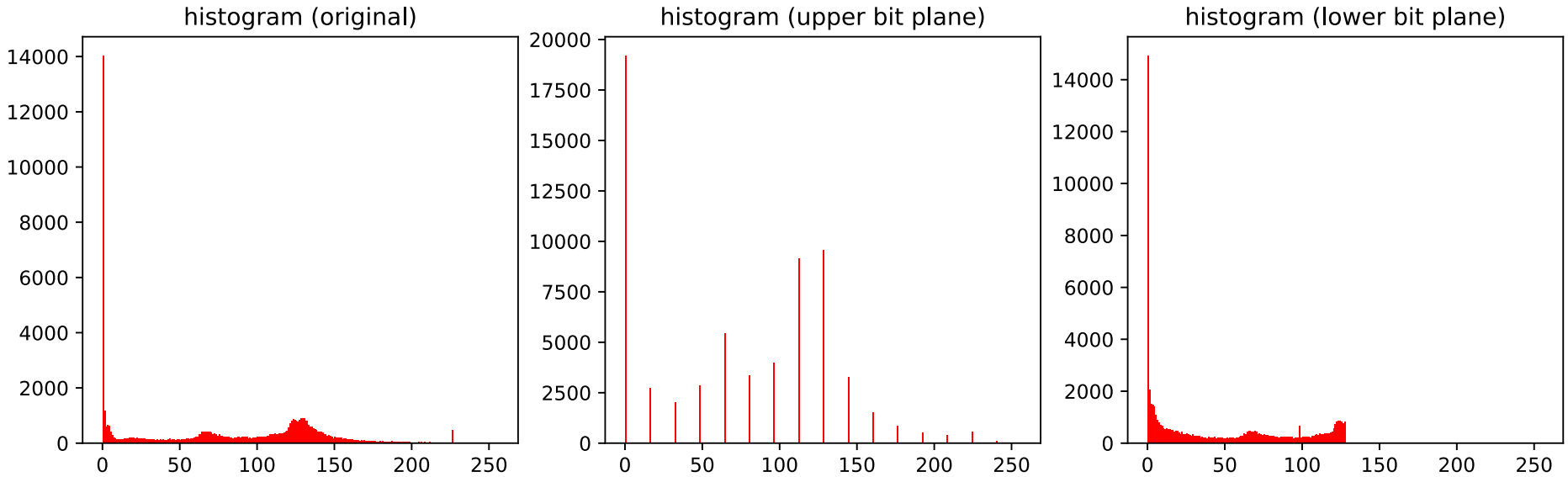
In [6]:
```
def unset_bit_planes(image, bits):
    new_img = np.copy(image)
    mask = 255
    for i in bits:
        mask ^= 1 << i

    height, width = image.shape
    for i in range(height):
        for j in range(width):
            new_img[i][j] &= mask

    return new_img
```

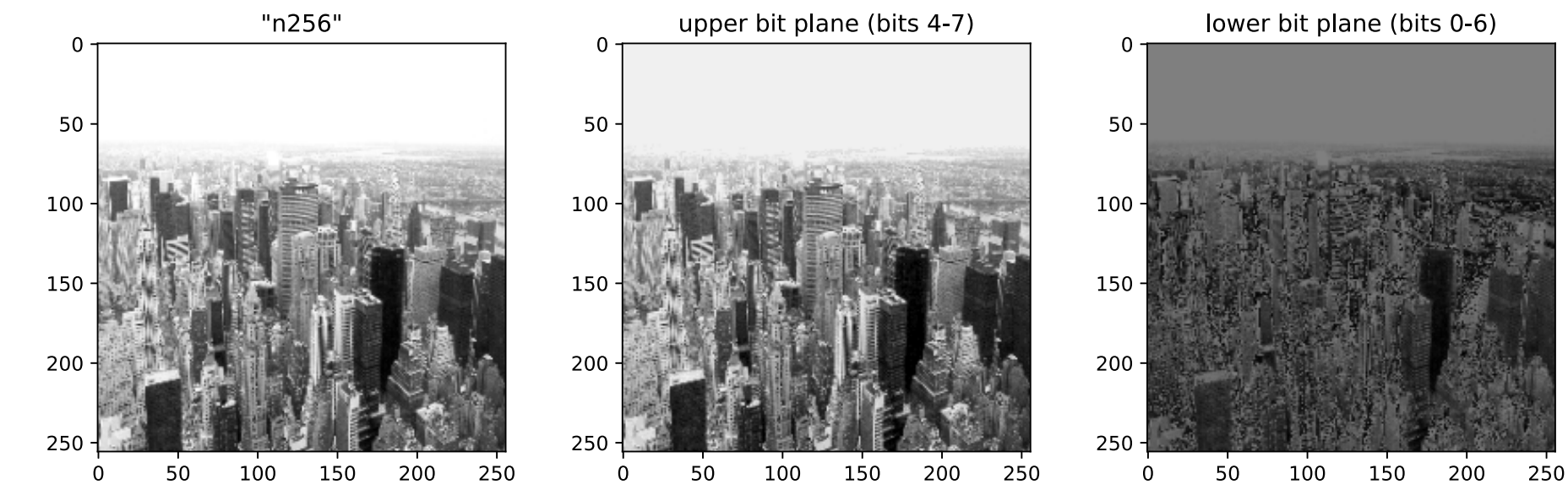
In [7]:
```
rows, cols = len(images) * 2, 3

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)
```

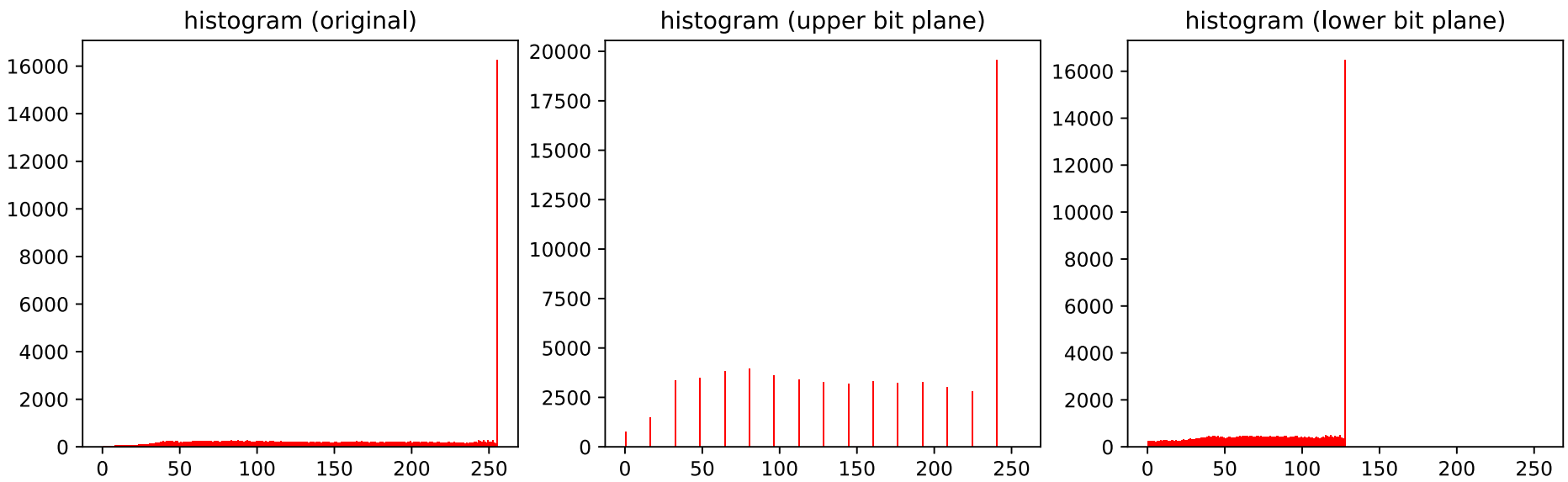
```
# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    orig = image_dict['orig']
    upper = unset_bit_planes(orig, [0, 1, 2, 3])
    lower = unset_bit_planes(orig, [7])

    hist_orig = gen_histogram(orig)
    hist_upper = gen_histogram(upper)
    hist_lower = gen_histogram(lower)

    axs[2 * idx, 0].set_title('"{}"'.format(filename))
    axs[2 * idx, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[2 * idx, 1].set_title('upper bit plane (bits 4-7)'.format(filename))
    axs[2 * idx, 1].imshow(upper, cmap='gray', vmin=0, vmax=255)

    axs[2 * idx, 2].set_title('lower bit plane (bits 0-6)'.format(filename))
    axs[2 * idx, 2].imshow(lower, cmap='gray', vmin=0, vmax=255)

    axs[2 * idx + 1, 0].set_title('histogram (original)')
    axs[2 * idx + 1, 0].hist(orig.flatten(), 256, [0, 256], color = 'r')

    axs[2 * idx + 1, 1].set_title('histogram (upper bit plane)')
    axs[2 * idx + 1, 1].hist(upper.flatten(), 256, [0, 256], color = 'r')

    axs[2 * idx + 1, 2].set_title('histogram (lower bit plane)')
    axs[2 * idx + 1, 2].hist(lower.flatten(), 256, [0, 256], color = 'r')

    # Save pixel values of threshold image as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_hist' + ext_inp,
        hist_orig,
        fmt=' %d',
        newline=' \n'
    )

    # Save pixel values of threshold image as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_hist_upper' + ext_inp,
        hist_upper,
        fmt=' %d',
        newline=' \n'
    )

    # Save pixel values of threshold image as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_hist_lower' + ext_inp,
        hist_lower,
        fmt=' %d',
        newline=' \n'
    )

# Save and display the figure
plt.savefig('histogram_comp.jpg')
plt.show()
```

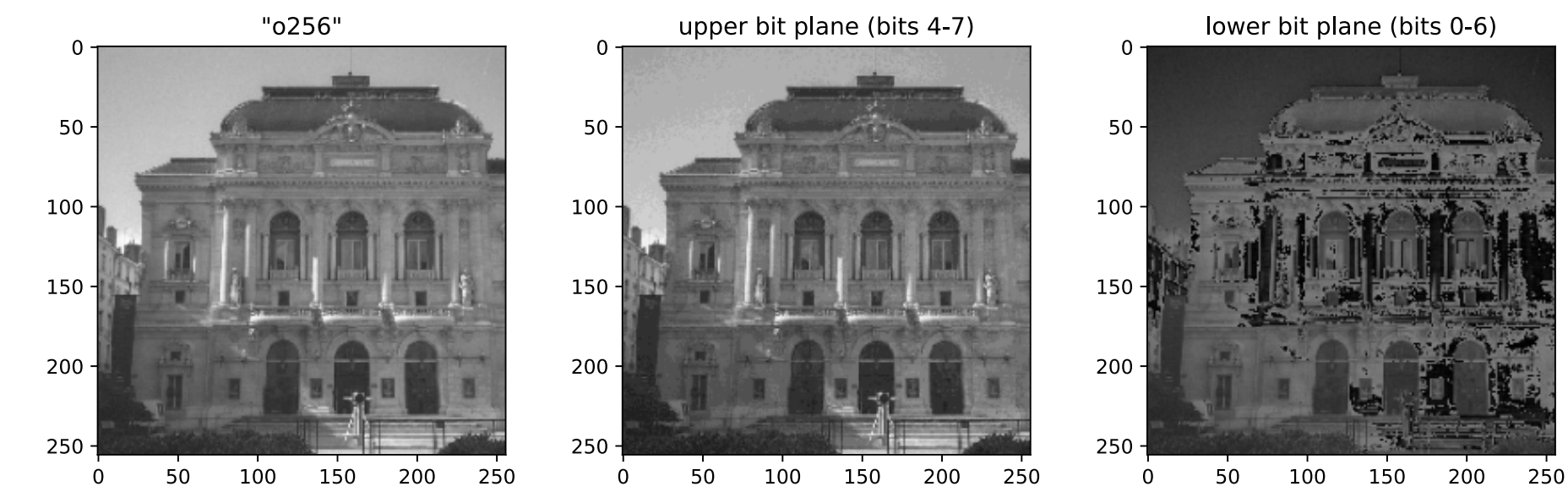



"ba256"
upper bit plane (bits 4-7)
lower bit plane (bits 0-6)

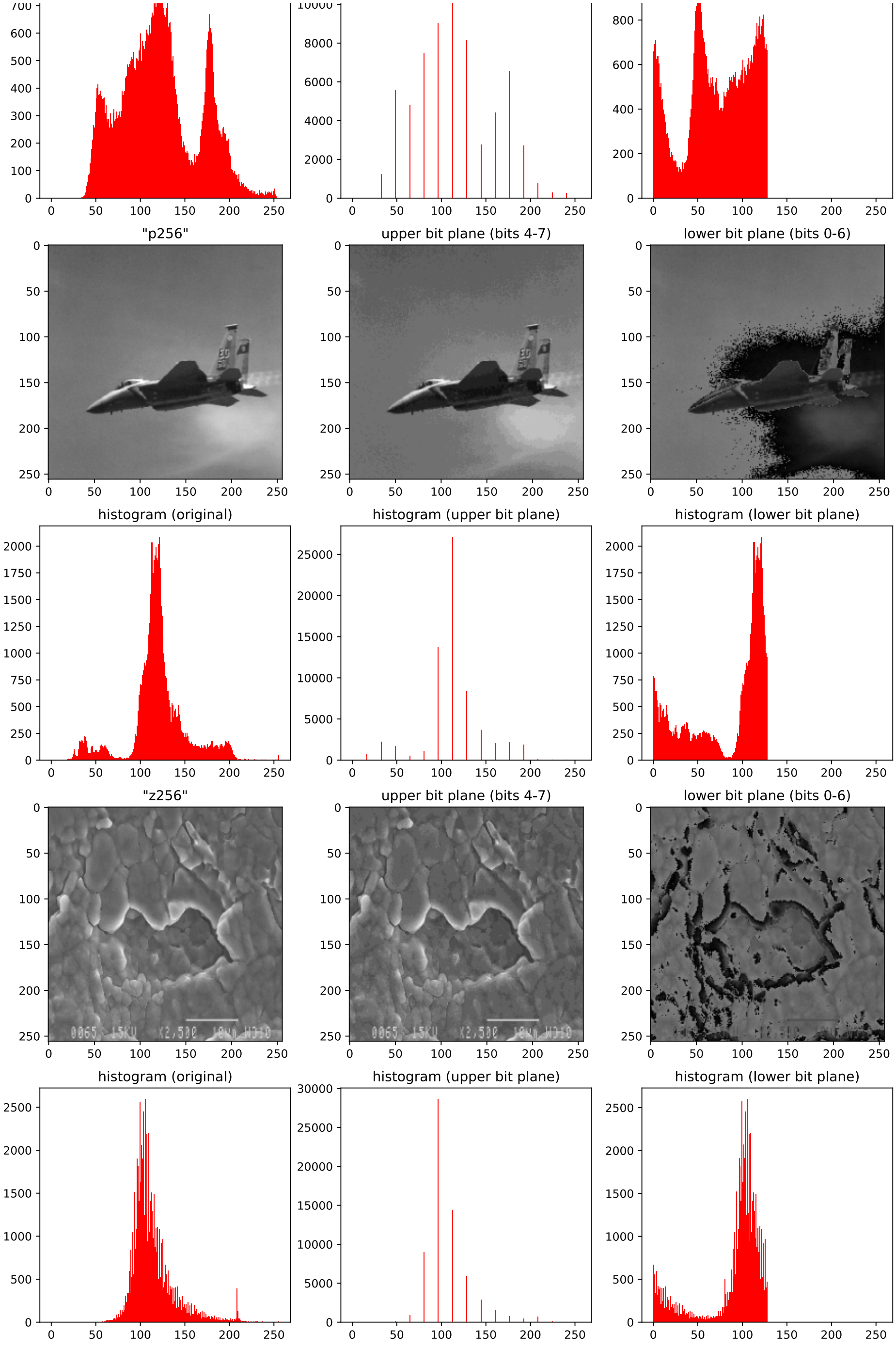

histogram (original)
histogram (upper bit plane)
histogram (lower bit plane)


"n256"
upper bit plane (bits 4-7)
lower bit plane (bits 0-6)


histogram (original)
histogram (upper bit plane)
histogram (lower bit plane)


"o256"
upper bit plane (bits 4-7)
lower bit plane (bits 0-6)

histogram (original)
histogram (upper bit plane)
histogram (lower bit plane)



Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 7

Write a program to implement histogram equalization of an 8-bit/pixel gray scale image. Show that a second pass of histogram equalization will produce exactly the same result as the first pass.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'ba256',
    'f256',
    'l256',
    'o256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image,
        'equalized': None
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 2
rows = -(-len(filenames) // cols)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[int(idx // cols), idx % cols].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[int(idx // cols), idx % cols].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
```

```
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```

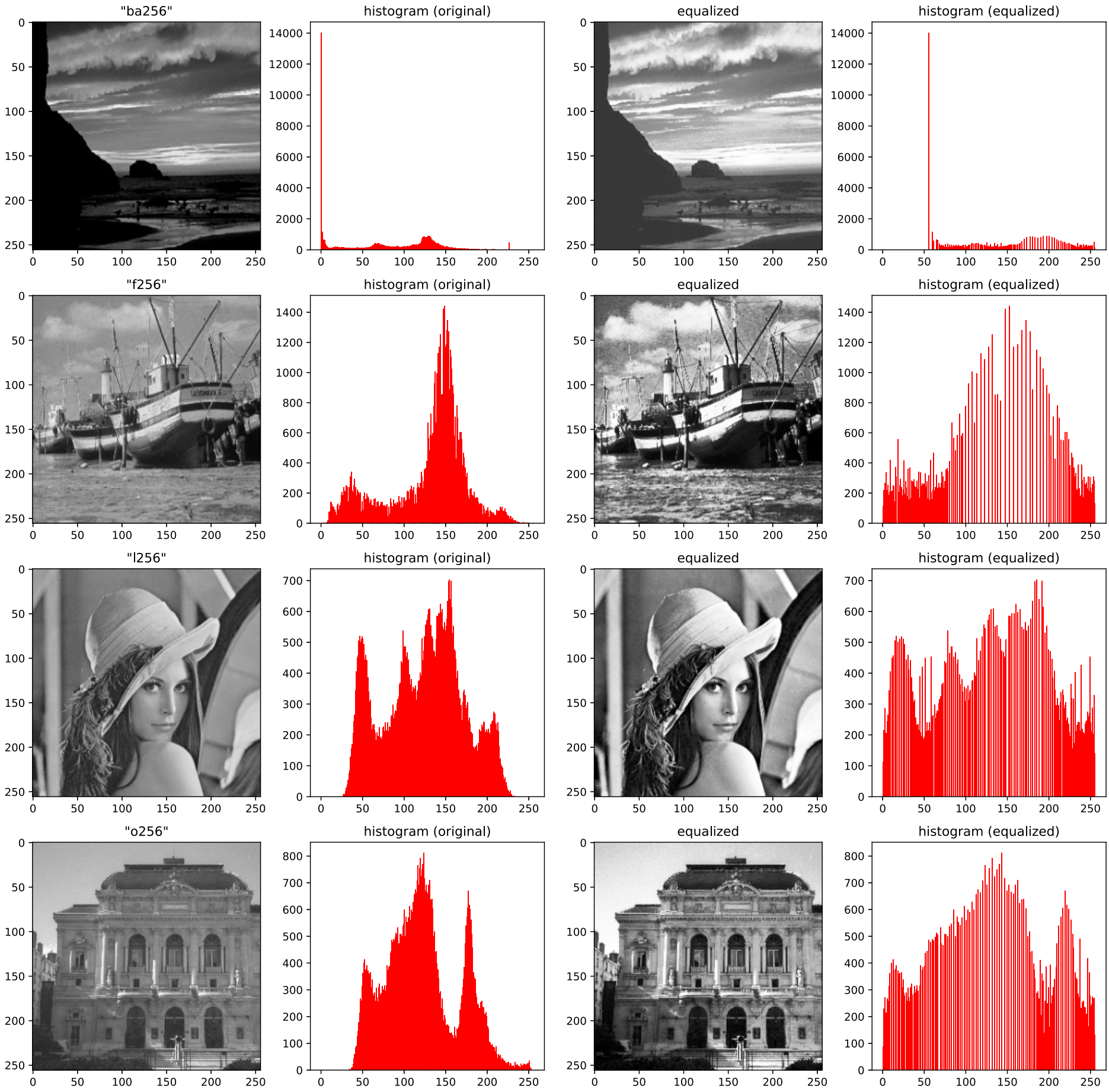
Histogram Equalization

In [5]:
```
def gen_histogram(image):
    histogram = np.zeros(256)

    height, width = image.shape
    for i in range(height):
        for j in range(width):
            histogram[image[i][j]] += 1

    return histogram
```

In [6]:
```
def equalize_histogram(image, histogram):
    height, width = image.shape

    new_levels = np.zeros(256)
    equalized = np.zeros((height, width))

    curr = 0
    for i in range(256):
        curr += histogram[i]
        new_levels[i] = round((curr * 255) / (height * width))

    for i in range(height):
        for j in range (width):
            equalized[i][j] = new_levels[image[i][j]]
    equalized = equalized.astype('uint8')

    return equalized
```

In [7]:
```
rows, cols = len(images), 4

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
```

```
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    original = image_dict['orig']
    hist_original = gen_histogram(original)

    equalized = equalize_histogram(original, hist_original)
    hist_equalized = gen_histogram(equalized)

    images[idx]['equalized'] = equalized

    axs[idx, 0].set_title('"{}"'.format(filename))
    axs[idx, 0].imshow(original, cmap='gray', vmin=0, vmax=255)

    axs[idx, 1].set_title('histogram (original)')
    axs[idx, 1].hist(original.flatten(), 256, [0, 256], color = 'r')

    axs[idx, 2].set_title('equalized'.format(filename))
    axs[idx, 2].imshow(equalized, cmap='gray', vmin=0, vmax=255)

    axs[idx, 3].set_title('histogram (equalized)')
    axs[idx, 3].hist(equalized.flatten(), 256, [0, 256], color = 'r')

    # Save pixel values of original image's histogram as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_hist' + ext_inp,
        hist_original,
        fmt=' %d',
        newline=' \n'
    )

    # Save pixel values of equalized image's histogram as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_hist_equalized' + ext_inp,
        hist_equalized,
        fmt=' %d',
        newline=' \n'
    )

# Save and display the figure
plt.savefig('histogram_comp.jpg')
plt.show()
```

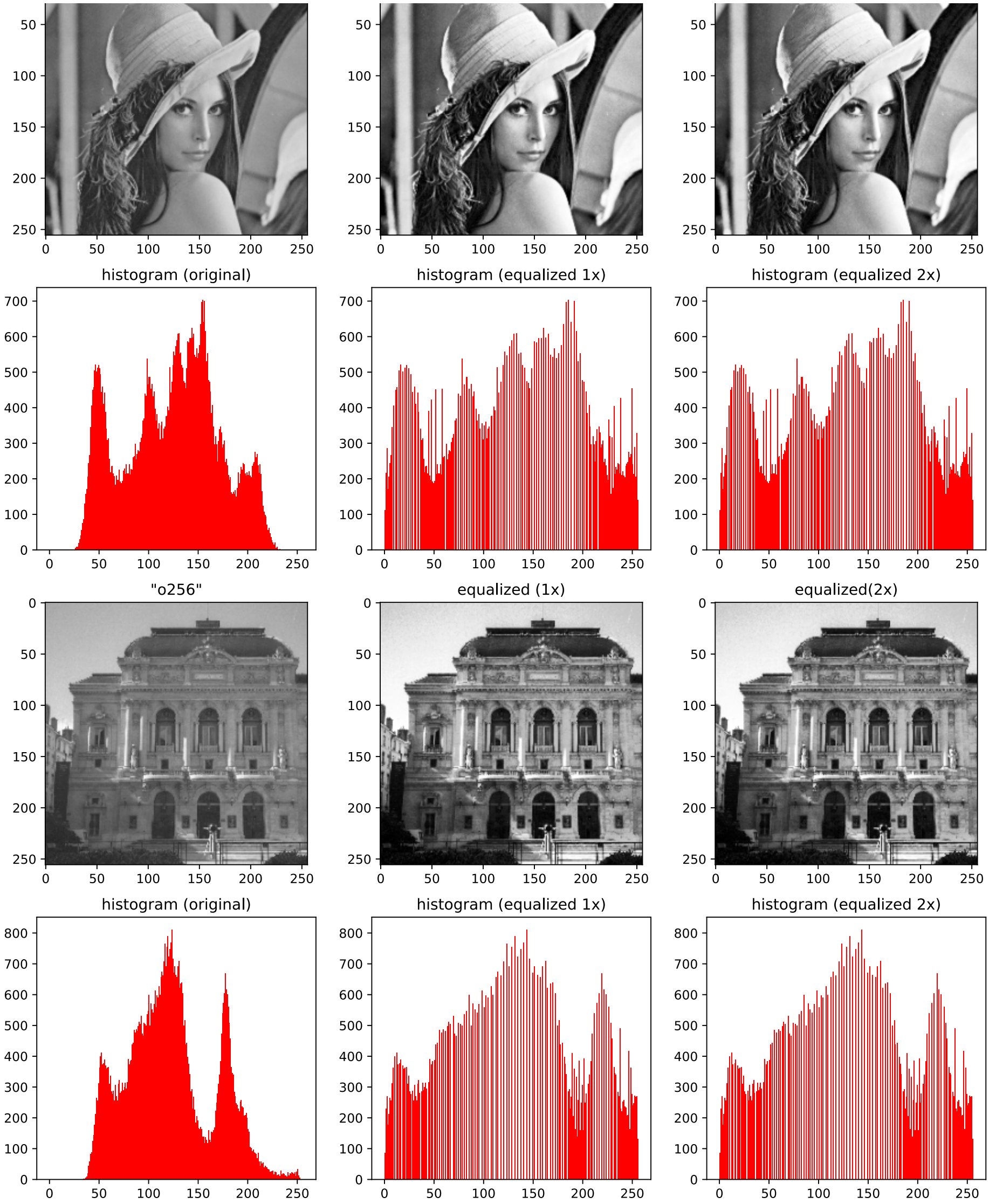


Histogram Equalization 2nd pass

In [8]:

```
rows, cols = 2 * len(images), 3

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    original = image_dict['orig']
    hist_original = gen_histogram(original)

    equalized = image_dict['equalized']
    hist_equalized = gen_histogram(equalized)

    equalized_2 = equalize_histogram(equalized, hist_equalized)
    hist_equalized_2 = gen_histogram(equalized_2)

    axs[2 * idx, 0].set_title('"{}"'.format(filename))
    axs[2 * idx, 0].imshow(original, cmap='gray', vmin=0, vmax=255)

    axs[2 * idx, 1].set_title('equalized (1x)'.format(filename))
    axs[2 * idx, 1].imshow(equalized, cmap='gray', vmin=0, vmax=255)

    axs[2 * idx, 2].set_title('equalized(2x)'.format(filename))
    axs[2 * idx, 2].imshow(equalized_2, cmap='gray', vmin=0, vmax=255)

    axs[2 * idx + 1, 0].set_title('histogram (original)')
    axs[2 * idx + 1, 0].hist(original.flatten(), 256, [0, 256], color = 'r')
```

```
    axs[2 * idx + 1, 1].set_title('histogram (equalized 1x)')
    axs[2 * idx + 1, 1].hist(equalized.flatten(), 256, [0, 256], color = 'r')

    axs[2 * idx + 1, 2].set_title('histogram (equalized 2x)')
    axs[2 * idx + 1, 2].hist(equalized_2.flatten(), 256, [0, 256], color = 'r')

    # Save pixel values of equalized image's histogram as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_hist_equalized_2x' + ext_inp,
        hist_equalized_2,
        fmt=' %d',
        newline=' \n'
    )

# Save and display the figure
plt.savefig('histogram_comp_2x.jpg')
plt.show()
```



Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 8

Write a Program to implement additive noise corruption of an image by manipulating `p%` randomly selected pixel values by an amount of `q%` (may be a rand function from `0%` to `15%`) for respective gray values.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'f256',
    'l256',
    'o256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

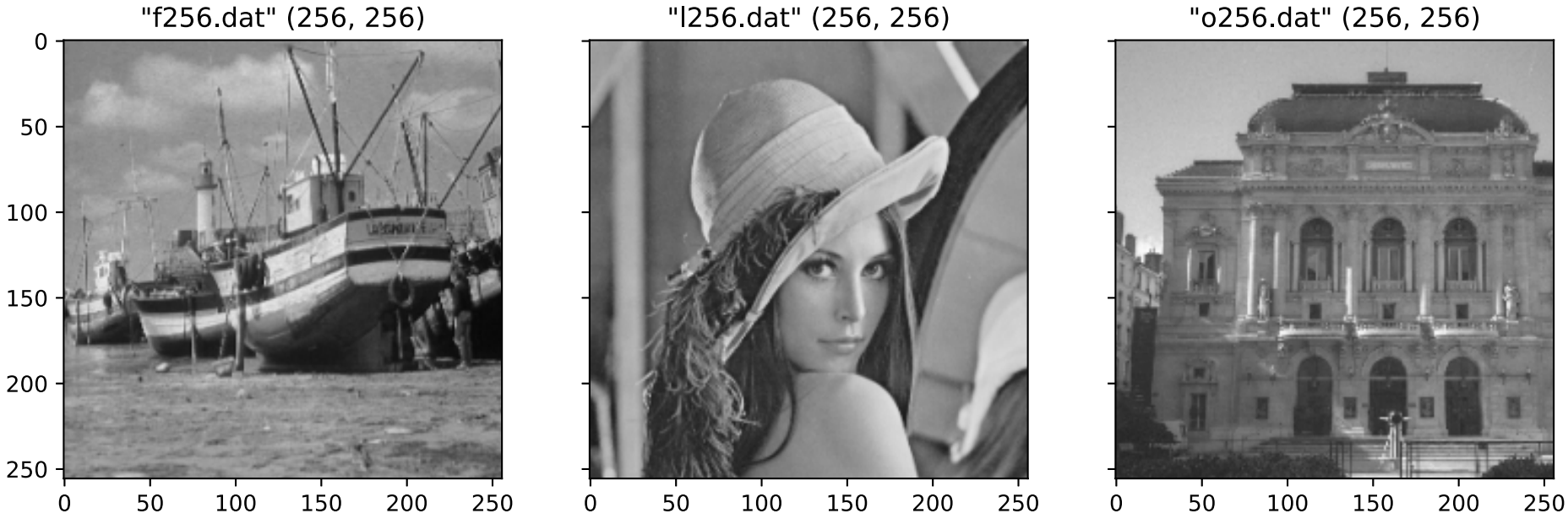
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image,
        'equalized': None
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = 1

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[idx].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[idx].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
```

```
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



Additive Noise Corruption

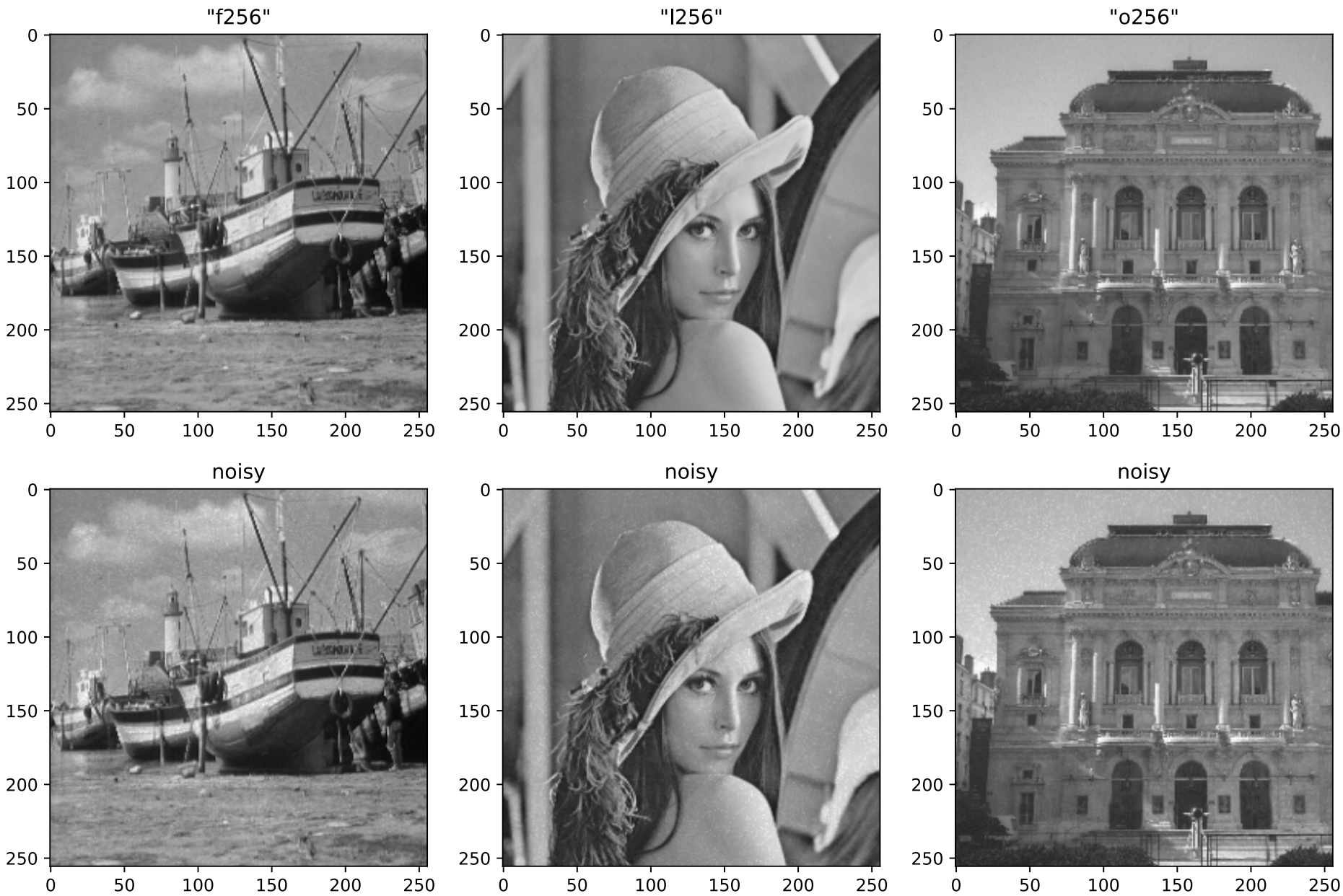
In [5]:

```
def add_noise(image, p: int, q: int):
    height, width = image.shape

    n_pixels = height * width
    n_p = (n_pixels * p) // 100
    pixels = set()
    for i in range(n_p):
        while True:
            curr = random.randint(0, n_pixels)
            row = curr // width
            col = curr % width
            if (row, col) not in pixels:
                pixels.add((row, col))
                break

    noisy_image = np.zeros((height, width))

    def min(a, b):
        return a if a < b else b

    for i in range(height):
        for j in range(width):
            noisy_image[i][j] = image[i][j]

    for row, col in pixels:
        noisy_image[row][col] = min(
            255,
            int(noisy_image[row][col]) + int(image[row][col] * (random.randint(0, q) / 100))
        )

    noisy_image = noisy_image.astype('uint8')

    return noisy_image
```

In [6]:

```
rows, cols = 2, len(images)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    orig = image_dict['orig']
    noisy = add_noise(orig, 25, 15)

    axs[0, idx].set_title('"{}"'.format(filename))
    axs[0, idx].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[1, idx].set_title('noisy'.format(filename))
    axs[1, idx].imshow(noisy, cmap='gray', vmin=0, vmax=255)

    # Save pixel values of original image's histogram as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_noisy' + ext_inp,
        noisy,
```

```
        fmt=' %d',
        newline=' \n'
    )

    # Save noisy image as .bmp file
    plt.imsave(
        path_out_conv + ext_out[1:] + '/' + filename + '_noisy' + ext_out,
        noisy,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Save and display the figure
plt.savefig('add_noise.jpg')
plt.show()
```



Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 9

Do the image averaging operation for `K = 8, 16, 32` number images (changing the value of `p` and `q`) and find the difference between the original and averaged image.

Plot the histogram of difference image. Repeat the steps for all `K`. Show the histogram in all cases.

Observe the shifting in width and the mean position of the histogram of difference images. Plot the histogram of the difference image for all three cases.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'f256',
    'l256',
    'o256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

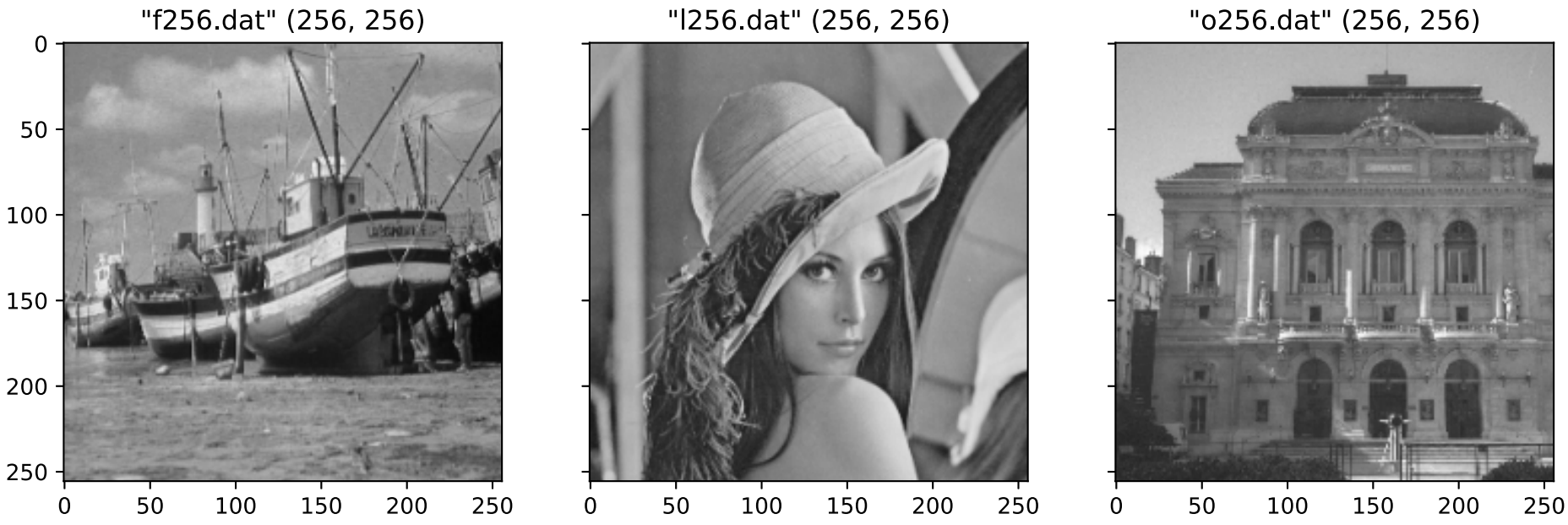
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image,
        'equalized': None
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = 1

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[idx].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[idx].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )
```

```
    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



Additive Noise Corruption

In [5]:

```
def add_noise(image, p: int, q: int):
    height, width = image.shape

    n_pixels = height * width
    n_p = (n_pixels * p) // 100
    pixels = set()
    for i in range(n_p):
        while True:
            curr = random.randint(0, n_pixels - 1)
            row = curr // width
            col = curr % width
            if (row, col) not in pixels:
                pixels.add((row, col))
                break

    noisy_image = np.zeros((height, width))

    def min(a, b):
        return a if a < b else b

    for i in range(height):
        for j in range(width):
            noisy_image[i][j] = image[i][j]

    for row, col in pixels:
        noisy_image[row][col] = min(
            255,
            int(noisy_image[row][col]) + int(image[row][col] * (random.randint(0, q) / 100))
        )

    noisy_image = noisy_image.astype('uint8')

    return noisy_image
```

Image Averaging

In [6]:

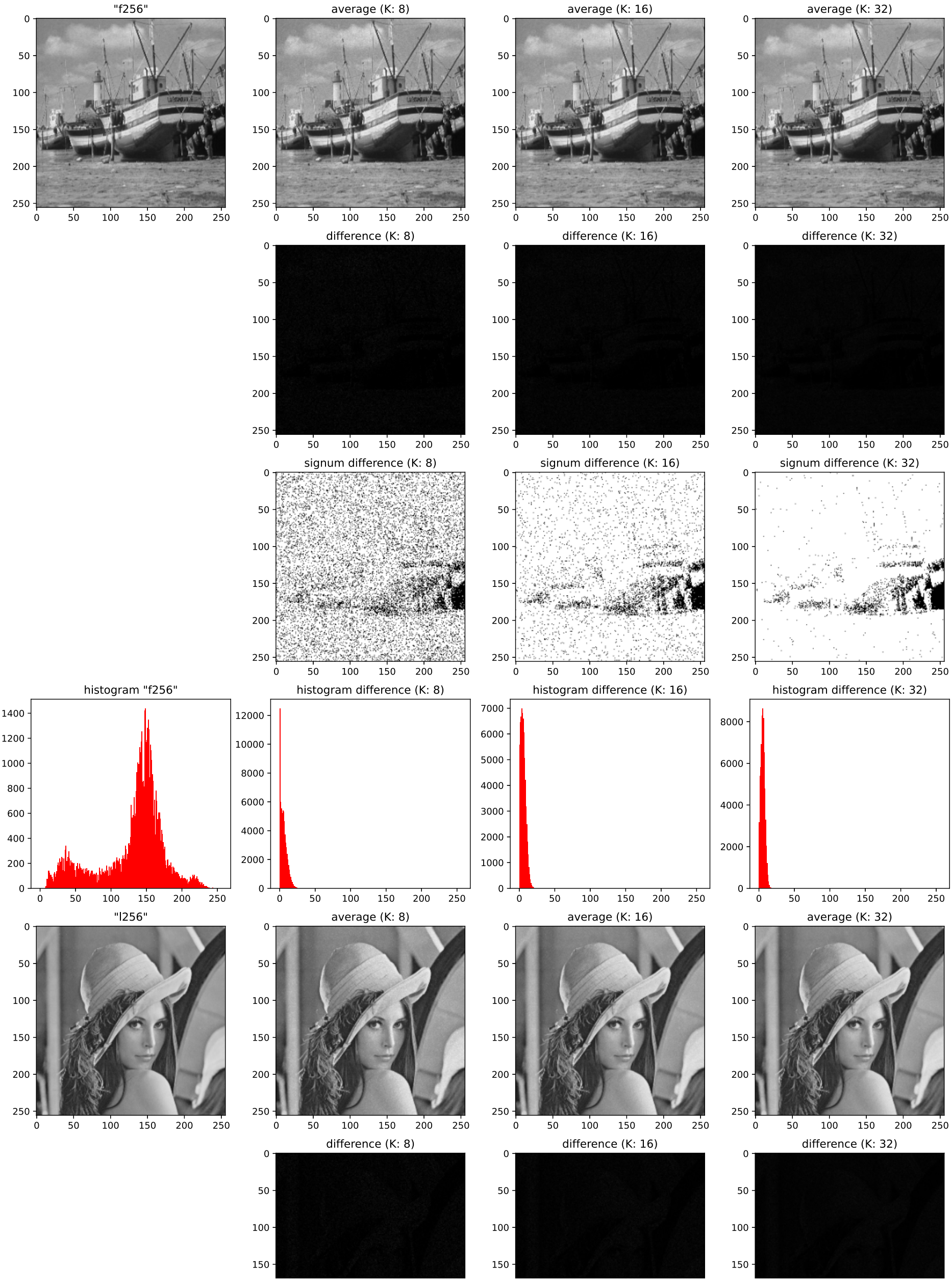
```
def average_images(images):
    height, width = images[0].shape
    img = np.zeros((height, width))

    for i in range(height):
        for j in range(width):
            sum = 0
            for k in range(len(images)):
                sum += images[k][i][j]
            img[i][j] = sum // len(images)
    img.astype('uint8')

    return img
```

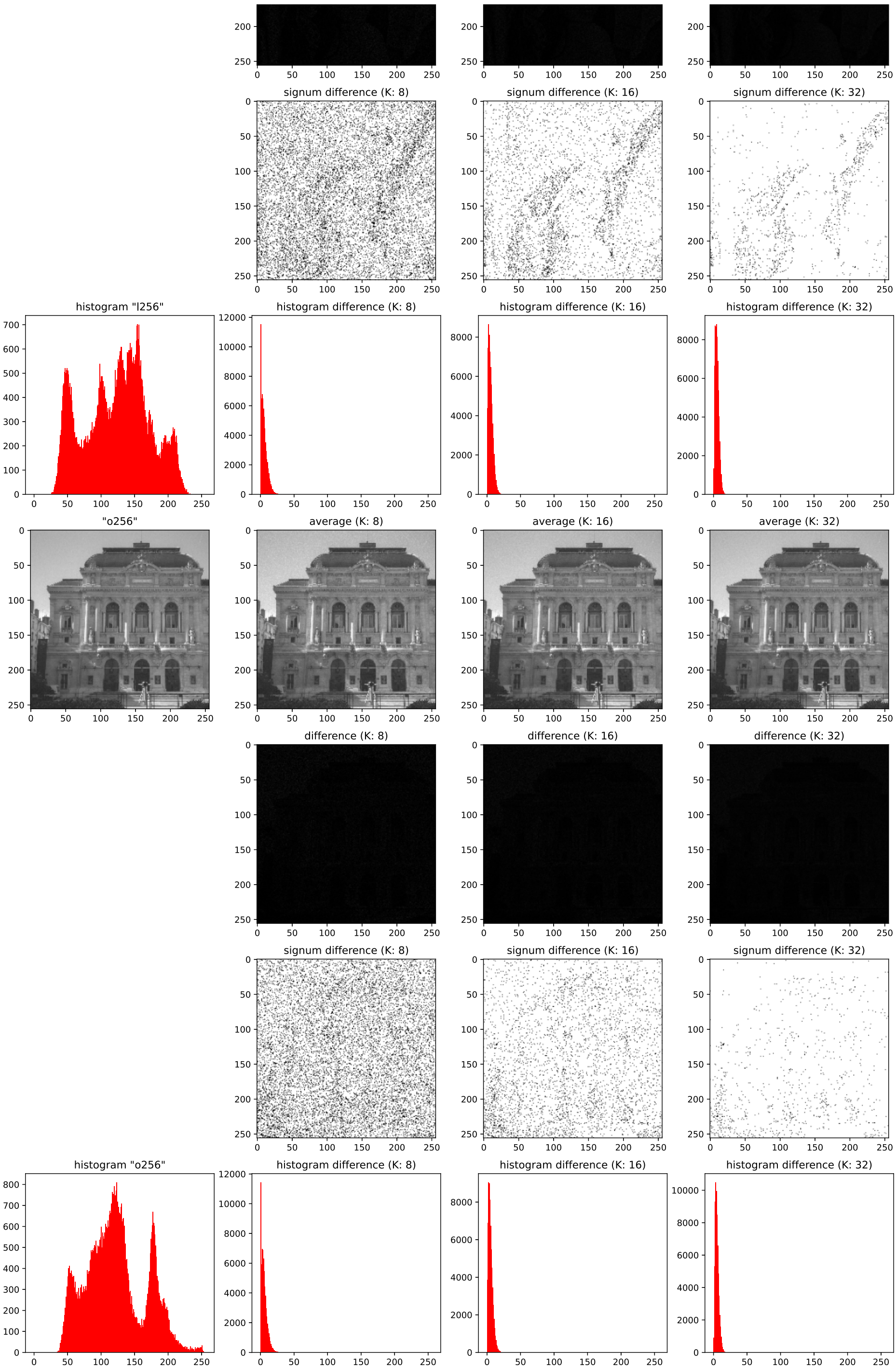
Image Difference

In [7]:
```
def difference_image(image_a, image_b):
    return abs(image_a - image_b)

def difference_image_signum(image_a, image_b):
    height, width = image_a.shape
    img = np.zeros((height, width))

    for i in range(height):
        for j in range(width):
            img[i][j] = 0 if image_a[i][j] == image_b[i][j] else 255
    img.astype('uint8')

    return img
```

In [8]:
```
rows, cols = 4 * len(images), 4

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    orig = image_dict['orig']
    noisy = []
    for i in range(32):
        noisy.append(add_noise(orig, 25, 35))

    g = [8, 16, 32]
    avgs = list(map(lambda x: average_images(noisy[:x]), g))
    diff = list(map(lambda x: difference_image(x, orig), avgs))
    diff_s = list(map(lambda x: difference_image_signum(x, orig), avgs))

    axs[4 * idx, 0].set_title('"{}"'.format(filename))
    axs[4 * idx, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[4 * idx + 1, 0].spines['bottom'].set_color('white')
    axs[4 * idx + 1, 0].spines['top'].set_color('white')
    axs[4 * idx + 1, 0].spines['left'].set_color('white')
    axs[4 * idx + 1, 0].spines['right'].set_color('white')
    axs[4 * idx + 1, 0].tick_params(axis='x', colors='white')
    axs[4 * idx + 1, 0].tick_params(axis='y', colors='white')

    axs[4 * idx + 2, 0].spines['bottom'].set_color('white')
    axs[4 * idx + 2, 0].spines['top'].set_color('white')
    axs[4 * idx + 2, 0].spines['left'].set_color('white')
    axs[4 * idx + 2, 0].spines['right'].set_color('white')
    axs[4 * idx + 2, 0].tick_params(axis='x', colors='white')
    axs[4 * idx + 2, 0].tick_params(axis='y', colors='white')

    axs[4 * idx + 3, 0].set_title(f'histogram "{filename}"')
    axs[4 * idx + 3, 0].hist(orig.flatten(), 256, [0, 256], color = 'r')

    for (i, item) in enumerate(avgs):
        axs[4 * idx, i + 1].set_title(f'average (K: {g[i]})')
        axs[4 * idx, i + 1].imshow(item, cmap='gray', vmin=0, vmax=255)

        axs[4 * idx + 1, i + 1].set_title(f'difference (K: {g[i]})')
        axs[4 * idx + 1, i + 1].imshow(diff[i], cmap='gray', vmin=0, vmax=255)

        axs[4 * idx + 2, i + 1].set_title(f'signum difference (K: {g[i]})')
        axs[4 * idx + 2, i + 1].imshow(diff_s[i], cmap='gray', vmin=0, vmax=255)

        axs[4 * idx + 3, i + 1].set_title(f'histogram difference (K: {g[i]})')
        axs[4 * idx + 3, i + 1].hist(diff[i].flatten(), 256, [0, 256], color = 'r')

        # Save pixel values of original image's histogram as a 2D matrix in a .dat file
        np.savetxt(
            path_out_conv + ext_inp[1:] + '/' + filename + f'_avg_{g[i]}' + ext_inp,
            item,
            fmt=' %d',
            newline=' \n'
        )

        # Save noisy image as .bmp file
        plt.imsave(
            path_out_conv + ext_out[1:] + '/' + filename + f'_avg_{g[i]}' + ext_out,
            item,
            cmap='gray',
            vmin=0,
            vmax=255
        )

        # Save pixel values of original image's histogram as a 2D matrix in a .dat file
        np.savetxt(
            path_out_conv + ext_inp[1:] + '/' + filename + f'_diff_{g[i]}' + ext_inp,
            diff[i],
            fmt=' %d',
```

```
            newline=' \n'
        )

        # Save noisy image as .bmp file
        plt.imsave(
            path_out_conv + ext_out[1:] + '/' + filename + f'_avg_{g[i]}' + ext_out,
            diff[i],
            cmap='gray',
            vmin=0,
            vmax=255
        )

# Save and display the figure
plt.savefig('noise_average.jpg')
plt.show()
```





Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 10

Write a Program to implement spatial mean operation (say considering 3×3 window) and applying it on a gray scale noisy image. Show the filtering effect after the variable window size (5×5 , 7×7). Discuss the limiting effect of repeatedly applying a 3×3 low pass spatial filter to a digital image (apply 3×3 window two times i.e. twice). Show that the filtering results are equivalent i.e. output image obtained after applying 3×3 window twice is equivalent applying 5×5 window once.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'f256',
    'l256',
    'o256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

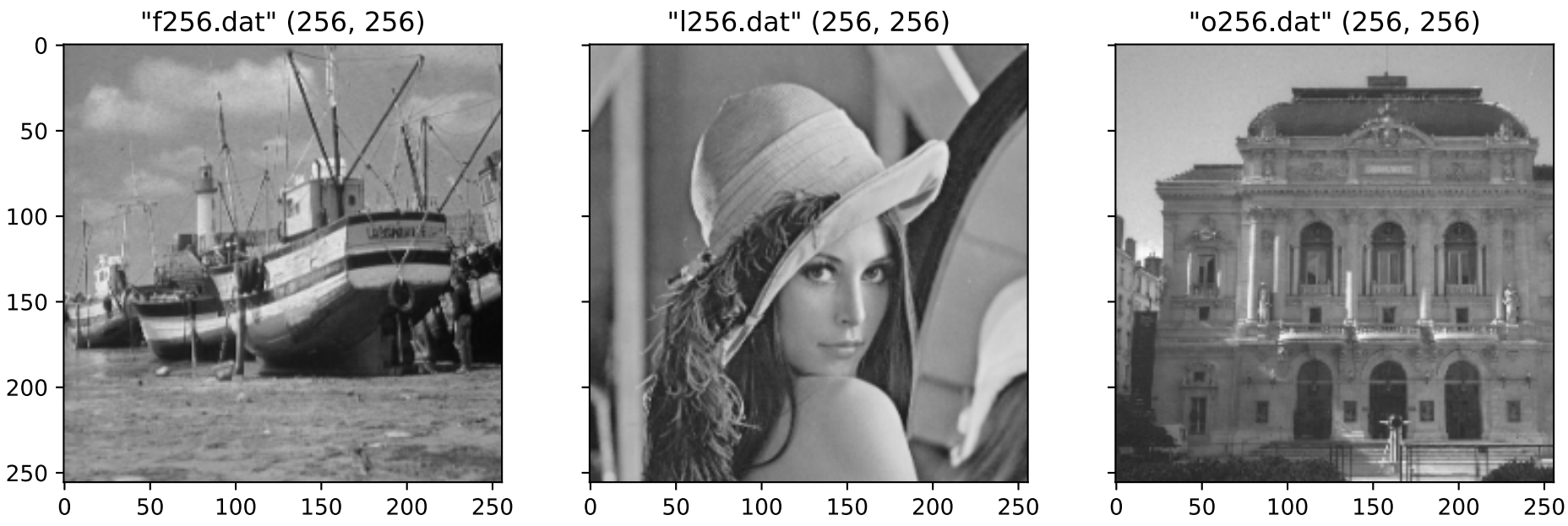
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image,
        'equalized': None
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = 1

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[idx].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[idx].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save original image as .bmp file
    plt.imsave(
```

```
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



Additive Noise Corruption

In [5]:

```
def add_noise(image, p: int, q: int):
    height, width = image.shape

    n_pixels = height * width
    n_p = (n_pixels * p) // 100
    pixels = set()
    for i in range(n_p):
        while True:
            curr = random.randint(0, n_pixels - 1)
            row = curr // width
            col = curr % width
            if (row, col) not in pixels:
                pixels.add((row, col))
                break

    noisy_image = np.zeros((height, width))

    def min(a, b):
        return a if a < b else b

    for i in range(height):
        for j in range(width):
            noisy_image[i][j] = image[i][j]

    for row, col in pixels:
        noisy_image[row][col] = min(
            255,
            int(noisy_image[row][col]) + int(image[row][col] * (random.randint(0, q) / 100))
        )

    noisy_image = noisy_image.astype('uint8')

    return noisy_image
```

Image Difference

In [6]:

```
def difference_image(image_a, image_b):
    height, width = image_a.shape
    img = abs(image_a - image_b)
    for i in range(height):
        for j in range(width):
            img[i][j] = 0 if img[i][j] < 0 else img[i][j]
    return img
```

Image Filtering

In [7]:

```
def mean(elems):
    sum = 0
    for i in elems:
```

```
        sum += i
    return sum // len(elems)
```

In [8]:

```
def filter_img(image, operation, w_size):
    height, width = image.shape
    img = np.zeros(image.shape)

    def get_pixel(i, j):
        return image[i][j] if (i >= 0 and j >=0) and (i < height and j < width) else 0

    p_list = list(range(w_size))
    for i in range(w_size):
        p_list[i] -= w_size // 2

    for i in range(height):
        for j in range(width):
            elems = []
            for m in p_list:
                for n in p_list:
                    elems.append(get_pixel(i + m, j + n))
            img[i][j] = operation(elems)
    img.astype('uint8')

    return img
```

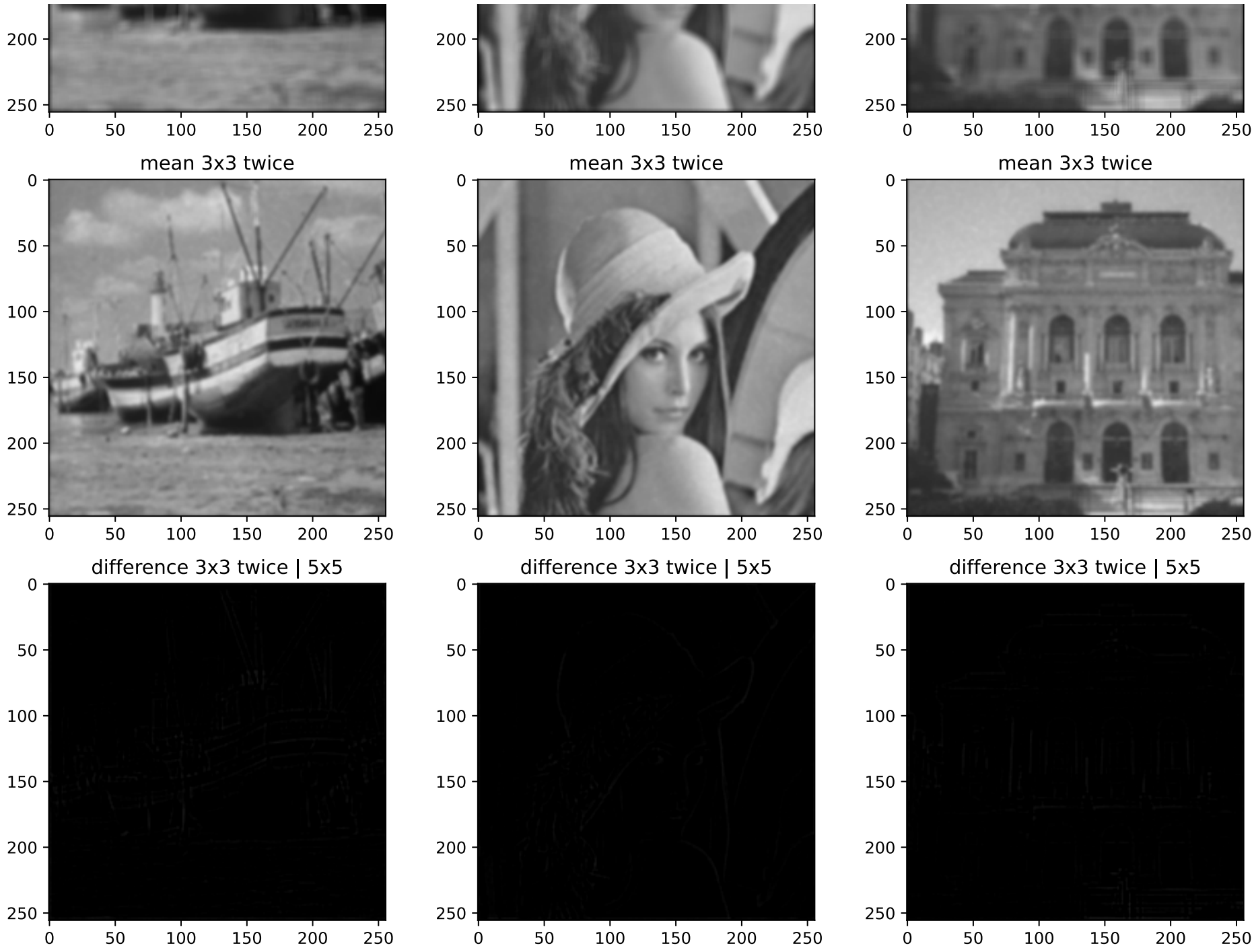
In [9]:

```
def save_dat(filename, data):
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + ext_inp,
        data,
        fmt=' %d',
        newline=' \n'
    )

def save_img(filename, image):
    plt.imsave(
        path_out_conv + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )
```

In [10]:

```
rows, cols = 7, len(images)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    orig = image_dict['orig']
    noisy = add_noise(orig, 25, 15)

    axs[0, idx].set_title(f'"{filename}"')
    axs[0, idx].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[1, idx].set_title(f'noisy')
    axs[1, idx].imshow(noisy, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_noisy', noisy)
    save_img(filename + f'_noisy', noisy)

    mean_3 = filter_img(noisy, mean, 3)
    axs[2, idx].set_title(f'mean 3x3')
    axs[2, idx].imshow(mean_3, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_mean_3', mean_3)
    save_img(filename + f'_mean_3', mean_3)

    mean_5 = filter_img(noisy, mean, 5)
    axs[3, idx].set_title(f'mean 5x5')
    axs[3, idx].imshow(mean_5, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_mean_5', mean_5)
    save_img(filename + f'_mean_5', mean_5)

    mean_7 = filter_img(noisy, mean, 7)
    axs[4, idx].set_title(f'mean 7x7')
    axs[4, idx].imshow(mean_7, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_mean_7', mean_7)
    save_img(filename + f'_mean_7', mean_7)

    mean_3_x2 = filter_img(filter_img(noisy, mean, 3), mean, 3)
    axs[5, idx].set_title(f'mean 3x3 twice')
    axs[5, idx].imshow(mean_3_x2, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_mean_3_x2', mean_3_x2)
    save_img(filename + f'_mean_3_x2', mean_3_x2)

    diff = mean_3_x2 - mean_5
    axs[6, idx].set_title(f'difference 3x3 twice | 5x5')
    axs[6, idx].imshow(diff, cmap='gray', vmin=0, vmax=255)
```

```
    save_dat(filename + f'_diff_5__3_x2', diff)
    save_img(filename + f'_diff_5__3_x2', diff)

# Save and display the figure
plt.savefig('mean_filter.jpg')
plt.show()
```



Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 11

Write a program to implement both mode and median filtering operation and applying it on gray scale noisy image. Show the filtering effect for variable window size.

Show that the median filtering results are equivalent i.e. output image obtained after applying 3×3 window twice is equivalent applying 5×5 window once.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
from scipy import stats
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'f256',
    'l256',
    'o256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

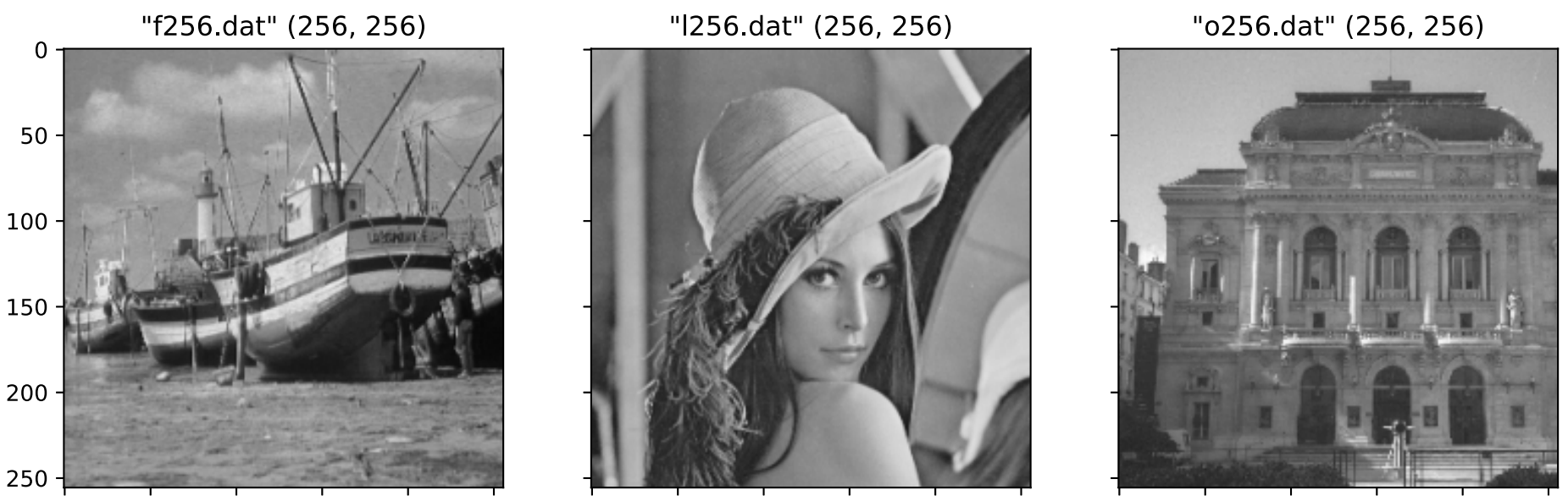
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image,
        'equalized': None
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = 1

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[idx].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[idx].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )
```

```
    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



Additive Noise Corruption

In [5]:

```
def add_noise(image, p: int, q: int):
    height, width = image.shape

    n_pixels = height * width
    n_p = (n_pixels * p) // 100
    pixels = set()
    for i in range(n_p):
        while True:
            curr = random.randint(0, n_pixels - 1)
            row = curr // width
            col = curr % width
            if (row, col) not in pixels:
                pixels.add((row, col))
                break

    noisy_image = np.zeros((height, width))

    def min(a, b):
        return a if a < b else b

    for i in range(height):
        for j in range(width):
            noisy_image[i][j] = image[i][j]

    for row, col in pixels:
        noisy_image[row][col] = min(
            255,
            int(noisy_image[row][col]) + int(image[row][col] * (random.randint(0, q) / 100))
        )

    noisy_image = noisy_image.astype('uint8')

    return noisy_image
```

Image Difference

In [6]:

```
def difference_image(image_a, image_b):
    height, width = image_a.shape
    img = abs(image_a - image_b)
    for i in range(height):
        for j in range(width):
            img[i][j] = 0 if img[i][j] < 0 else img[i][j]
    return img
```

Image Filtering

In [7]:

```
def mode(elems):
    return stats.mode(elems)[0][0]
```

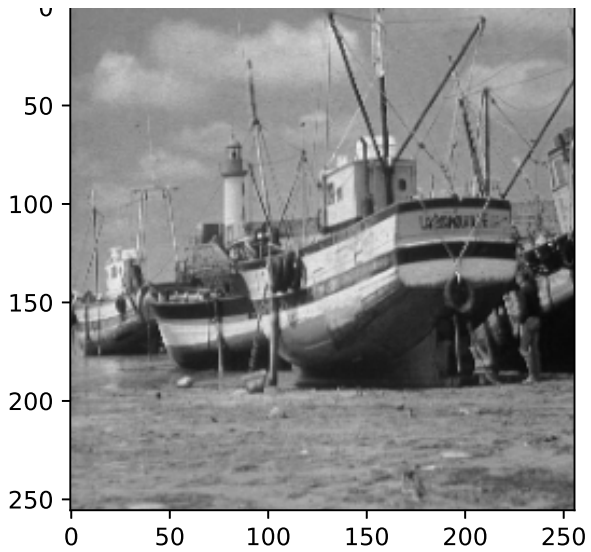
```
def median(elems):
    return np.median(elems)
```

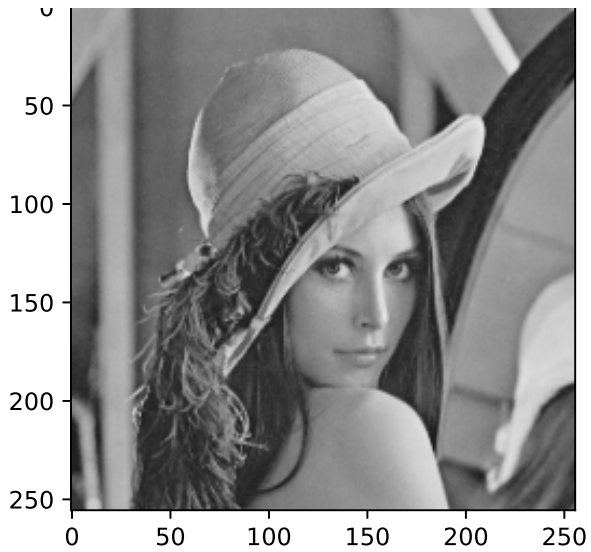
In [8]:
```
def filter_img(image, operation, w_size):
    height, width = image.shape
    img = np.zeros(image.shape)

    def get_pixel(i, j):
        return image[i][j] if (i >= 0 and j >=0) and (i < height and j < width) else 0

    p_list = list(range(w_size))
    for i in range(w_size):
        p_list[i] -= w_size // 2

    for i in range(height):
        for j in range(width):
            elems = []
            for m in p_list:
                for n in p_list:
                    elems.append(get_pixel(i + m, j + n))
            img[i][j] = operation(elems)
    img.astype('uint8')

    return img
```

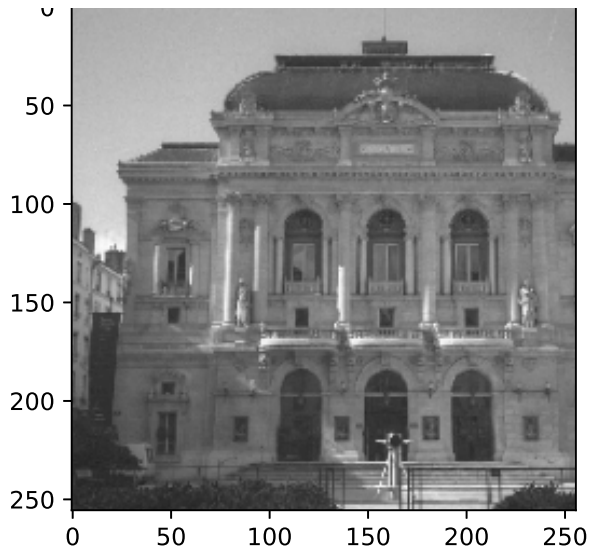
In [9]:
```
def save_dat(filename, data):
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + ext_inp,
        data,
        fmt=' %d',
        newline=' \n'
    )

def save_img(filename, image):
    plt.imsave(
        path_out_conv + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )
```

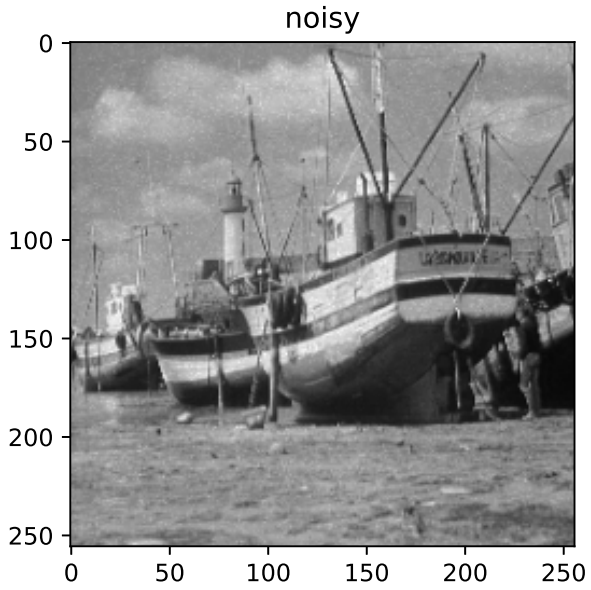
Mode Filtering

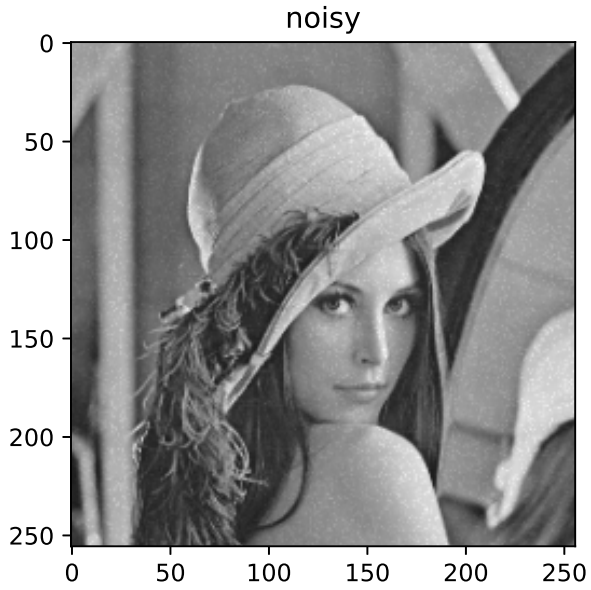
In [10]:
```
rows, cols = 5, len(images)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    orig = image_dict['orig']
    noisy = add_noise(orig, 25, 15)

    axs[0, idx].set_title(f'"{filename}"')
    axs[0, idx].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[1, idx].set_title(f'noisy')
    axs[1, idx].imshow(noisy, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_noisy', noisy)
    save_img(filename + f'_noisy', noisy)

    mode_3 = filter_img(noisy, mode, 3)
    axs[2, idx].set_title(f'mode 3x3')
    axs[2, idx].imshow(mode_3, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_mode_3', mode_3)
    save_img(filename + f'_mode_3', mode_3)

    mode_5 = filter_img(noisy, mode, 5)
    axs[3, idx].set_title(f'mode 5x5')
    axs[3, idx].imshow(mode_5, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_mode_5', mode_5)
    save_img(filename + f'_mode_5', mode_5)

    mode_7 = filter_img(noisy, mode, 7)
    axs[4, idx].set_title(f'mode 7x7')
    axs[4, idx].imshow(mode_7, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_mode_7', mode_7)
    save_img(filename + f'_mode_7', mode_7)

# Save and display the figure
plt.savefig('mode_filter.jpg')
plt.show()
```

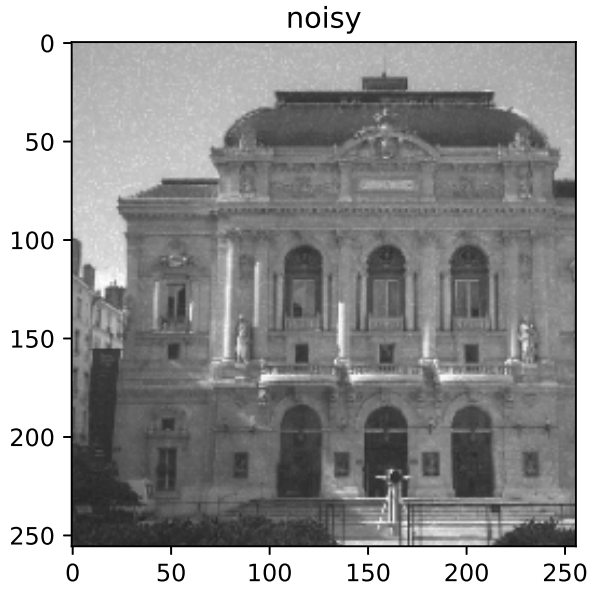


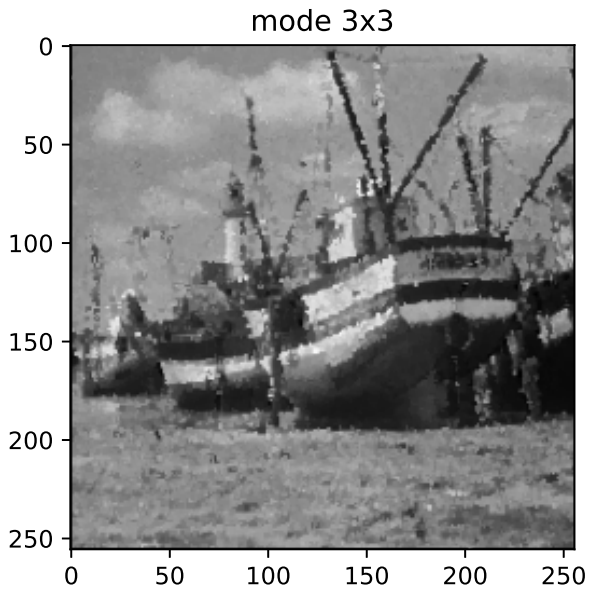


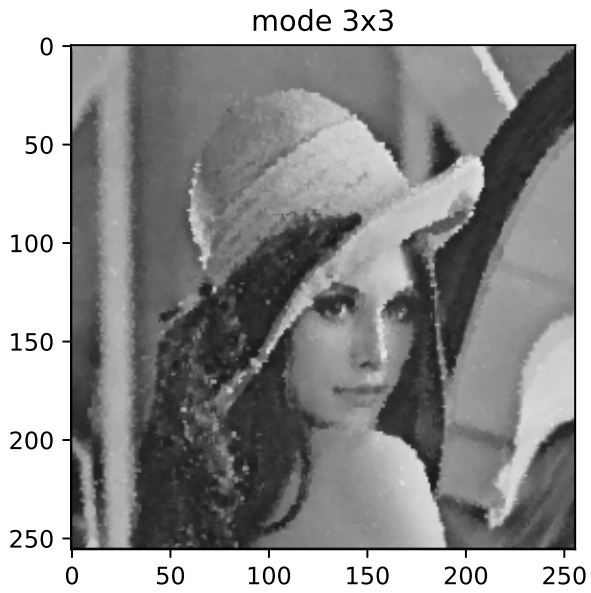


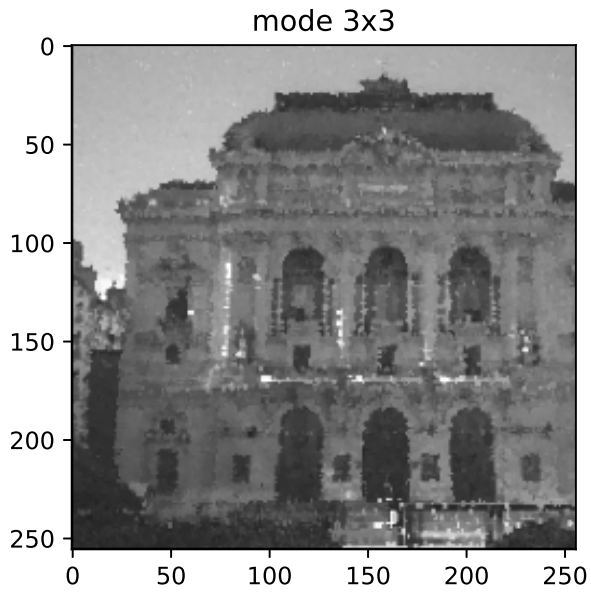

noisy

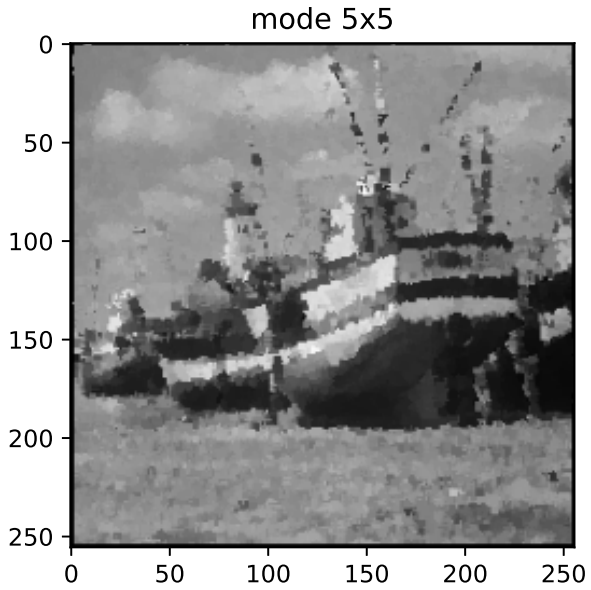

noisy

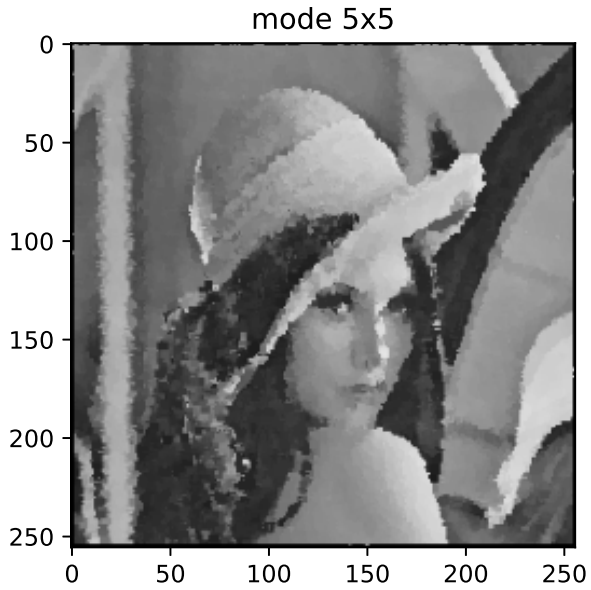

noisy

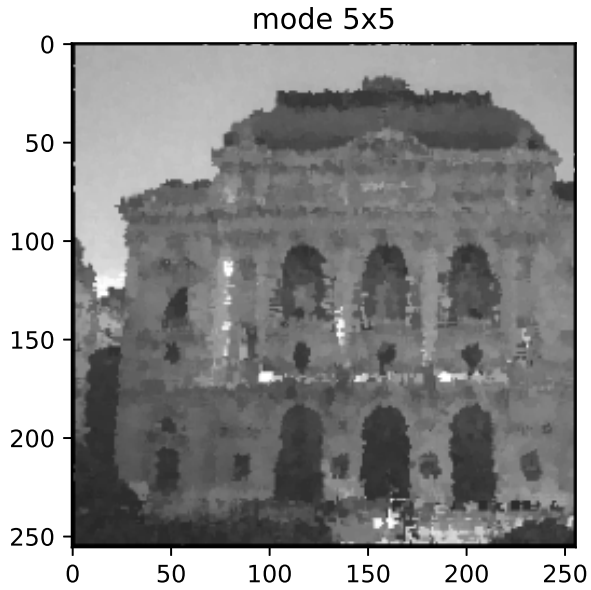

mode 3x3

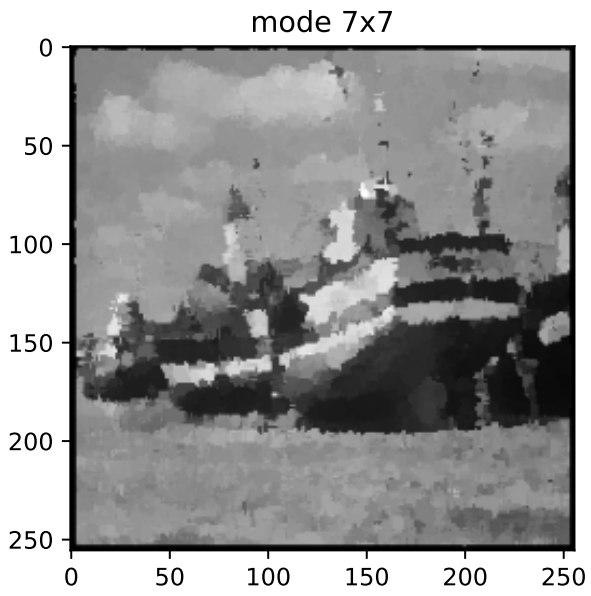

mode 3x3

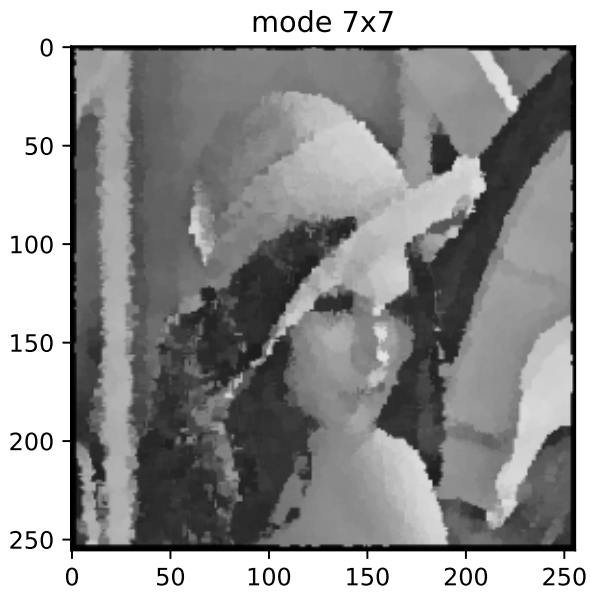

mode 3x3

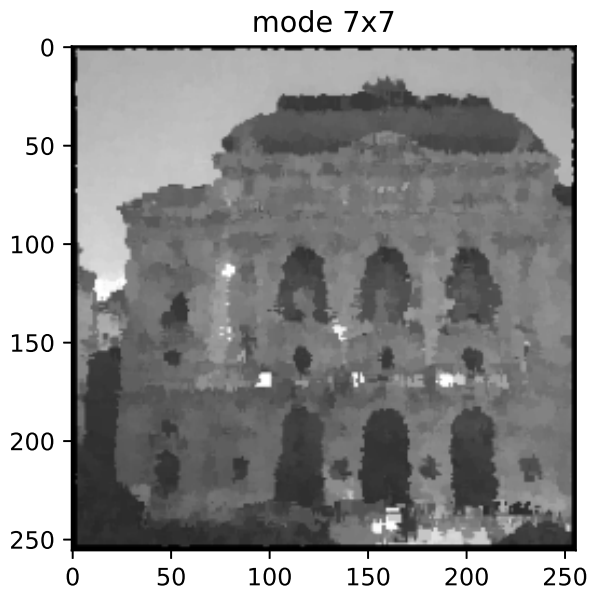

mode 5x5


mode 5x5


mode 5x5

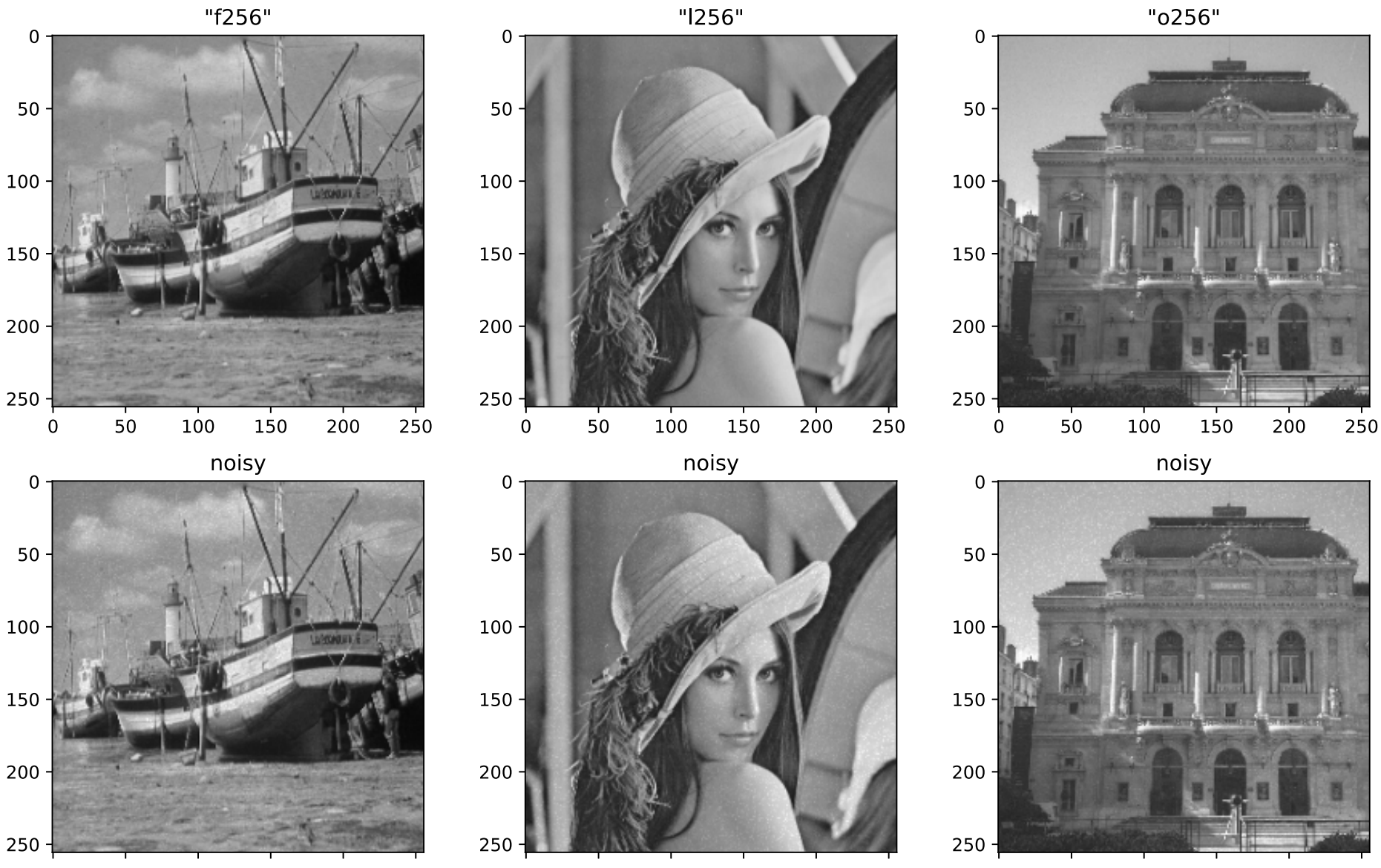

mode 7x7


mode 7x7


mode 7x7

Median Filtering

In [11]:
```
rows, cols = 7, len(images)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    orig = image_dict['orig']
    noisy = add_noise(orig, 25, 15)

    axs[0, idx].set_title(f'"{filename}"')
    axs[0, idx].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[1, idx].set_title(f'noisy')
    axs[1, idx].imshow(noisy, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_noisy', noisy)
    save_img(filename + f'_noisy', noisy)

    median_3 = filter_img(noisy, median, 3)
    axs[2, idx].set_title(f'median 3x3')
    axs[2, idx].imshow(median_3, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_median_3', median_3)
    save_img(filename + f'_median_3', median_3)

    median_5 = filter_img(noisy, median, 5)
    axs[3, idx].set_title(f'median 5x5')
    axs[3, idx].imshow(median_5, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_median_5', median_5)
    save_img(filename + f'_median_5', median_5)

    median_7 = filter_img(noisy, median, 7)
    axs[4, idx].set_title(f'median 7x7')
    axs[4, idx].imshow(median_7, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_median_7', median_7)
    save_img(filename + f'_median_7', median_7)

    median_3_x2 = filter_img(filter_img(noisy, median, 3), median, 3)
    axs[5, idx].set_title(f'median 3x3 twice')
    axs[5, idx].imshow(median_3_x2, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_median_3_x2', median_3_x2)
    save_img(filename + f'_median_3_x2', median_3_x2)

    diff = difference_image(median_5, median_3_x2)
    axs[6, idx].set_title(f'difference 3x3 twice | 5x5')
    axs[6, idx].imshow(diff, cmap='gray', vmin=0, vmax=255)
    save_dat(filename + f'_diff_5__3_x2', diff)
    save_img(filename + f'_diff_5__3_x2', diff)

# Save and display the figure
plt.savefig('median_filter.jpg')
plt.show()
```



median 3x3
median 3x3
median 3x3
median 5x5
median 5x5
median 5x5
median 7x7
median 7x7
median 7x7
median 3x3 twice
median 3x3 twice
median 3x3 twice
difference 3x3 twice | 5x5
difference 3x3 twice | 5x5
difference 3x3 twice | 5x5

Question 12

Write a program to calculate Δx and Δy for each pixel of an input image $f(x,y)$ using gradient operator.

```
-1  0      0 -1
 0  1      1  0
```

Replace each pixel point by $|\Delta x| + |\Delta y|$ and then implement image-sharpening operation.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'fu256',
    'l256',
    'n256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

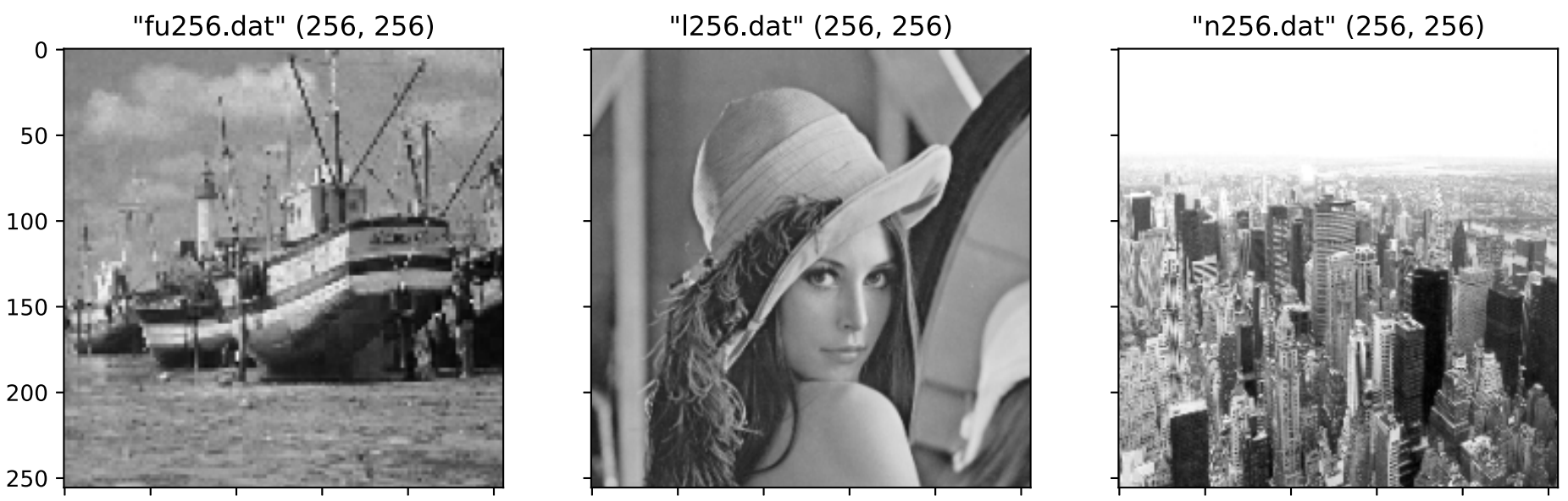
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image,
        'equalized': None
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = 1

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[idx].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[idx].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )
```

```
    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



Sharpen Images

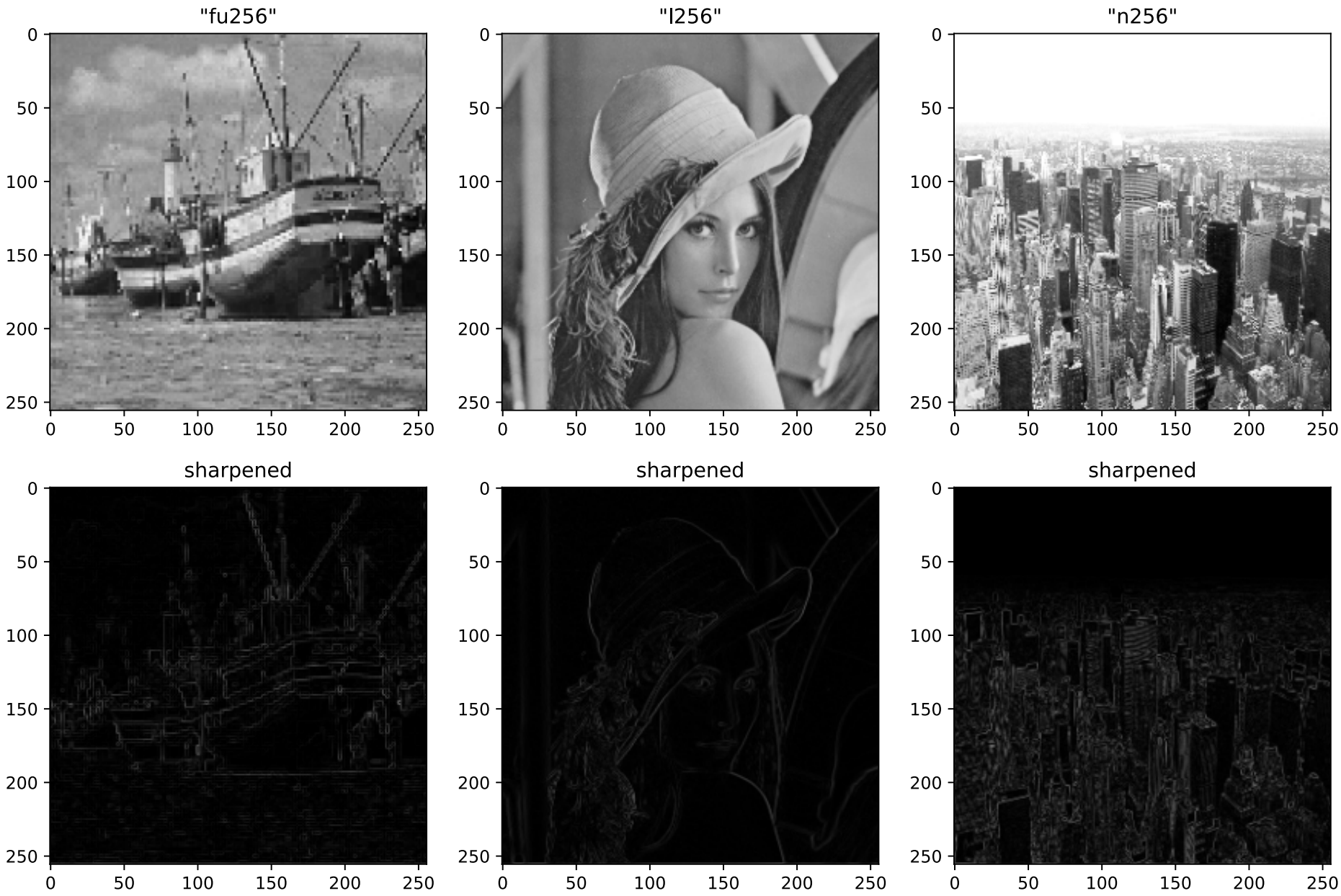
In [5]:

```
def sharpen_image(image):
    k_x = np.array([[-1, 0], [0, 1]], dtype='float32')
    k_y = np.array([[0, -1], [1, 0]], dtype='float32')

    height, width = image.shape
    tmp = np.zeros((height + 1, width + 1))
    img = np.zeros((height, width))

    for i in range(height):
        for j in range(width):
            tmp[i][j] = image[i][j]

    for i in range(height):
        for j in range(width):
            s_x = 0
            for m in [0, 1]:
                for n in [0, 1]:
                    s_x += tmp[i + m][j + n] * k_x[m][n]
            s_x = abs(s_x // 4)

            s_y = 0
            for m in [0, 1]:
                for n in [0, 1]:
                    s_y += tmp[i + m][j + n] * k_y[m][n]
            s_y = abs(s_y // 4)

            img[i][j] = s_x + s_y
    img.astype('uint8')
    return img
```

In [6]:

```
rows, cols = 2, len(images)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    orig = image_dict['orig']
    sharp = sharpen_image(orig)

    axs[0, idx].set_title('"{}"'.format(filename))
    axs[0, idx].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[1, idx].set_title('sharpened'.format(filename))
    axs[1, idx].imshow(sharp, cmap='gray', vmin=0, vmax=255)

    # Save pixel values of original image's histogram as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + f'_sharpened' + ext_inp,
```

```
        sharp,
        fmt=' %d',
        newline=' \n'
    )

    # Save noisy image as .bmp file
    plt.imsave(
        path_out_conv + ext_out[1:] + '/' + filename + f'_sharpened' + ext_out,
        sharp,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Save and display the figure
plt.savefig('sharpened_image.jpg')
plt.show()
```



Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 13

Write a program to calculate Δx and Δy for each pixel of an input image $f(x,y)$ using Sobel's operator (gradient operator).

```
-1  0  1     -1 -2 -1
-2  0  2      0  0  0
-1  0  1      1  2  1
```

Replace each pixel point by $|\Delta x| + |\Delta y|$ and then implement image-sharpening operation.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'fu256',
    'l256',
    'n256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

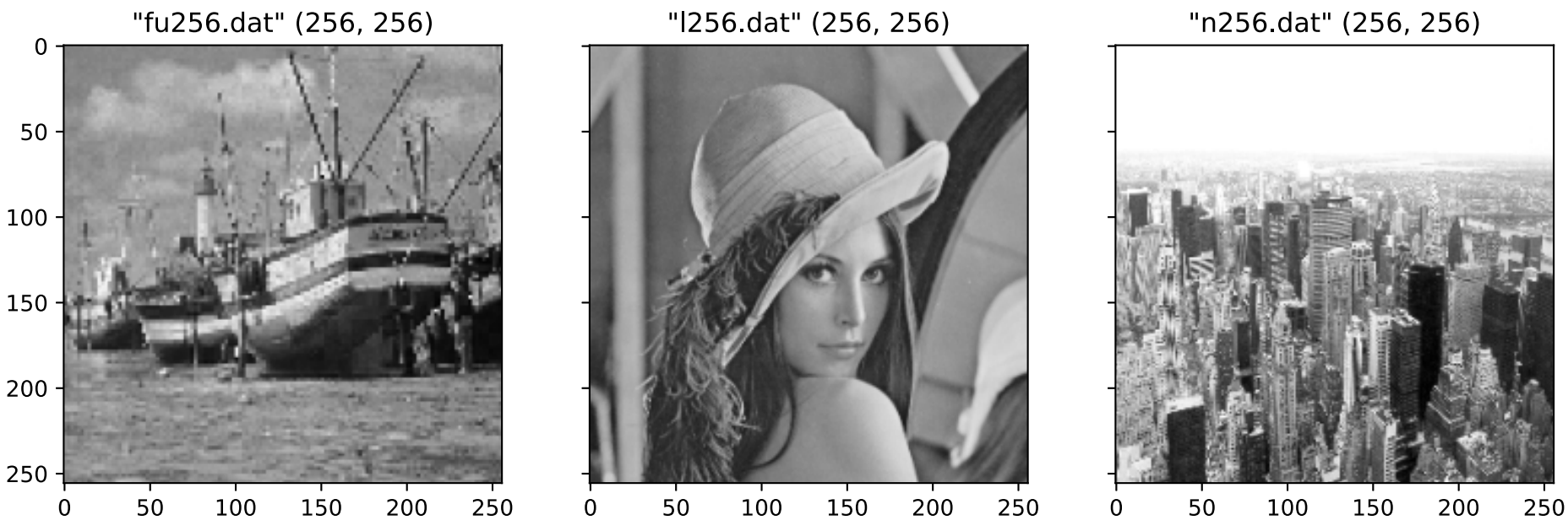
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image,
        'equalized': None
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = 1

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[idx].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[idx].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )
```

```
    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



Sharpen Images

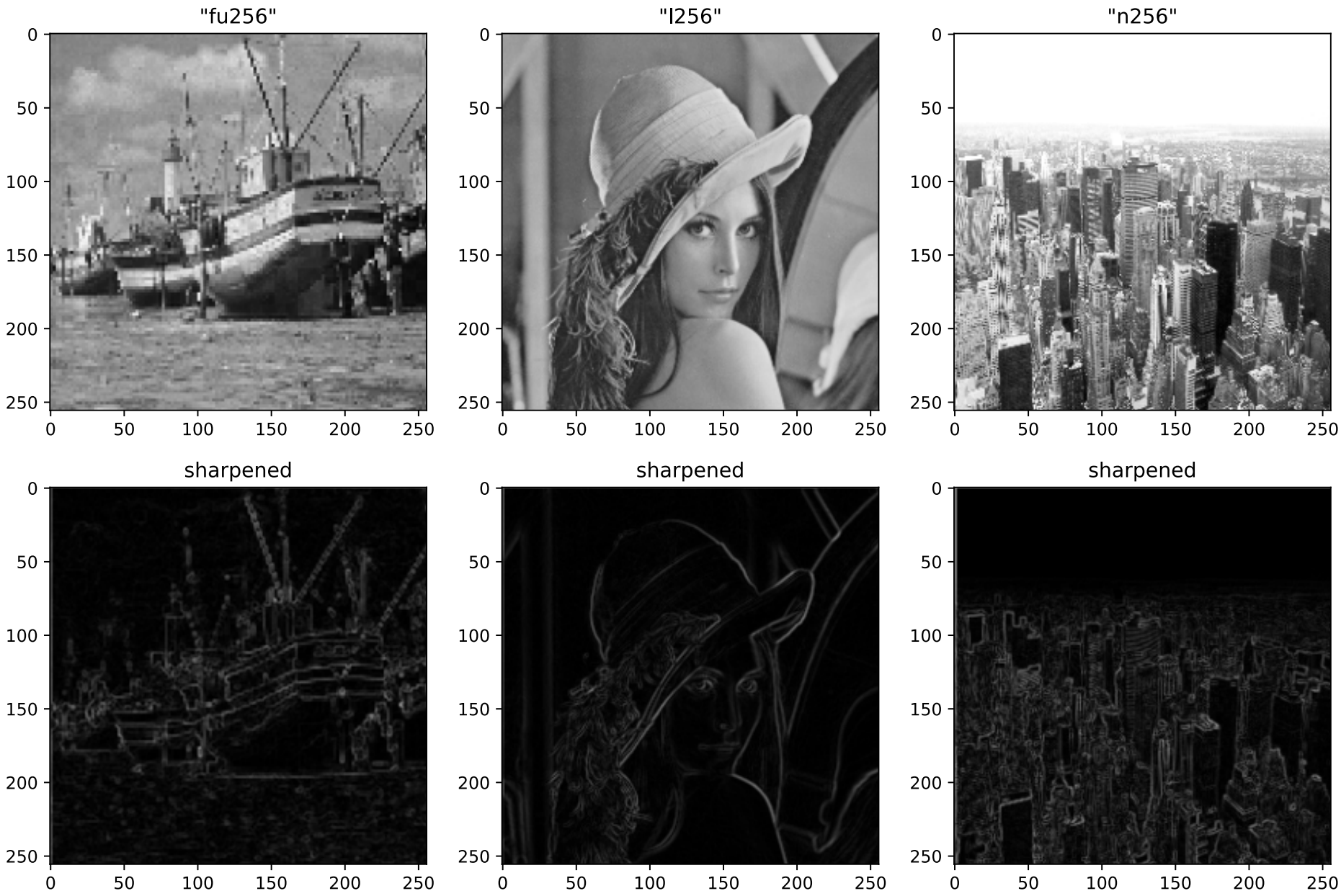
In [5]:

```
def sharpen_image(image):
    k_x = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype='float32')
    k_y = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], dtype='float32')

    height, width = image.shape
    tmp = np.zeros((height + 2, width + 2))
    img = np.zeros((height, width))

    for i in range(height):
        for j in range(width):
            tmp[i + 1][j + 1] = image[i][j]

    for i in range(height):
        for j in range(width):
            s_x = 0
            for m in [-1, 0, 1]:
                for n in [-1, 0, 1]:
                    s_x += tmp[i + m][j + n] * k_x[m + 1][n + 1]
            s_x = abs(s_x // 9)

            s_y = 0
            for m in [-1, 0, 1]:
                for n in [-1, 0, 1]:
                    s_y += tmp[i + m][j + n] * k_y[m + 1][n + 1]
            s_y = abs(s_y // 9)

            img[i][j] = s_x + s_y
    img.astype('uint8')
    return img
```

In [6]:

```
rows, cols = 2, len(images)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    orig = image_dict['orig']
    sharp = sharpen_image(orig)

    axs[0, idx].set_title('"{}"'.format(filename))
    axs[0, idx].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[1, idx].set_title('sharpened'.format(filename))
    axs[1, idx].imshow(sharp, cmap='gray', vmin=0, vmax=255)

    # Save pixel values of original image's histogram as a 2D matrix in a .dat file
    np.savetxt(
```

```
        path_out_conv + ext_inp[1:] + '/' + filename + f'_sharpened' + ext_inp,
        sharp,
        fmt=' %d',
        newline=' \n'
    )

    # Save noisy image as .bmp file
    plt.imsave(
        path_out_conv + ext_out[1:] + '/' + filename + f'_sharpened' + ext_out,
        sharp,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Save and display the figure
plt.savefig('sharpened_image.jpg')
plt.show()
```



Resource

GitHub repository: Image Processing and Pattern Recognition - Anindya Kundu (meganindya)

Question 14

Write a program to calculate Δx and Δy for each pixel of an input image $f(x,y)$ using Prewitt's operator (gradient operator).

```
-1  0  1     -1 -1 -1
-1  0  1      0  0  0
-1  0  1      1  1  1
```

Replace each pixel point by $|\Delta x| + |\Delta y|$ and then implement image-sharpening operation.

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'fu256',
    'l256',
    'n256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

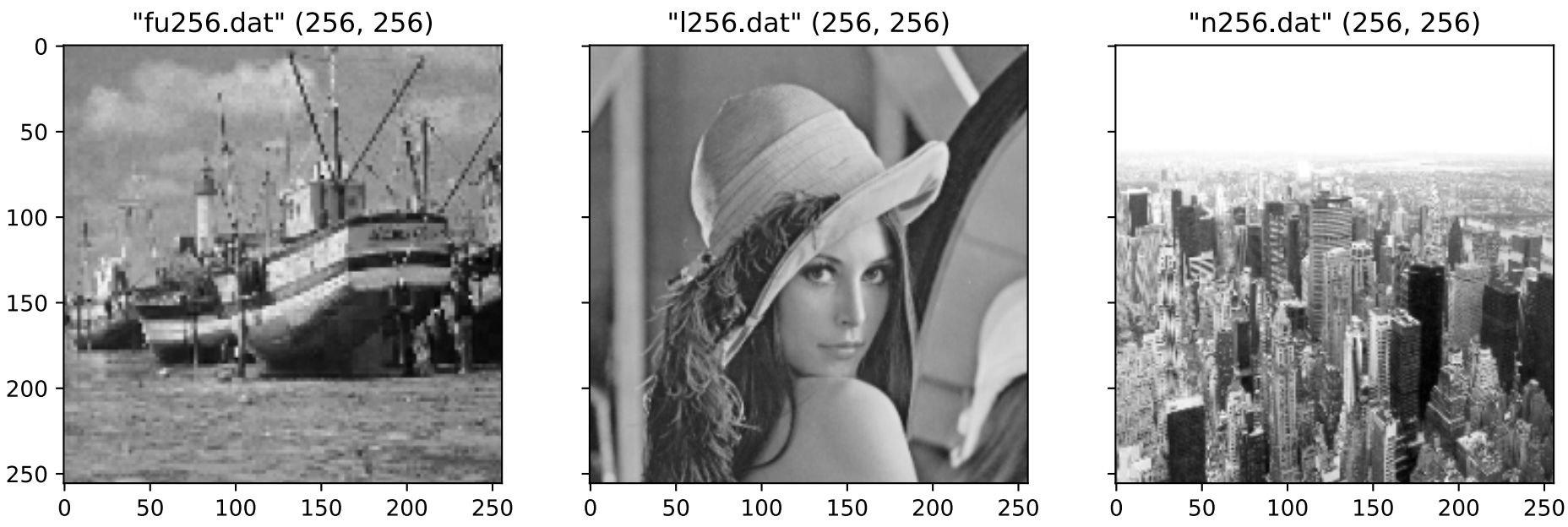
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image,
        'equalized': None
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = 1

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[idx].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[idx].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )
```

```
    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



Sharpen Images

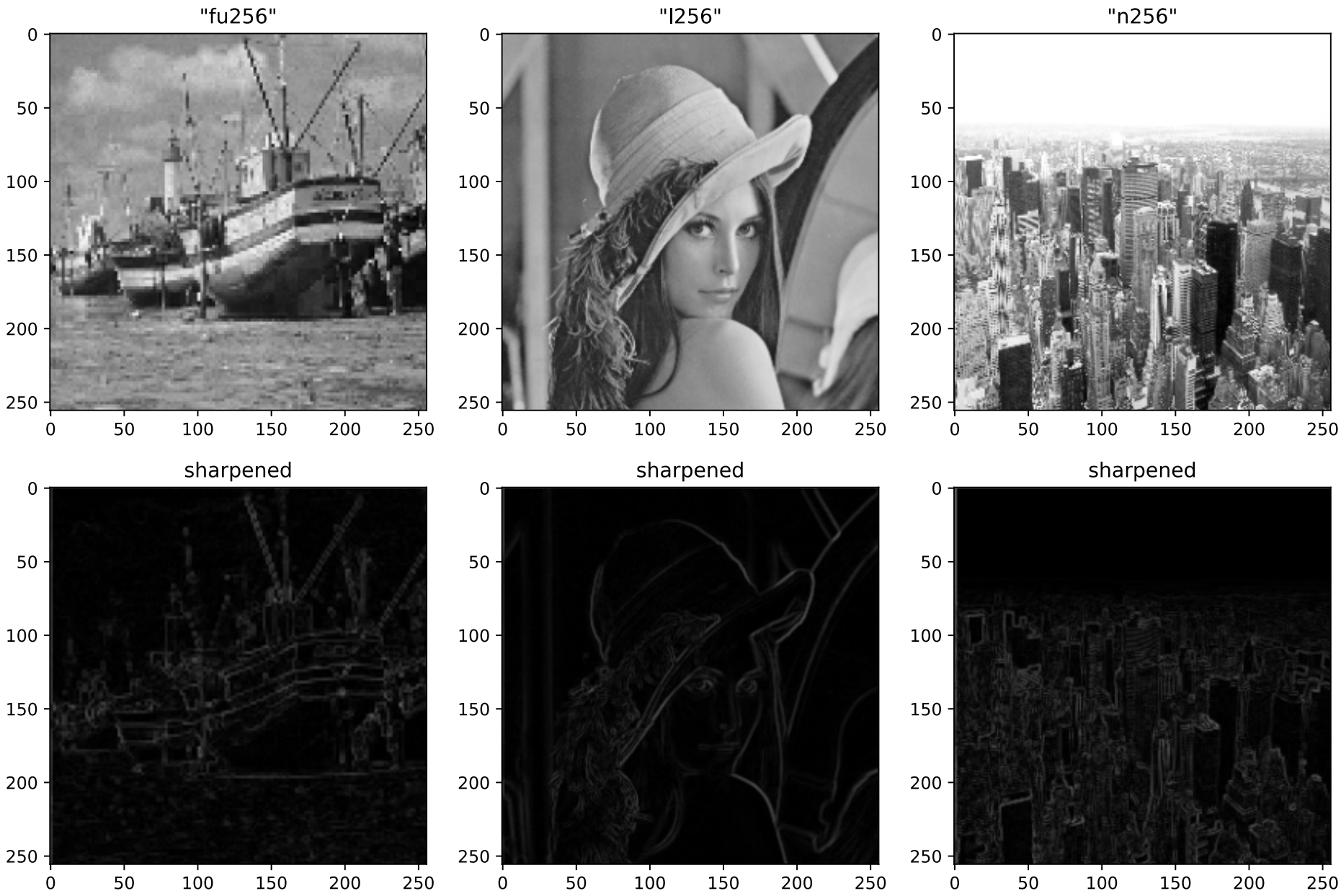
In [5]:

```
def sharpen_image(image):
    k_x = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]], dtype='float32')
    k_y = np.array([[-1, -1, -1], [0, 0, 0], [1, 1, 1]], dtype='float32')

    height, width = image.shape
    tmp = np.zeros((height + 2, width + 2))
    img = np.zeros((height, width))

    for i in range(height):
        for j in range(width):
            tmp[i + 1][j + 1] = image[i][j]

    for i in range(height):
        for j in range(width):
            s_x = 0
            for m in [-1, 0, 1]:
                for n in [-1, 0, 1]:
                    s_x += tmp[i + m][j + n] * k_x[m + 1][n + 1]
            s_x = abs(s_x // 9)

            s_y = 0
            for m in [-1, 0, 1]:
                for n in [-1, 0, 1]:
                    s_y += tmp[i + m][j + n] * k_y[m + 1][n + 1]
            s_y = abs(s_y // 9)

            img[i][j] = s_x + s_y
    img.astype('uint8')
    return img
```

In [6]:

```
rows, cols = 2, len(images)

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    orig = image_dict['orig']
    sharp = sharpen_image(orig)

    axs[0, idx].set_title('"{}"'.format(filename))
    axs[0, idx].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[1, idx].set_title('sharpened'.format(filename))
    axs[1, idx].imshow(sharp, cmap='gray', vmin=0, vmax=255)

    # Save pixel values of original image's histogram as a 2D matrix in a .dat file
    np.savetxt(
```

```
        path_out_conv + ext_inp[1:] + '/' + filename + f'_sharpened' + ext_inp,
        sharp,
        fmt=' %d',
        newline=' \n'
    )

    # Save noisy image as .bmp file
    plt.imsave(
        path_out_conv + ext_out[1:] + '/' + filename + f'_sharpened' + ext_out,
        sharp,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Save and display the figure
plt.savefig('sharpened_image.jpg')
plt.show()
```



Resource

GitHub repository: Image Processing and Pattern Recognition - Anindya Kundu (meganindya)

Question 15

Write a program to implement Laplacian operation for the input image $f(x,y)$ using the following operator.

```
 0  1  0     1  1  1
 1 -4  0     1 -8  1
 0  1  0     1  1  1
```

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'fu256',
    'l256',
    'n256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image,
        'equalized': None
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = 1

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[idx].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[idx].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save original image as .bmp file
    plt.imsave(
```

```
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```

Filter Images

In [5]:

```
def filter_image(image, kernel):
    height, width = image.shape
    tmp = np.zeros((height + 2, width + 2))
    img = np.zeros((height, width))

    for i in range(height):
        for j in range(width):
            tmp[i + 1][j + 1] = image[i][j]

    def max(a, b):
        return a if a > b else b

    def min(a, b):
        return a if a < b else b

    for i in range(height):
        for j in range(width):
            sum = 0
            for m in [-1, 0, 1]:
                for n in [-1, 0, 1]:
                    sum += tmp[i + m][j + n] * kernel[m + 1][n + 1]
            img[i][j] = max(min(sum // 9, 255), 0)
    img.astype('uint8')
    return img
```

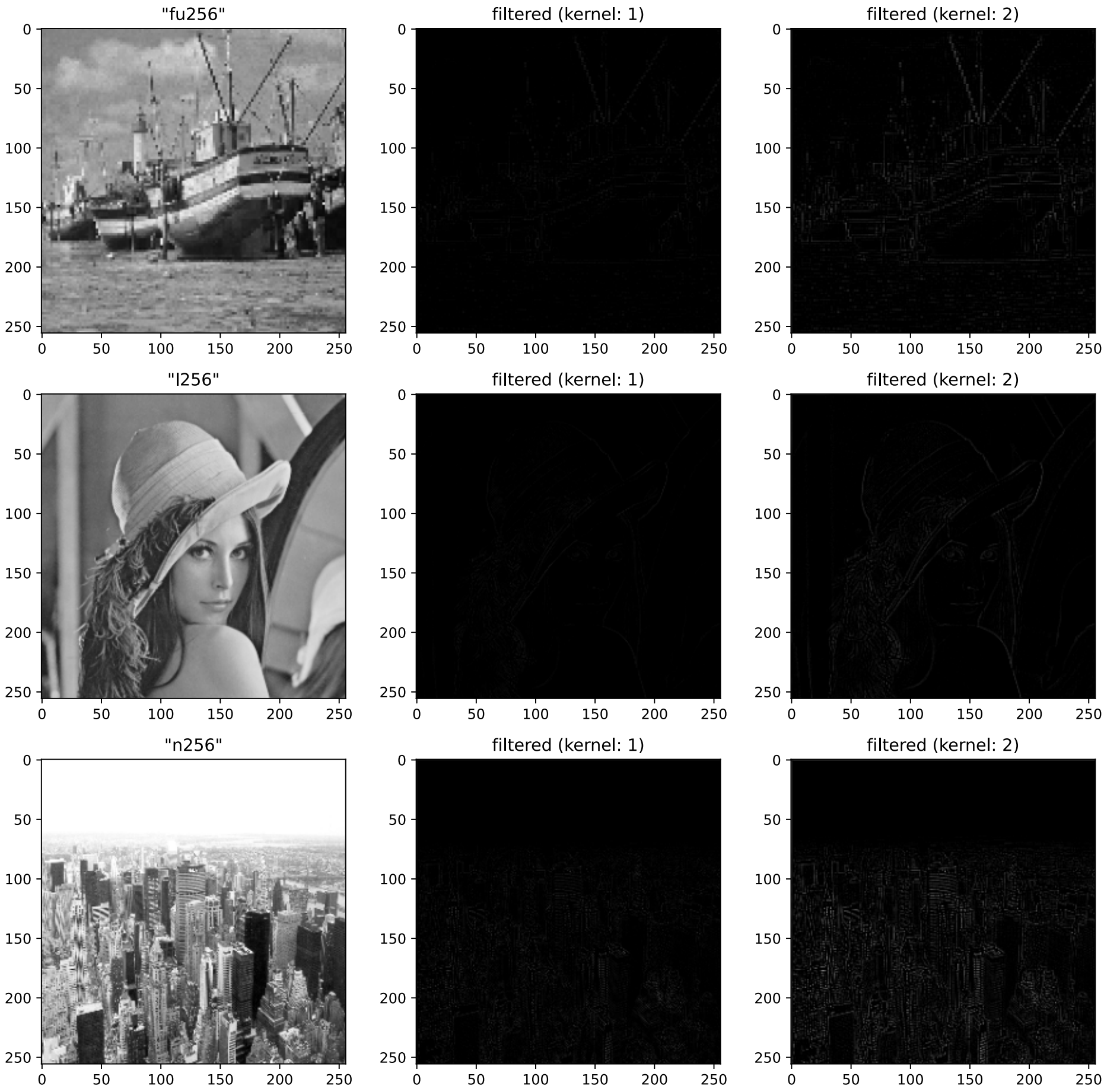
In [6]:

```
def get_kernel_1():
    return np.array([[0, 1, 0], [1, -4, 1], [0, 1, 0]], dtype='float32')

def get_kernel_2():
    return np.array([[1, 1, 1], [1, -8, 1], [1, 1, 1]], dtype='float32')
```

In [7]:

```
rows, cols = len(images), 3

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    orig = image_dict['orig']

    filtered = []
    filtered.append(filter_image(orig, get_kernel_1()))
    filtered.append(filter_image(orig, get_kernel_2()))

    axs[idx, 0].set_title('"{}"'.format(filename))
    axs[idx, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)

    for (i, item) in enumerate(filtered):
        axs[idx, i + 1].set_title(f'filtered (kernel: {i + 1})'.format(filename))
```

```
        axs[idx, i + 1].imshow(item, cmap='gray', vmin=0, vmax=255)

        # Save pixel values of original image's histogram as a 2D matrix in a .dat file
        np.savetxt(
            path_out_conv + ext_inp[1:] + '/' + filename + f'_filtered_kernel_{i + 1}' + ext_inp,
            item,
            fmt=' %d',
            newline=' \n'
        )

        # Save noisy image as .bmp file
        plt.imsave(
            path_out_conv + ext_out[1:] + '/' + filename + f'_filtered_kernel_{i + 1}' + ext_out,
            item,
            cmap='gray',
            vmin=0,
            vmax=255
        )

# Save and display the figure
plt.savefig('filtered_image.jpg')
plt.show()
```



Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 16

Write a program to implement high boost filtering using the operator as shown below. Show the effect for different `A` values.

```
 0  1  0    -1   -1  -1
 1 A-4 0    -1  A+8  -1
 0  1  0    -1   -1  -1
```

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'f256',
    'l256',
    'o256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

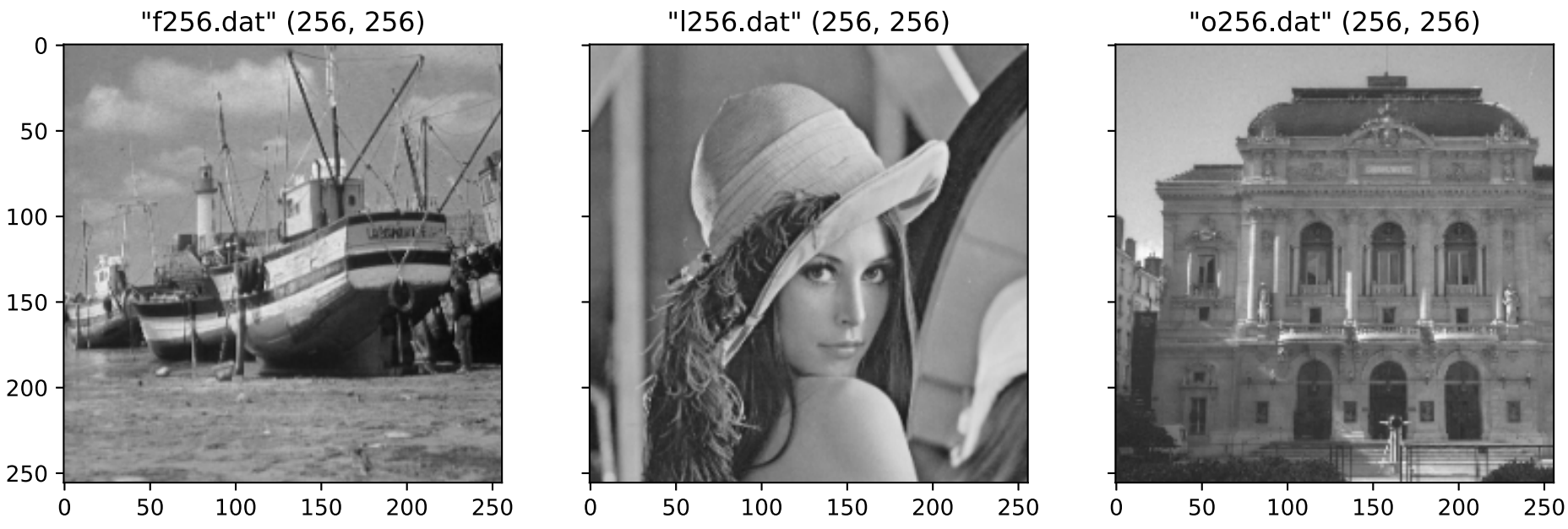
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image,
        'equalized': None
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = 1

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[idx].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[idx].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save original image as .bmp file
    plt.imsave(
```

```
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



Filter Images

In [5]:
```
def filter_image(image, kernel):
    height, width = image.shape
    tmp = np.zeros((height + 2, width + 2))
    img = np.zeros((height, width))

    for i in range(height):
        for j in range(width):
            tmp[i + 1][j + 1] = image[i][j]

    def min(a, b):
        return a if a < b else b

    for i in range(height):
        for j in range(width):
            sum = 0
            for m in [-1, 0, 1]:
                for n in [-1, 0, 1]:
                    sum += tmp[i + m][j + n] * kernel[m + 1][n + 1]
            img[i][j] = min(sum // 9, 255)
    img.astype('uint8')
    return img
```

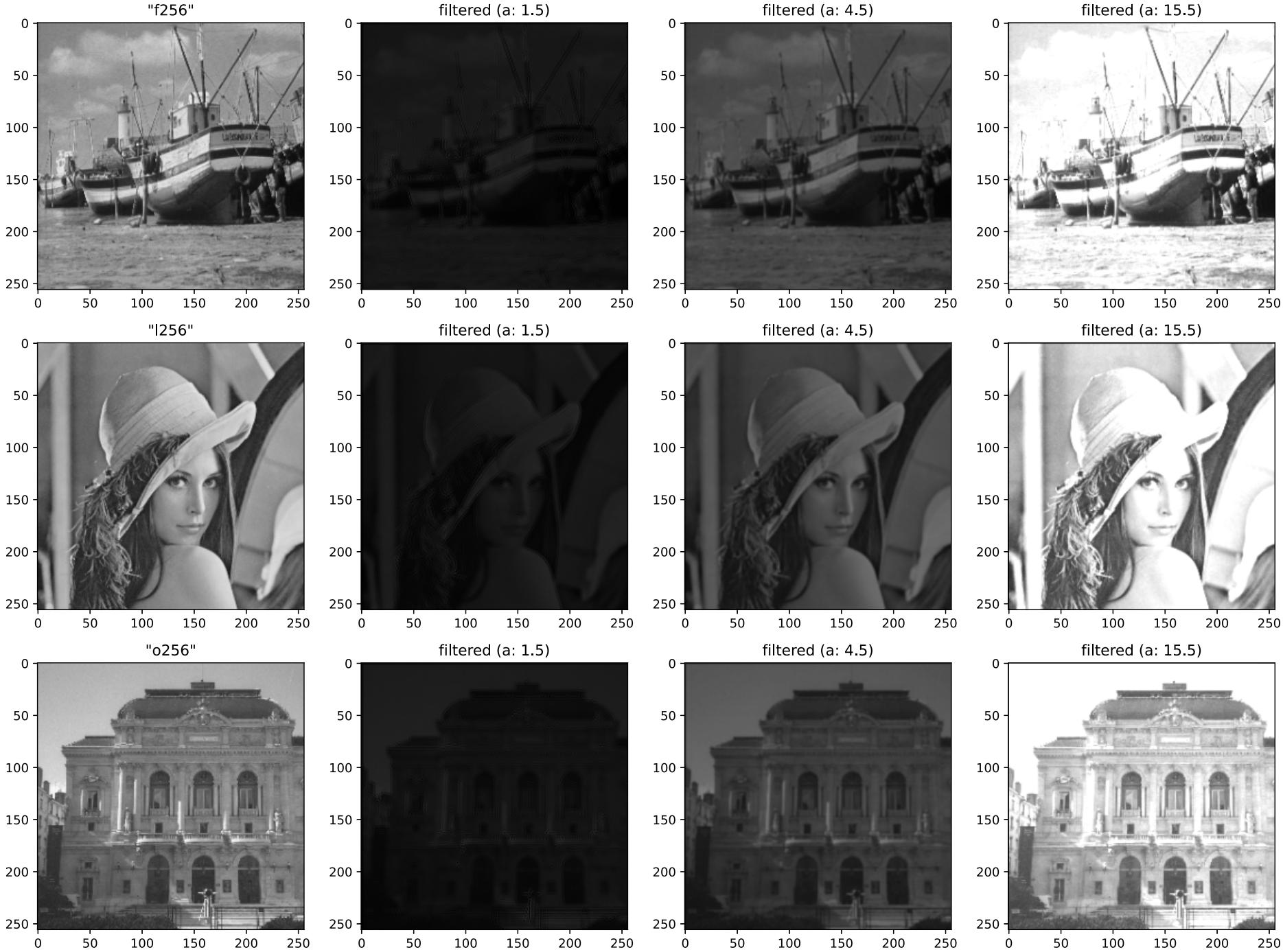
In [6]:
```
def get_kernel_1(a: float):
    return np.array([[0, 1, 0], [1, (a - 4), 1], [0, 1, 0]], dtype='float32')

def get_kernel_2(a: float):
    return np.array([[-1, -1, -1], [-1, a + 8, -1], [-1, -1, -1]], dtype='float32')
```

Kernel 1

```
  0  1  0
  1 A-4 0
  0  1  0
```

In [7]:
```
rows, cols = len(images), 4

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    orig = image_dict['orig']

    a = [1.5, 4.5, 15.5]
    filtered = list(map(lambda x: filter_image(orig, get_kernel_1(x)), a))

    axs[idx, 0].set_title('"{}"'.format(filename))
    axs[idx, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)
```

```
    for (i, item) in enumerate(filtered):
        axs[idx, i + 1].set_title(f'filtered (a: {a[i]})'.format(filename))
        axs[idx, i + 1].imshow(item, cmap='gray', vmin=0, vmax=255)

        # Save pixel values of original image's histogram as a 2D matrix in a .dat file
        np.savetxt(
            path_out_conv + ext_inp[1:] + '/' + filename + f'_filtered_kernel_1_{a[i]}' + ext_inp,
            item,
            fmt=' %d',
            newline=' \n'
        )

        # Save noisy image as .bmp file
        plt.imsave(
            path_out_conv + ext_out[1:] + '/' + filename + f'_filtered_kernel_1_{a[i]}' + ext_out,
            item,
            cmap='gray',
            vmin=0,
            vmax=255
        )

# Save and display the figure
plt.savefig('filtered_image_kernel_1.jpg')
plt.show()
```



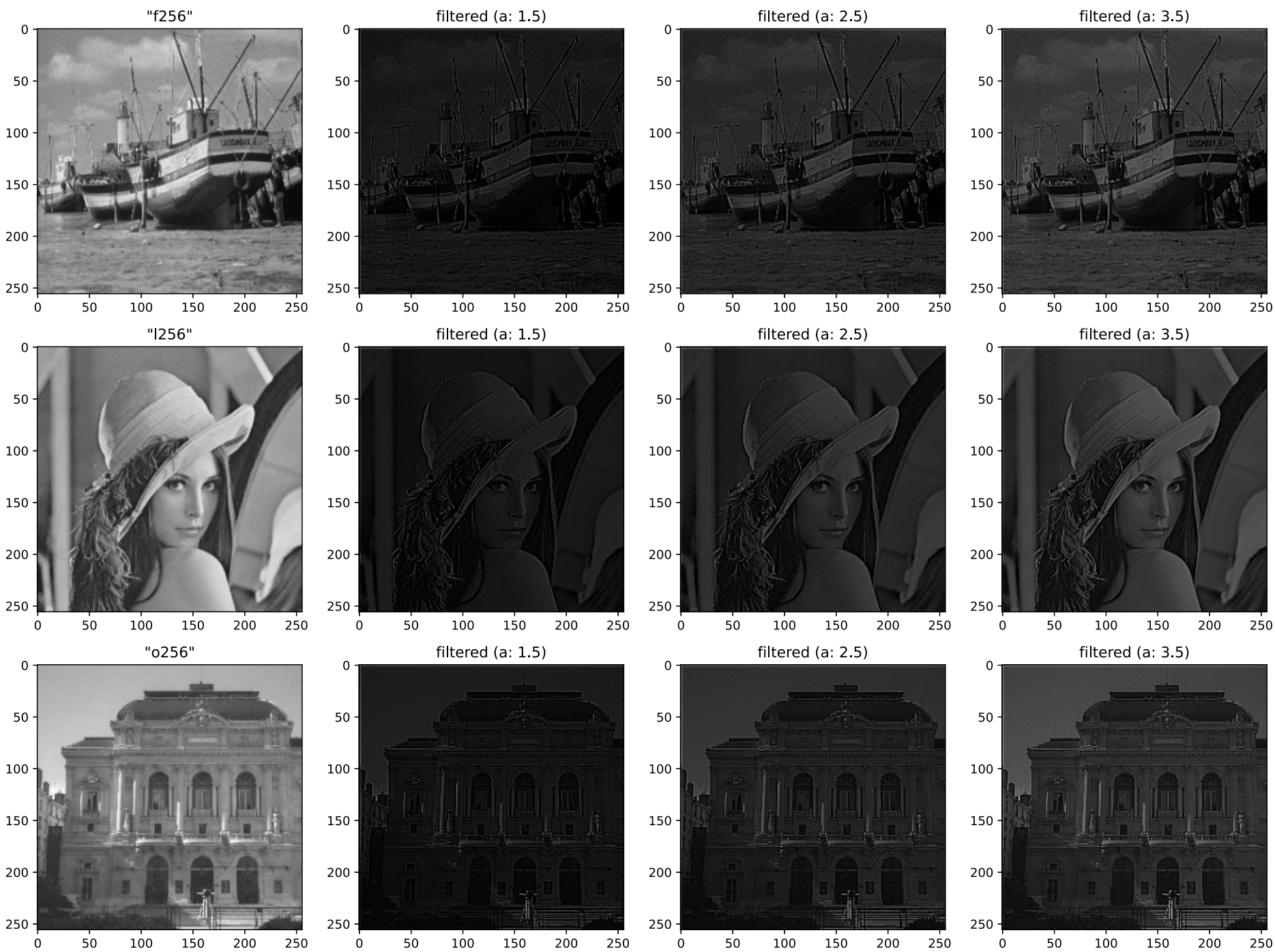
Kernel 2

$$\begin{matrix} -1 & -1 & -1 \\ -1 & A+8 & -1 \\ -1 & -1 & -1 \end{matrix}$$

In [8]:

```
rows, cols = len(images), 4

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    orig = image_dict['orig']

    a = [1.5, 2.5, 3.5]
    filtered = list(map(lambda x: filter_image(orig, get_kernel_2(x)), a))
```

```
    axs[idx, 0].set_title('"{}"'.format(filename))
    axs[idx, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)

    for (i, item) in enumerate(filtered):
        axs[idx, i + 1].set_title(f'filtered (a: {a[i]})'.format(filename))
        axs[idx, i + 1].imshow(item, cmap='gray', vmin=0, vmax=255)

        # Save pixel values of original image's histogram as a 2D matrix in a .dat file
        np.savetxt(
            path_out_conv + ext_inp[1:] + '/' + filename + f'_filtered_kernel_2_{a[i]}' + ext_inp,
            item,
            fmt=' %d',
            newline=' \n'
        )

        # Save noisy image as .bmp file
        plt.imsave(
            path_out_conv + ext_out[1:] + '/' + filename + f'_filtered_kernel_2_{a[i]}' + ext_out,
            item,
            cmap='gray',
            vmin=0,
            vmax=255
        )

# Save and display the figure
plt.savefig('filtered_image_kernel_2.jpg')
plt.show()
```



Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 17

Write a program to implement 2D transformation by an amount of 5 units in the right and 7 units upward to each pixel of a given input image *f(x, y)* .

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
from skimage.transform import rescale
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'f256',
    'l256',
    'o256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

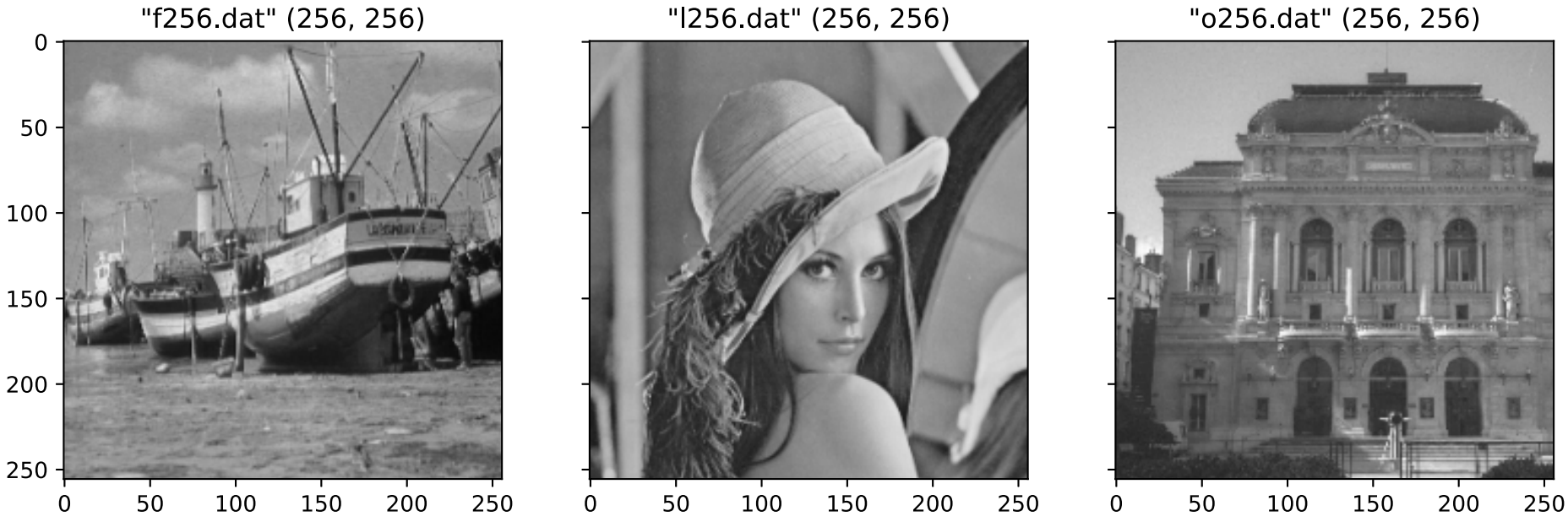
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image,
        'equalized': None
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = 1

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[idx].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[idx].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
```

```
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```

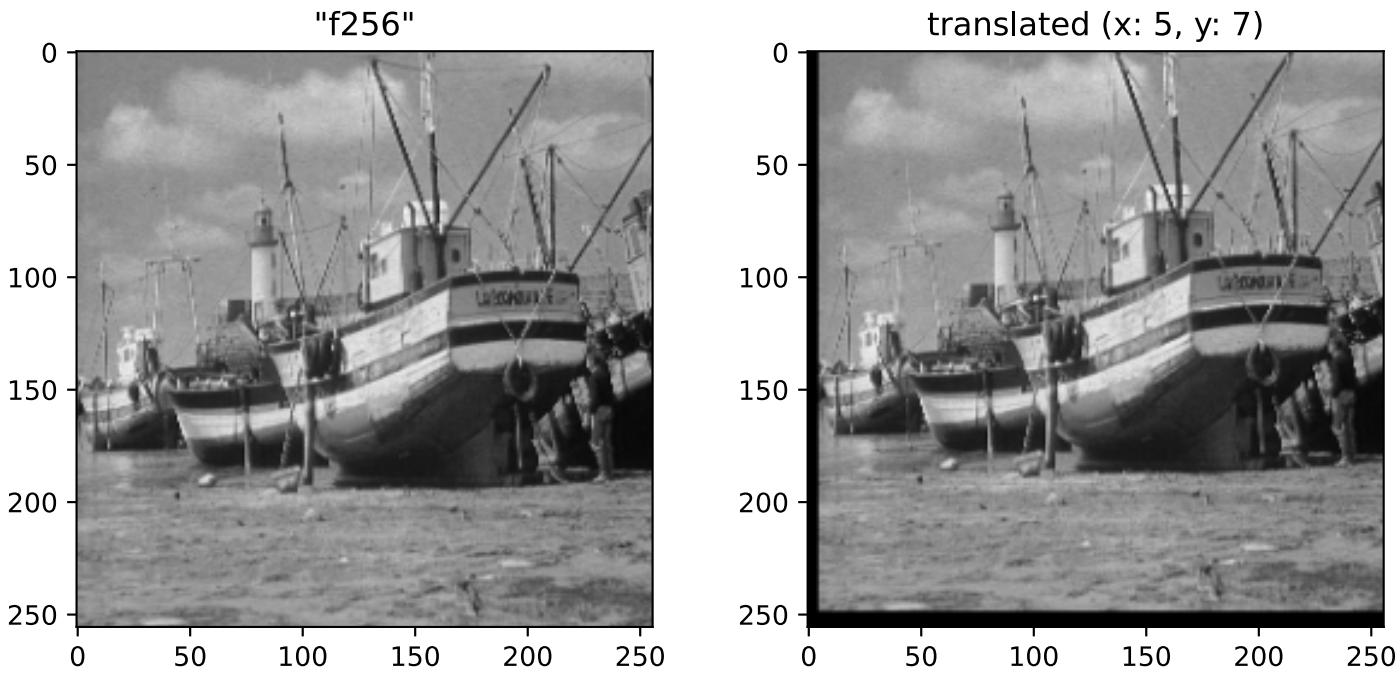


Translate Images

In [5]:
```
def translate_image(image, x:int, y: int):
    height, width = image.shape
    img = np.zeros(image.shape)

    def max(a, b):
        return a if a > b else b

    def min(a, b):
        return a if a < b else a

    for i in range(height):
        for j in range(width):
            img[i][j] = image[i + y][j - x] if (i + y) < 256 and (j - x) > -1 else 0
    img.astype('uint8')

    return img
```

In [6]:
```
rows, cols = len(images), 2

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    FACTOR_X, FACTOR_Y = 5, 7
    orig = image_dict['orig']
    translated = translate_image(orig, FACTOR_X, FACTOR_Y)

    axs[idx, 0].set_title('"{}"'.format(filename))
    axs[idx, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[idx, 1].set_title(f'translated (x: {FACTOR_X}, y: {FACTOR_Y})'.format(filename))
    axs[idx, 1].imshow(translated, cmap='gray', vmin=0, vmax=255)

    # Save pixel values of original image's histogram as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_translated' + ext_inp,
        translated,
        fmt=' %d',
        newline=' \n'
    )

    # Save noisy image as .bmp file
    plt.imsave(
        path_out_conv + ext_out[1:] + '/' + filename + '_translated' + ext_out,
        translated,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Save and display the figure
```

```
plt.savefig('translate_image.jpg')
plt.show()
```



Resource

GitHub repository: Image Processing and Pattern Recognition - Anindya Kundu (meganindya)

Question 18

Write a program to implement image scaling operation in horizontal direction by an amount `14.2` and vertical direction by an amount `1.6` .

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
from skimage.transform import rescale
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'f256',
    'l256',
    'o256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

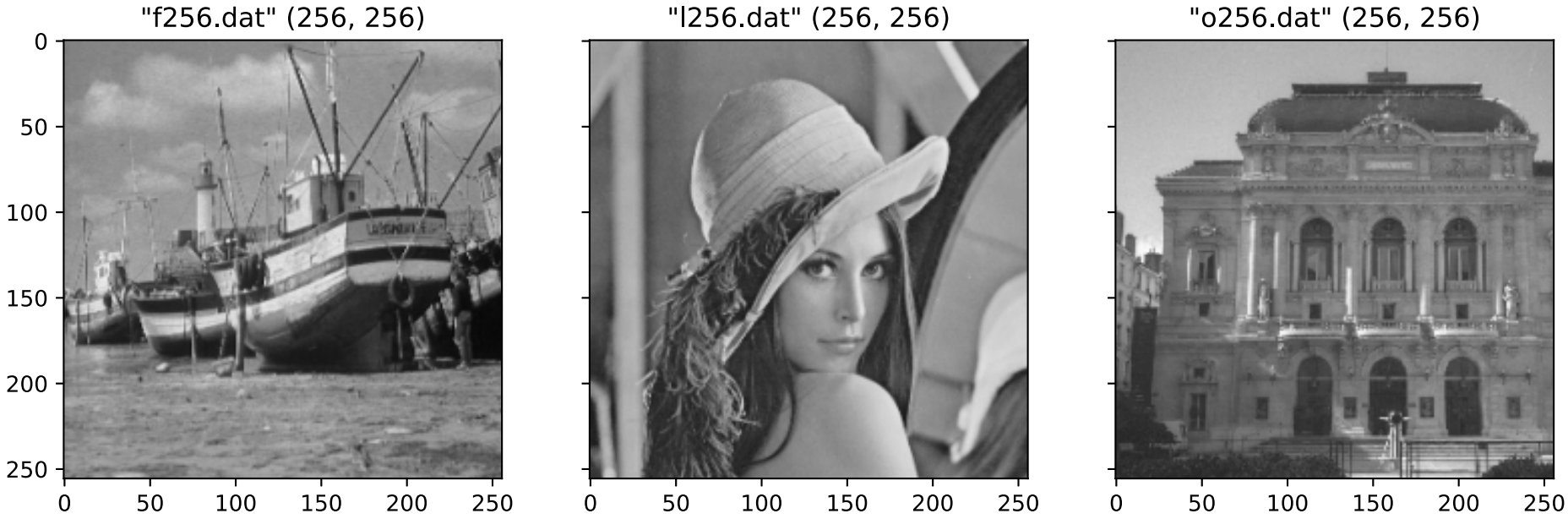
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image,
        'equalized': None
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = 1

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[idx].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[idx].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
```

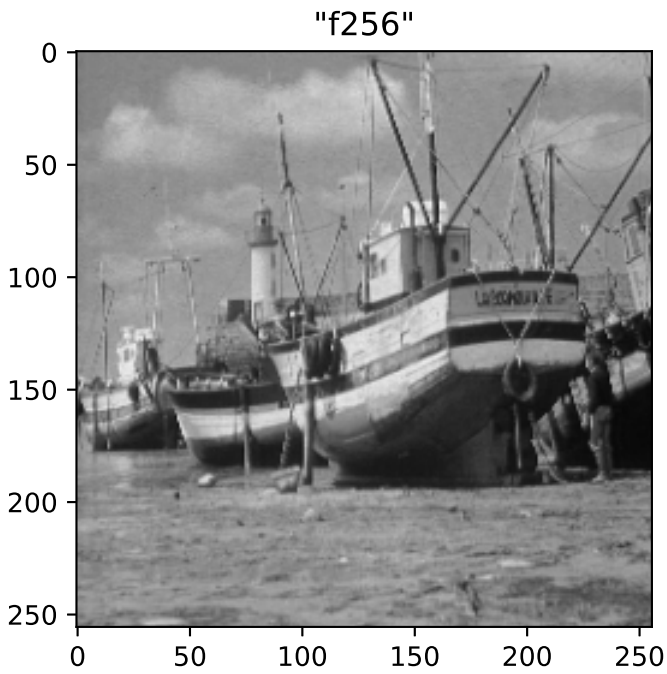
```
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```

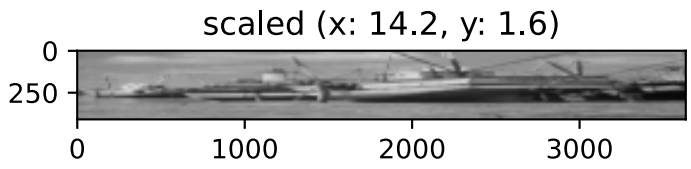


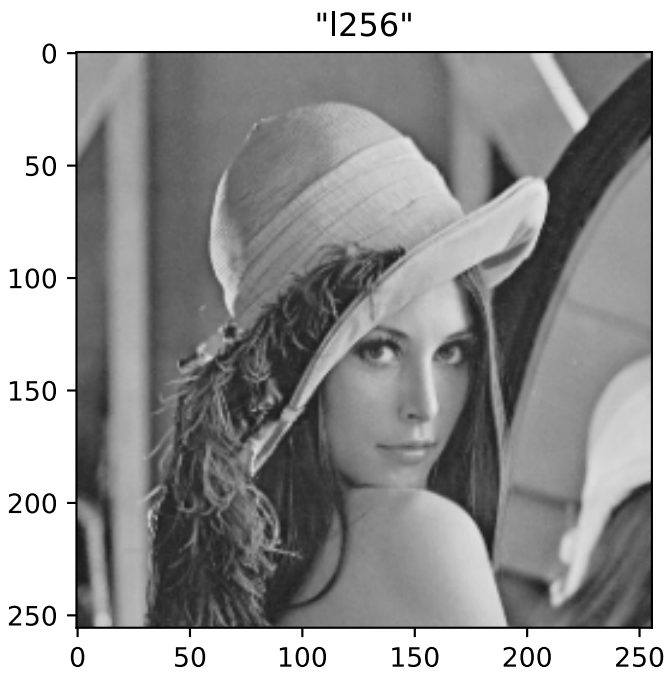
Scale Images

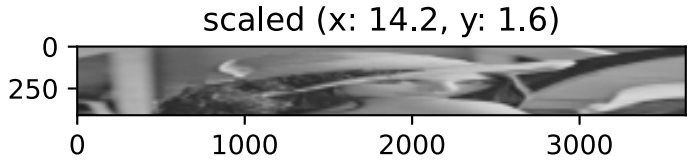
In [5]:
```
def scale_image(image, scale_x: int, scale_y: int):
    return rescale(image, (scale_y, scale_x)) * 255
```

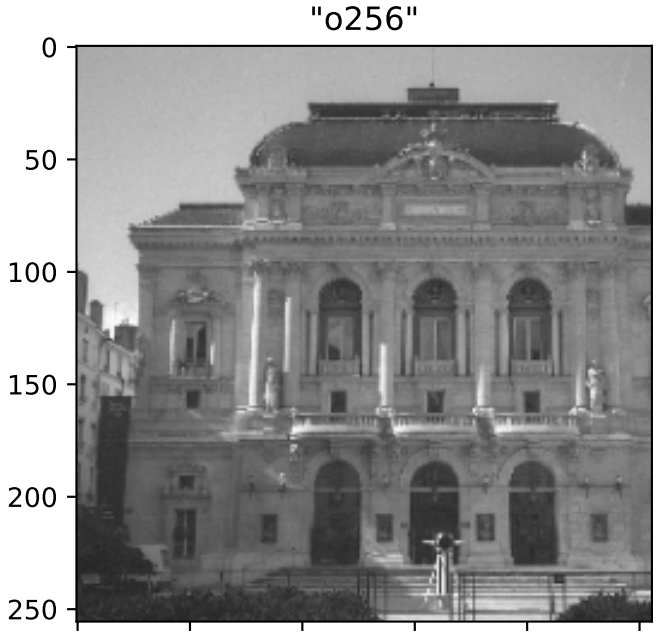
In [6]:
```
rows, cols = len(images), 2

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    SCALE_X, SCALE_Y = 14.2, 1.6
    orig = image_dict['orig']
    scaled = scale_image(orig, SCALE_X, SCALE_Y)

    axs[idx, 0].set_title('"{}"'.format(filename))
    axs[idx, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)

    axs[idx, 1].set_title(f'scaled (x: {SCALE_X}, y: {SCALE_Y})'.format(filename))
    axs[idx, 1].imshow(scaled, cmap='gray', vmin=0, vmax=255)

    # Save pixel values of original image's histogram as a 2D matrix in a .dat file
    np.savetxt(
        path_out_conv + ext_inp[1:] + '/' + filename + '_scaled' + ext_inp,
        scaled,
        fmt=' %d',
        newline=' \n'
    )

    # Save noisy image as .bmp file
    plt.imsave(
        path_out_conv + ext_out[1:] + '/' + filename + '_scaled' + ext_out,
        scaled,
        cmap='gray',
        vmin=0,
        vmax=255
    )

# Save and display the figure
plt.savefig('scale_image.jpg')
plt.show()
```

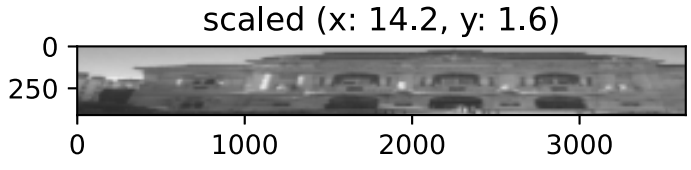












Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`

Question 19

Write a program to implement image rotation operation by amount of a) 25° b) 45° c) 60°

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import random
from skimage.transform import rotate
```

Images to process

In [2]:
```
path_inp = '../../images/dat/'  # path for input files
path_out_orig = 'originals/'    # path for output files: originals
path_out_conv = 'converted/'    # path for output files: converted

filenames = [
    'f256',
    'l256',
    'o256'
]

ext_inp = '.dat'    # file extention for input
ext_out = '.bmp'    # file extention for output
```

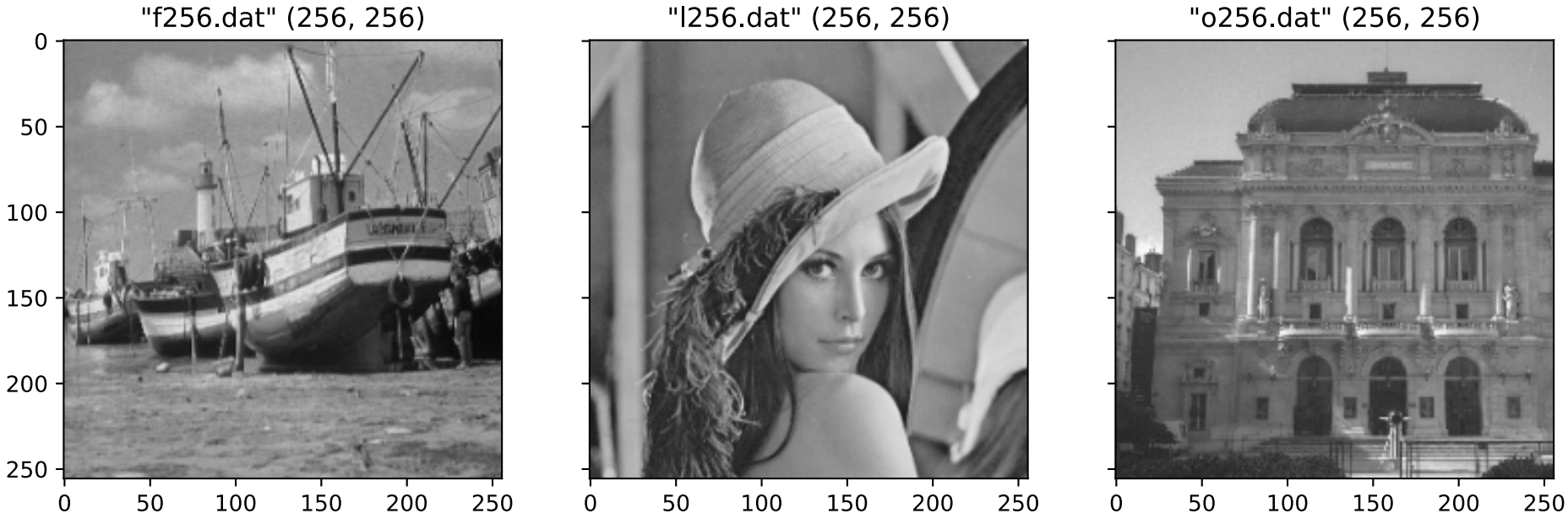
Convert images to numpy array and store in a list of tuples as (filename, np.array)

In [3]:
```
# Stores the list of dictionaries for the filename, original image, converted image/s
images = []

# Iterate for all filenames
for idx, filename in enumerate(filenames):
    # Store image pixels as uint8 2D array
    image = np.array(
        [i.strip().split() for i in open(path_inp + filename + ext_inp).readlines()],
        dtype='uint8'
    )

    # Add (filename, numpy array of image) into images list
    images.append({
        'filename': filename,
        'orig': image,
        'equalized': None
    })

    # Save original image as .dat file
    np.savetxt(
        path_out_orig + ext_inp[1:] + '/' + filename + ext_inp,
        image,
        fmt=' %d',
        newline=' \n'
    )
```

Display input images

In [4]:
```
# Matrix dimensions
cols = 3
rows = 1

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80, sharex=True, sharey=True)
fig.set_size_inches(4 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']
    image = image_dict['orig']

    # Set subplot title as '"filename" (rows, cols)'
    axs[idx].set_title('"{}" {}'.format(
        filename + ext_inp,
        image.shape
    ))
    # Add subplot to figure plot buffer
    axs[idx].imshow(
        image,
        cmap='gray',
        vmin=0,
        vmax=255
    )

    # Save original image as .bmp file
    plt.imsave(
        path_out_orig + ext_out[1:] + '/' + filename + ext_out,
        image,
        cmap='gray',
```

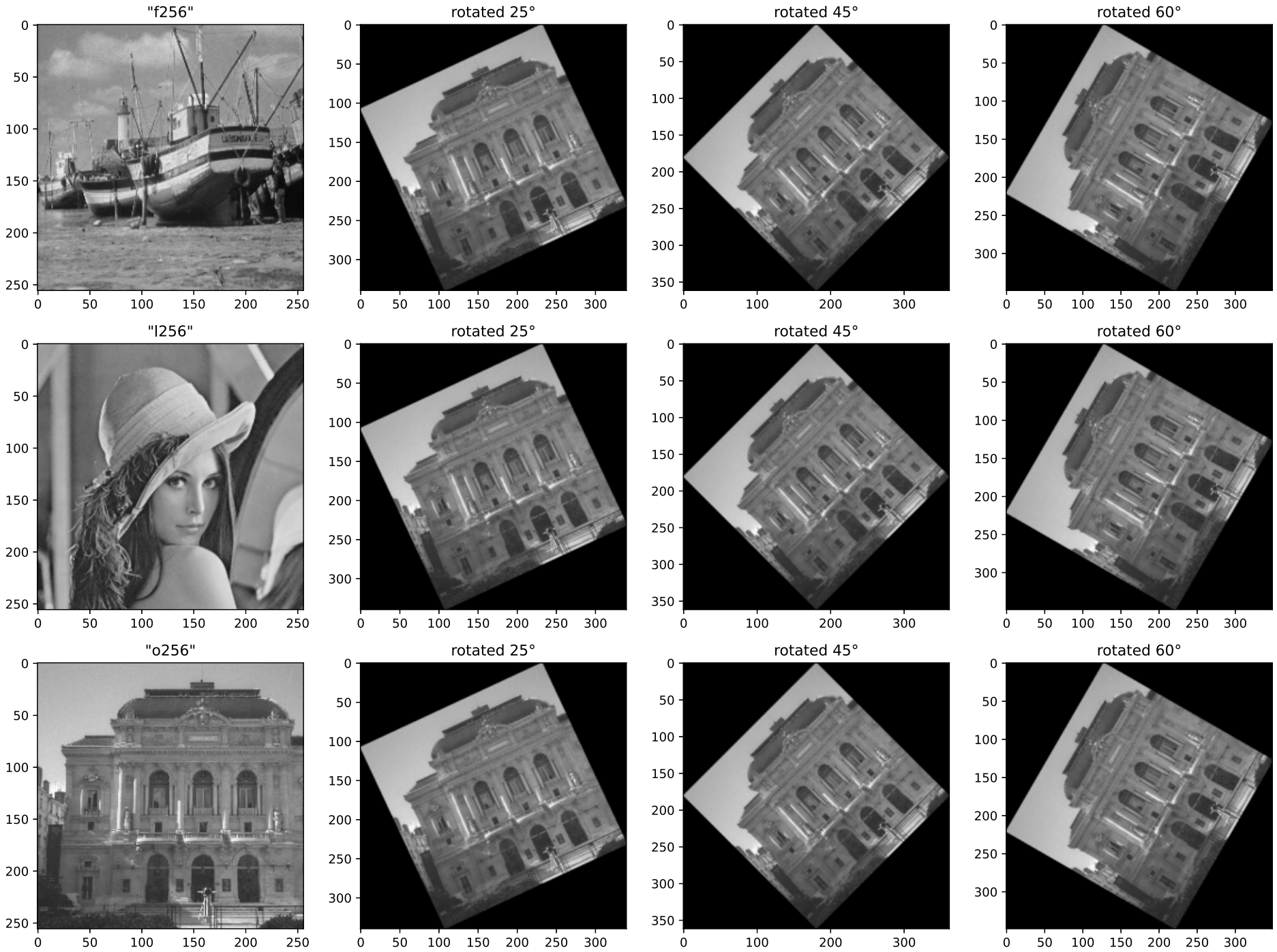
```
        vmin=0,
        vmax=255
    )

# Hide x labels and tick labels for top plots and y ticks for right plots
for ax in axs.flat:
    ax.label_outer()

# Display the figure
plt.show()
```



Rotate Image

In [5]:
```
def rotate_image(image, degrees: int):
    return rotate(image, degrees, resize=True) * 255
```

In [6]:
```
rows, cols = len(images), 4

# Create figure with rows × cols subplots
fig, axs = plt.subplots(rows, cols, dpi=80)
fig.set_size_inches(4.5 * cols, 4.5 * rows)

# Iterate for all images
for idx, image_dict in enumerate(images):
    filename = image_dict['filename']

    angles = [25, 45, 60]
    orig = image_dict['orig']
    rotated = list(map(lambda x: rotate_image(image, x), angles))

    axs[idx, 0].set_title('"{}"'.format(filename))
    axs[idx, 0].imshow(orig, cmap='gray', vmin=0, vmax=255)

    for i, angle in enumerate(angles):
        axs[idx, i + 1].set_title(f'rotated {angle}°'.format(filename))
        axs[idx, i + 1].imshow(rotated[i], cmap='gray', vmin=0, vmax=255)

        # Save pixel values of original image's histogram as a 2D matrix in a .dat file
        np.savetxt(
            path_out_conv + ext_inp[1:] + '/' + filename + f'_rotated_{angle}' + ext_inp,
            rotated[i],
            fmt=' %d',
            newline=' \n'
        )

        # Save noisy image as .bmp file
        plt.imsave(
            path_out_conv + ext_out[1:] + '/' + filename + f'_rotated_{angle}' + ext_out,
            rotated[i],
            cmap='gray',
            vmin=0,
            vmax=255
        )

# Save and display the figure
plt.savefig('rotate_image.jpg')
plt.show()
```



Resource

GitHub repository: `Image Processing and Pattern Recognition - Anindya Kundu (meganindya)`