The efficient coding principle

Information theory

A bit is the most basic unit of information (I), or entropy (H). For example, given a probability distribution $P(a)$ over the allowed values of a, the amount of information needed to get the exact value of a is

$$H(a) = I(a) = \sum_a P(a)\{-\log_2 P(a)\} \text{ bits} = -\sum_a P(a)\log_2 P(a) \text{ bits}$$

Entropy H is a measure of uncertainty, or the amount of information missing before one knows the exact value of the variable. It is the highest when all possibilities are equally likely (the variable is uniformly distributed).

When a signal is transmitted through a noisy channel, the transmission process can be described by 3 components: (1) signal S, (2) noise N, and (3) output $O = S + N$. In a noisy channel, the output is unlikely to be equal to the signal itself. Rather, by observing the output, we learn additional information about the signal. This information is called the mutual information:

$$I(O,S) = I(S,O) = H(S) + H(O) - H(O,S) = H(S) - H(S|O) = H(O) - H(O|S)$$

In a Gaussian channel (S, N, and O are Gaussian random variables), larger standard deviation leads to higher entropy. Moreover, the mutual information between S and O depends on the signal-to-noise ratio σ_s^2/σ_n^2 :

$$I(O,S) = \log_2 \frac{\sigma_O}{\sigma_n} = \frac{1}{2}\log_2\left(1 + \frac{\sigma_s^2}{\sigma_n^2}\right)$$

A non-zero mutual information means some information is shared between the variables. Thus, they are redundant:

$$H(O) + H(S) \geq H(O,S)$$

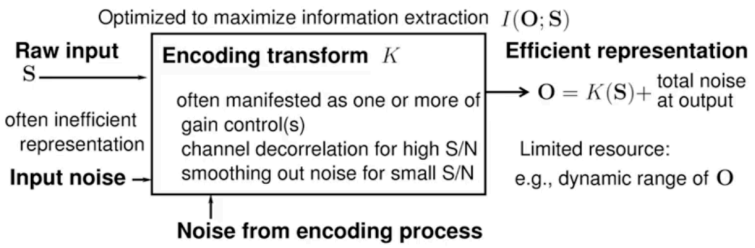
$$\text{Redundancy} = \frac{H(O)+H(S)}{H(O,S)} - 1 \geq 0$$

Redundant information requires more coding resources than non-redundant information. However, redundant information is not inherently bad, as it helps in error correction.

Formulation of the efficient coding principle

The primary bottleneck in the visual system is the optic nerve. The efficient coding principle proposes a hypothesis on how the information from the retina may be condensed and transmitted to V1 by maintaining as much visual information as possible while minimizing the neural connections. Mathematically, the goal of the efficient coding principle is to find the optimal K to minimize the neural cost required to transmit the information:

$$E(K) \equiv \text{neural cost} - \lambda I(O,S)$$



When the input noise is negligible, the transformation decorrelates the input channels to reduce information redundancy. When the noise is high, however, the input redundancy is maintained for error correction and the noise is smoothed.

Application on input sampling in contrast, color, and space

A few examples of phenomena explained by the efficient coding principle:

1. Contrast sampling in a fly's compound eye: the highest sensitivity to most probable inputs
2. Spatial sampling by photoreceptor distribution on the retina: cone density matches the distribution density of the objects
3. Optimal color sampling by the cones on the light spectrum

Formulation and solution of efficient coding by linear neural receptive fields

The transform K describes the properties of the neuron's receptive field, e.g., spatial or temporal receptive field, color tuning, ocular dominance properties, etc. The optimal encoding transform K that minimizes $E(K)$ is found by solving $\delta E/\delta K = 0$. The solution factors into:

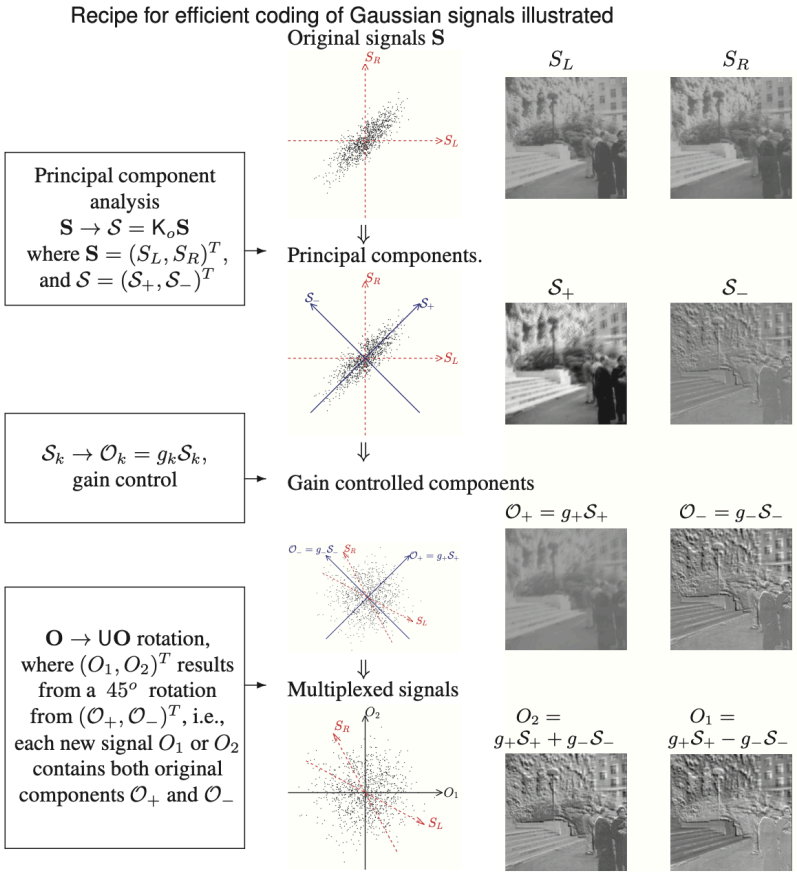
$$K = UgK_0$$

where K_0 – principal component decomposition, g – gain control according to signal-to-noise ratio (SNR) of each component, and U – multiplexing. In high SNR conditions, the gain amplifies the weaker signal to achieve decorrelation and redundancy reduction; in low SNR, the gain drops to smooth out the noise.

Case study: efficient coding for stereo vision

A neuron in V1 receives a weighted sum of left and right eye inputs. The preference towards one eye over the other can be observed in the ocular dominance columns.
Let's examine how the efficient coding principle can be applied for efficient coding in stereo vision:

1) The principle component decomposition transforms the inputs from both eyes into decorrelated signals of (1) ocular summation and (2) ocular opponency.
2) The gain control reshapes the probability distributions of the decorrelated signals (1) and (2) so that their variances are equal.
    a) When SNR is high, ocular contrast is enhanced (whitening).
    b) When SNR is low, ocular contrast is abandoned and the summation channel is enhanced (smoothing).
3) A special case of multiplexing, $U = K_0^{-1}$, is applied. This matrix gives the lowest distortion between the inputs and outputs, and minimizes required neural wiring.



The ocularity of V1 neurons depends on the input statistics, for example:

- In bright conditions, more neurons will be binocular than in a dim environment
- In animals with long interocular distances, there will be fewer binocular cells
- More binocular cells are tuned to horizontal orientation than to vertical orientation

Efficient coding by the receptive fields of the retinal ganglion cells

The efficient coding principle can be applied to 3 feature dimensions, space, time, and color:

(1) **Retinal spatial coding**. Input is a vector with as many components as there are spatial locations.

1. Decorrelation – Fourier transform of the spatial image.
2. Gain control gives a specific weight to each Fourier component determined by its signal power.
3. Multiplexing – inverse Fourier transform.

The shape of the receptive field has the center-surround receptive field, and is the same across all neurons with the center of the receptive field translated between the neurons. When the noise is high, the receptive field size is larger and the correlations between neighboring ganglion cells will be observed more easily.

(2) **Retinal temporal coding**. Input is a signal in time. The efficient coding principle algorithm is analogous to spatial coding.

(3) **Retinal color coding**. The algorithm is similar to the ocular coding. Performing it with three color dimensions results in 3 decorrelated channels: luminance, yellow-blue and red-green.

Coding in V1, the primary visual cortex

To apply the efficient coding principle to V1 neurons, two additional goals must be fulfilled: (1) translational invariance, and (2) scale invariance. This can be achieved by using a different K transform. However, the representation in V1 is overcomplete and redundant, which goes against the efficient coding principle as the redundant representation requires high neuronal cost.

V1 neurons can be tuned to many features, such as orientation, color, spatial scale or frequency, disparity, eye of origin, motion direction, temporal frequencies, etc. However, no single cell is tuned to all the feature dimensions at once. Considering the contrast between features requires high enough signal power. This can be achieved by integrating along some feature dimensions, while taking the contrast of other feature dimensions. That is the reason why a very low number of V1 neurons are tuned to more than two features at once.

Other formulations, and unsupervised learning of the efficient codes

Alternative formulations of the efficient coding principle have been proposed, such as sparse coding, maximum entropy coding, and unsupervised learning approaches. However, these alternatives do not explain the overrepresentation in V1 and lack predictive power. Despite its limitations and because of its predictive power, the efficient coding principle remains the dominating theory in the field.

Questions

1. What is the main bottleneck in the visual system?
    a. V1
    b. V2
    c. Connections from the thalamus to V1
    d. Optic nerve

    Explanation: The optic nerve is the main bottleneck in the visual system because it has a limited capacity to transmit the vast amount of visual information from the retina to the brain, requiring significant data compression and processing.

2. Why is gain control an essential part of the efficient coding principle?
    a. It increases the speed of neural responses.
    b. It adjusts sensitivity to optimize information transmission.
    c. It ensures all sensory neurons respond equally.
    d. It reduces the energy consumption of the brain.

    Explanation: Gain control is essential in the efficient coding principle because it dynamically adjusts the sensitivity of sensory neurons to match the statistical properties of the input signal, thereby optimizing information transmission and preventing saturation or loss of information.

3. Which additional goals should be considered in order to extend the efficient coding principle to be applicable in V1?
    a. Temporal invariance.
    b. Translational invariance.
    c. Scale invariance.
    d. Ocular invariance.

    Explanation: To apply the efficient coding principle to V1 neurons, two additional goals must be fulfilled: (1) translational invariance, and (2) scale invariance. This can be achieved by using a different K transform.

4. Why does V1 have few cells tuned to more than two feature dimensions?
    a. Such tuning is not necessary for V1 to fulfill its function.
    b. Integrating over features is required to obtain high enough signal power.
    c. The complexity of neural processing decreases with fewer dimensions.

    Explanation: Considering the contrast between features requires high enough signal power. This can be achieved by integrating along some feature dimensions, while taking the contrast of other feature dimensions. That is the reason why a very low number of V1 neurons are tuned to more than two features at once.

5. The ocularity of V1 neurons depends on the input statistics, for example:
    a. **In bright conditions, more neurons will be binocular than in a dim environment.**
    b. **In animals with long interocular distances, there will be fewer binocular cells**
    c. **More binocular cells are tuned to horizontal orientation than to vertical orientation**

    Explanation: All of the answers are correct.

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

Homework 2 (4)

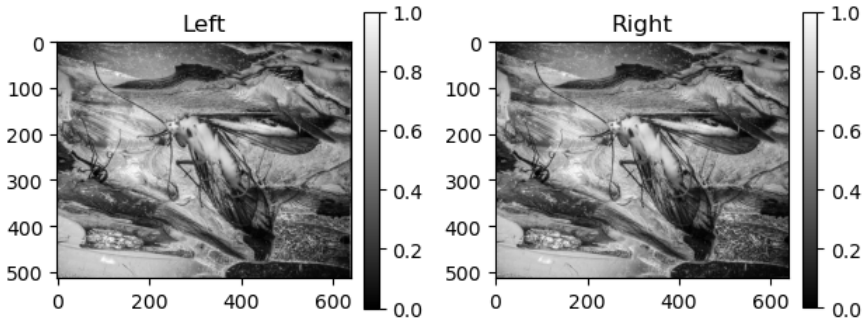
(A) Selecting a pair of stereo images

Take a pair of stereo images, one image, denoted as $S_L(x, y)$, is for the left eye, and one $S_R(x, y)$ for the right eye. Here x and y are the coordinates of the image pixel locations. For example $x = 1, 2, \ldots, 256$ and $y = 1, 2, \ldots, 256$. You may find some examples online, for example at this website https://www.londonstereo.com/3-D-gallery1.html. If the original images are colored, remove the color. Plot out the images, each as a luminance image.

In [2]:
```
left_original = imread("resources/left.png")
right_original = imread("resources/right.png")
```

In [3]:
```
def plot_left_and_right(left, right, apply_lim = False, title_left = "Left", title_right = "Right"):
    max_lim = np.max(np.abs([left, right]))

    plt.subplot(1, 2, 1)
    if apply_lim:
        plt.imshow(left, cmap = "gray", vmin = -max_lim, vmax = max_lim)
    else:
        plt.imshow(left, cmap = "gray")
    plt.colorbar(fraction = 0.046, pad = 0.04)

    if title_left != "":
        plt.title(title_left)

    plt.subplot(1, 2, 2)
    if apply_lim:
        plt.imshow(right, cmap = "gray", vmin = -max_lim, vmax = max_lim)
    else:
        plt.imshow(right, cmap = "gray")
    plt.colorbar(fraction = 0.046, pad = 0.04)

    if title_right != "":
        plt.title(title_right)

    plt.tight_layout()
    plt.show()
```

In [4]:
```
plot_left_and_right(left_original, right_original)
```



(B) Normalizing the images

Please normalize each image as follows. For $S_i(x, y)$, with $i = L$ or $i = R$, find S_i^{min} and S_i^{max} as the mininum and maximum of $S_i(x, y)$ across all pixel locations (x, y). Then, for $\hat{S} = 255$, do

$$ S_i(x, y) \rightarrow \hat{S} \frac{S_i(x, y) - S_i^{min}}{S_i^{max} - S_i^{min}} $$

Now $0 \leq S_i(x, y) \leq \hat{S}$. Round each $S_i(x, y)$ into an integer value so that $S_i(x, y)$ is an integer between 0 and $\hat{S}$.

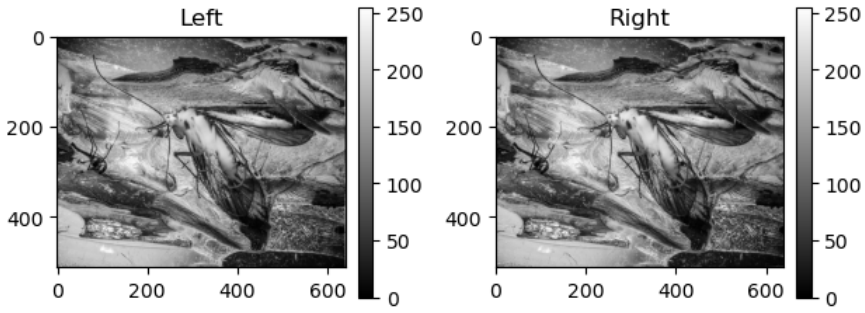
In [5]:
```
def normalize(image, s_hat = 255):
    s_min = np.min(image)
    s_max = np.max(image)

    image = s_hat * (image - s_min) / (s_max - s_min)
    image = np.round(image).astype(np.uint8)
```

```
    return image
```

In [6]:
```
left = normalize(left_original)
right = normalize(right_original)

print(f"Before normalization left: min = {np.min(left_original)}, max = {np.max(left_original)}")
print(f"After normalization left: min = {np.min(left)}, max = {np.max(left)}")

print(f"Before normalization right: min = {np.min(right_original)}, max = {np.max(right_original)}")
print(f"After normalization right: min = {np.min(right)}, max = {np.max(right)}")
```

```
Before normalization left: min = 0.0, max = 1.0
After normalization left: min = 0, max = 255
Before normalization right: min = 0.0, max = 1.0
After normalization right: min = 0, max = 255
```

In [7]:
```
plot_left_and_right(left, right)
```

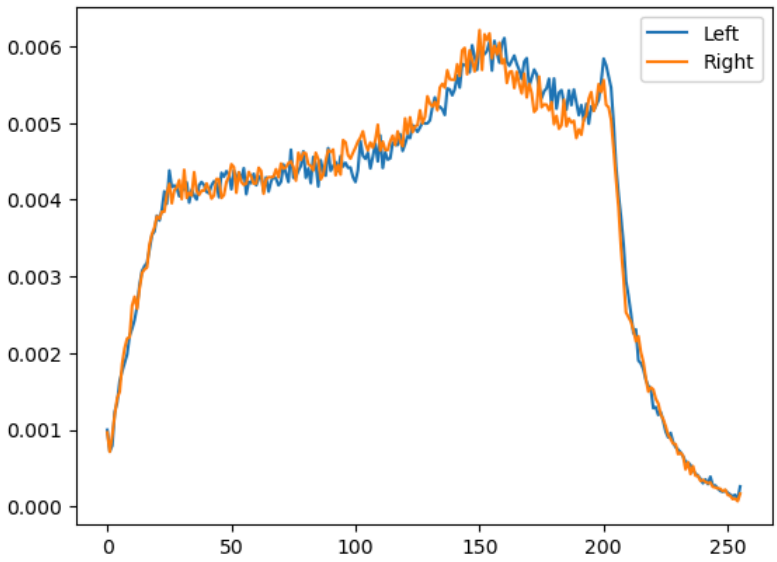


(C) Signal probability

For each $S_i(x, y)$, with $i = L$ or $i = R$, calculate the probability $P(S)$ for $S_i(x, y) = S$ across all (x, y). This means, for each $S = 0, 1, 2, \ldots, \hat{S}$, let $n(S)$ be the number of pixels (x, y) for which $S_i(x, y) = S$, and let N be the total number of pixels in S_i, then $P(S) = \frac{n(S)}{N}$. Plot out $P(S)$ vs S for each image S_i.

In [8]:
```
def probability(x, min_val = 0, max_val = 255):
    bins = np.linspace(min_val, max_val + 1, num = max_val + 2)
    return [bins[:-1], np.histogram(x, bins = bins)[0] / np.size(x)]
```

In [9]:
```
s, p = probability(left)
plt.plot(s, p, label = "Left")

s, p = probability(right)
plt.plot(s, p, label = "Right")

plt.legend()
plt.show()
```



(D) Signal entropy

For each $S_i(x, y)$, with $i = L$ or $i = R$, calculate the pixel entropy

$$H(S_i) = -\sum_{S_i} P(S_i)\ log_2\ P(S_i)$$

Please note that, if for some values S you have $P(S) = 0$, in such a case $P(S)\ log_2\ P(S) = 0$. Your computer will complain if you try to calculate $log_2\ P(S)$ when $P(S) = 0$. So omit these S values with zero $P(S)$ when doing the sum above.

Write out what $H(S)$ is for each image S_i.

In [10]:
```
def entropy(probs):
    return -np.sum([p * np.log2(p) if p > 0 else 0 for p in probs])
```

In [11]:
```
_, p = probability(left)
print(f"Entropy for i = L: {round(entropy(p), 2)}")

_, p = probability(right)
print(f"Entropy for i = R: {round(entropy(p), 2)}")
```

```
Entropy for i = L: 7.81
Entropy for i = R: 7.81
```

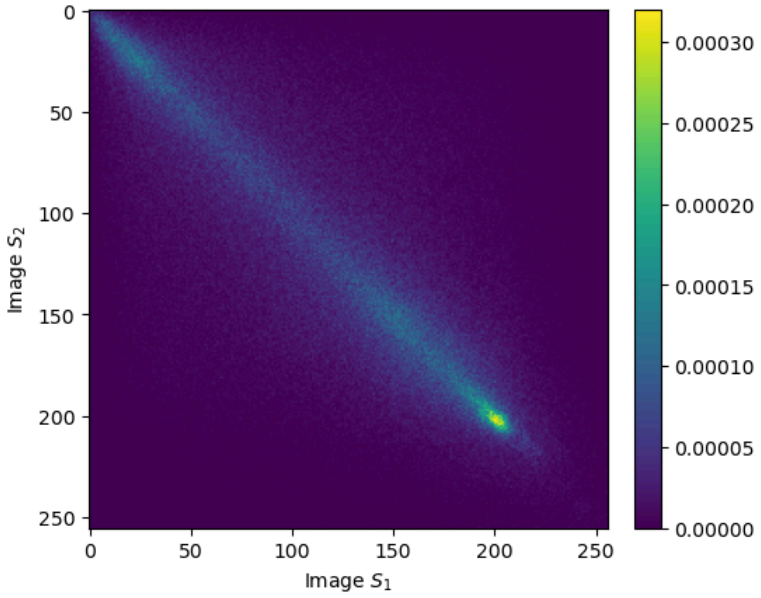
(E) Joint probability

Calculate the joint probability $P(S_1, S_2)$ of $S_L(x, y) = S_1$ and $S_R(x, y) = S_2$ for $S_1 = 0, 1, 2, \ldots, \hat{S}$ and $S_2 = 0, 1, 2, \ldots, \hat{S}$. This means, for each possible pair of values (S_1, S_2), go across all pixels (x, y) to find $n(S_1, S_2)$ as the number of pixels satisfying $S_L(x, y) = S_1$ and $S_R(x, y) = S_2$. Then $P(S_1, S_2) = \frac{n(S_1, S_2)}{N}$. Plot out $P(S_1, S_2)$ (which is the joint probability distribution function) versus S_1 and S_2.

In [12]:
```
def joint_probability(x, y, min_val = 0, max_val = 255):
    counts, _, _ = np.histogram2d(x.flatten(), y.flatten(), bins = max_val + 1)

    return counts / np.size(x)
```

In [13]:
```
probs = joint_probability(left, right)
```

In [14]:
```
plt.imshow(probs)
plt.xlabel(r'Image $S_1$')
plt.ylabel(r'Image $S_2$')
plt.colorbar(fraction = 0.046, pad = 0.04)
plt.show()
```



(F) Joint entropy

Calculate joint entropy as

$$H(S_1, S_2) = -\sum_{S_1, S_2} P(S_1, S_2)\ log_2\ P(S_1, S_2)$$

Write out the value for H(S_1, S_2).

In [15]:
```
print(f"Joint entropy: {round(entropy(probs.flatten()), 2)}")
```

```
Joint entropy: 14.72
```

(G) Mutual information

Calculate mutual information between corresponding pixels in the two images as

$$I(S_1, S_2) = \sum_{S_1, S_2} P(S_1, S_2)\, log_2\, \frac{P(S_1, S_2)}{P(S_1)P(S_2)} = H(S_1) + H(S_2) - H(S_1, S_2)$$

Write out the value for $I(S_1, S_2)$.

In [16]:
```
def mutual_information(x, y, min_val = 0, max_val = 255):
    _, p = probability(x, min_val = min_val, max_val = max_val)
    entropy_x = entropy(p)

    _, p = probability(y, min_val = min_val, max_val = max_val)
    entropy_y = entropy(p)

    p = joint_probability(x, y, min_val = min_val, max_val = max_val)
    entropy_xy = entropy(p.flatten())

    return entropy_x + entropy_y - entropy_xy
```

In [17]:
```
print(f"Mutual information: {round(mutual_information(left, right), 2)}")
```

```
Mutual information: 0.9
```

(H) Redundancy

Calculate the redundancy between the left and right eye images as

$$\text{Redundancy} = \frac{H(S_1) + H(S_2)}{H(S_1, S_2)} - 1$$

In [18]:
```
def redundancy(x, y, min_val = 0, max_val = 255):
    _, p = probability(x, min_val = min_val, max_val = max_val)
    entropy_x = entropy(p)

    _, p = probability(y, min_val = min_val, max_val = max_val)
    entropy_y = entropy(p)

    p = joint_probability(x, y, min_val = min_val, max_val = max_val)
    entropy_xy = entropy(p.flatten())

    return ((entropy_x + entropy_y) / entropy_xy) - 1
```

In [19]:
```
print(f"Redundancy: {round(redundancy(left, right), 2)}")
```

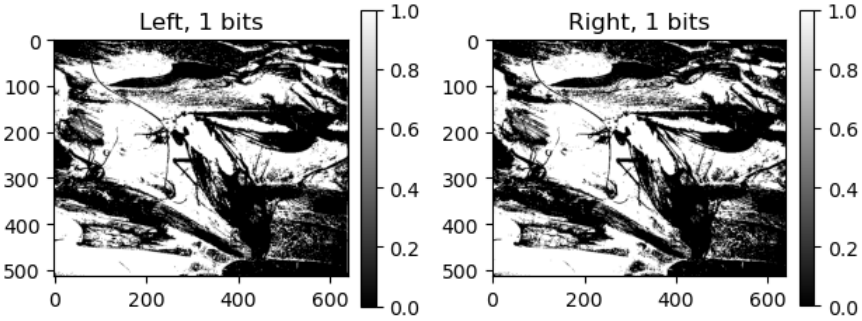
```
Redundancy: 0.06
```

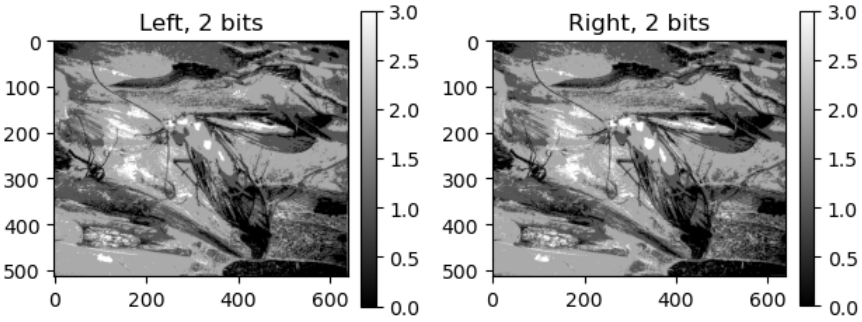
(I) Using n bits to present each image pixel

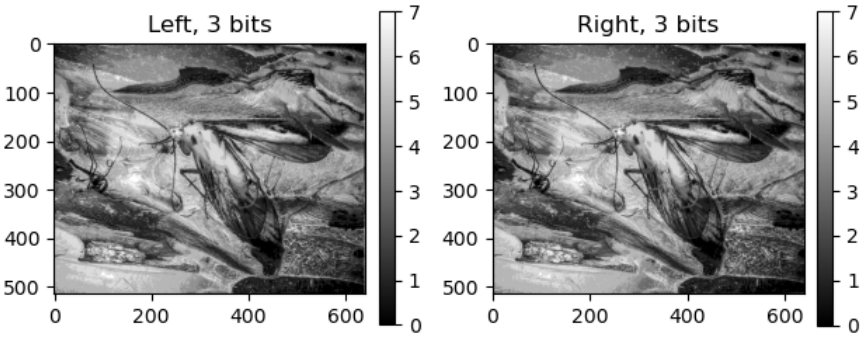
So far, you have done (B)-(H) when the highest pixel value is $\hat{S} = 255$. Now, repeat (B)-(H) for $\hat{S} = 127, 63, 31, 15, 7, 3, 1$. In other words, you can take $\hat{S} = 2^n - 1$ for $n = 1, 2, 3, \ldots, 8$ (so that $\hat{S} = 255$ when $n = 8$), so that you use n bits to present each image pixel. Plot $H(S_i)$, $H(S_1, S_2)$, $I(S_1, S_2)$ and Redundancy as functions of n. Also, please plot out the two images for each n value, and see if they make sense.

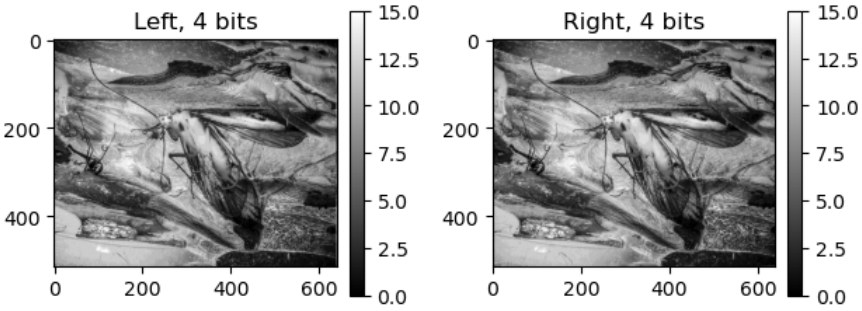
In [20]:
```
h_l = []
h_r = []
h_rl = []
i_rl = []
red = []
n_bits = []

for n_bit in range(1, 9):
    s_hat = round(math.pow(2, n_bit) - 1)
    n_bits.append(n_bit)

    _left = normalize(left_original, s_hat = s_hat)
    _right = normalize(right_original, s_hat = s_hat)

    plot_left_and_right(_left, _right, title_left = f"Left, {n_bit} bits", title_right = f"Right, {n_bit} bits"

    _, p = probability(_left, max_val = s_hat)
    h_l.append(entropy(p))

    _, p = probability(_right, max_val = s_hat)
    h_r.append(entropy(p))
```

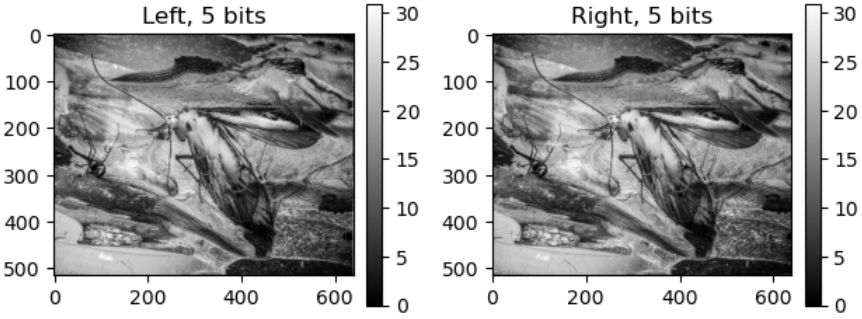
```
    p = joint_probability(_left, _right, max_val = s_hat)
    h_rl.append(entropy(p.flatten()))

    i_rl.append(mutual_information(_left, _right))

    red.append(redundancy(_left, _right, max_val = s_hat))
```

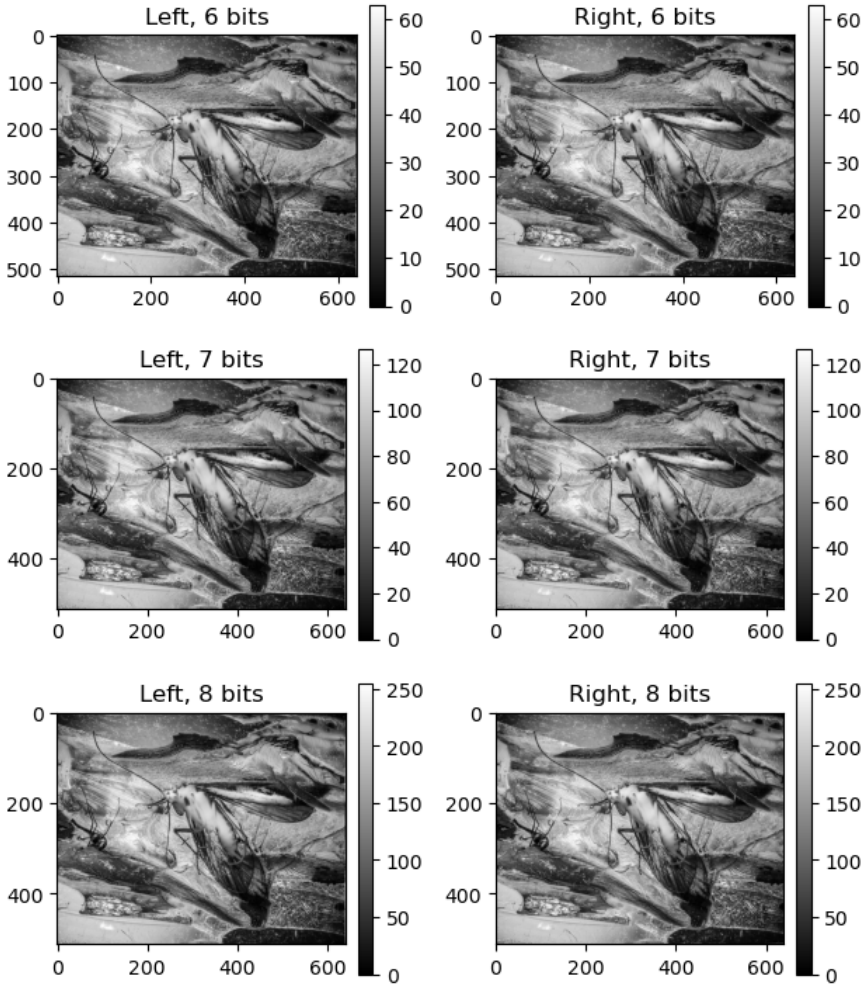




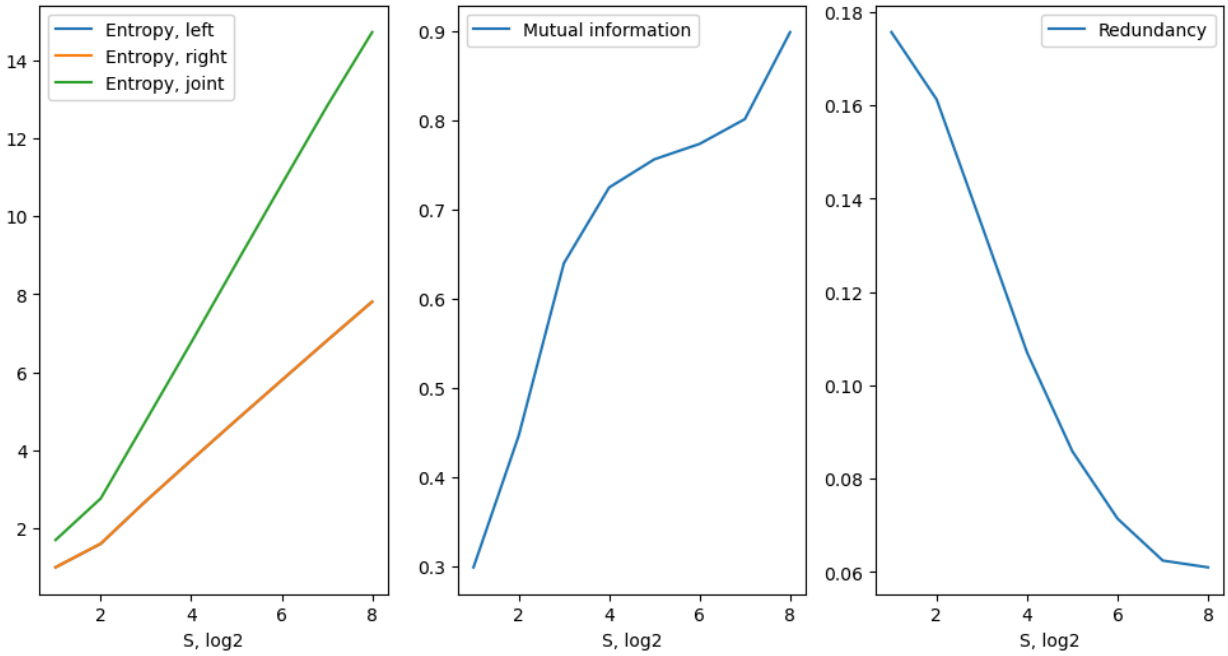








In [21]:
```
plt.figure(figsize = (12, 6))

plt.subplot(1, 3, 1)
plt.plot(n_bits, h_l, label = "Entropy, left")
plt.plot(n_bits, h_r, label = "Entropy, right")
plt.plot(n_bits, h_rl, label = "Entropy, joint")
plt.legend()
plt.xlabel("S, log2")

plt.subplot(1, 3, 2)
plt.plot(n_bits, i_rl, label = "Mutual information")
plt.legend()
plt.xlabel("S, log2")

plt.subplot(1, 3, 3)
plt.plot(n_bits, red, label = "Redundancy")
plt.legend()
plt.xlabel("S, log2")
plt.show()
```



(J) Correlation

Let us go back to take $\hat{S} = 255$, so that each $S(x, y)$ pixel is represented by 8 bits. For each image $S(x, y)$, let

$$\bar{S}_i = \sum_{x,y} \frac{S_i(x, y)}{N}$$

be the average value of $S(x, y)$ across all the image pixels. Then shift the pixel value

$$S_i(x, y) \to S_i(x, y) - \bar{S}_i$$

so that each image should now have a zero mean value.

Now, the correlation between $S_i(x, y)$ and $S_j(x, y)$ across pixels is

$$\begin{aligned} R^S_{ij} &= \langle S_i S_j \rangle \\ &= \frac{\sum_{x,y} S_i(x, y) S_j(x, y)}{\sum_{x,y} 1} \\ &= \frac{\sum_{x,y} S_i(x, y) S_j(x, y)}{N} \end{aligned}$$

So you can get a 2×2 matrix R^S with elements R^S_{ij} for $i = 1, 2$ and $j = 1, 2$.

$$R^S = \begin{pmatrix} R^S_{11} & R^S_{12} \\ R^S_{21} & R^S_{22} \end{pmatrix}$$

The diagonal element, R^S_{11} and R^S_{22} of this matrix are the variance of pixel values in each monocular image, and the off-diagonal values are the covariance between the two monocular images. Please write out this matrix value.

In [22]:
```
def covariance(x, y):
    _x = x - np.mean(x)
    _y = y - np.mean(y)
    N = np.size(x)

    return np.array([
        [np.sum(_x * _x) / N, np.sum(_x * _y) / N],
        [np.sum(_y * _x) / N, np.sum(_y * _y) / N]
    ])
```

In [23]:
```
def correlation_coefficient(x, y):
    _x = x - np.mean(x)
    _y = y - np.mean(y)

    return np.sum(_x * _y) / np.sqrt(np.sum(_x * _x) * np.sum(_y * _y))
```

In [24]:
```
def correlation(x, y):
    return np.array([
        [correlation_coefficient(x, x), correlation_coefficient(x, y)],
```

```
        [correlation_coefficient(y, x), correlation_coefficient(y, y)]
    ])
```

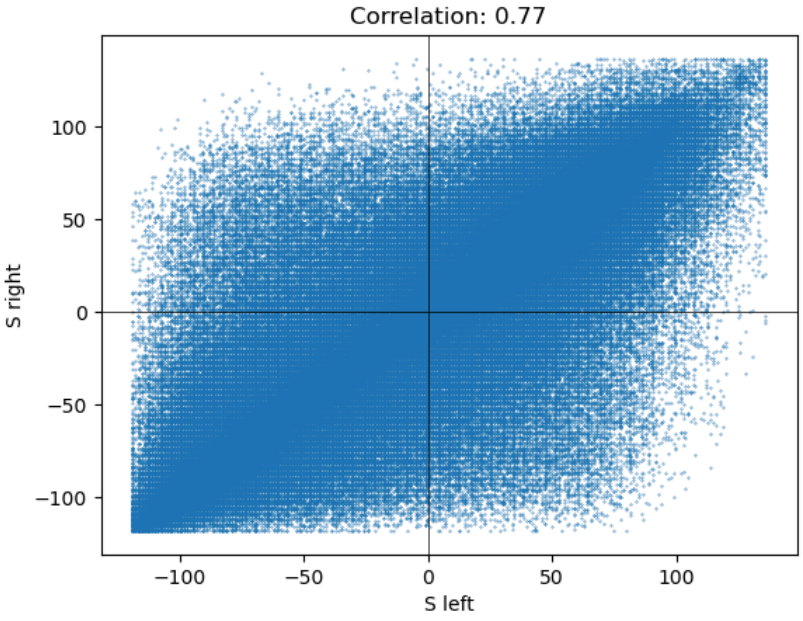
In [25]:
```
r = covariance(left, right)

print("Covariance:")
print(np.round(r, 2))

print("Correlation:")
print(np.round(correlation(left.flatten(), right.flatten()), 2))
```

```
Covariance:
[[3594.78 2744.89]
 [2744.89 3554.61]]
Correlation:
[[1.   0.77]
 [0.77 1.  ]]
```

(K) Scatter plot

Give a scatter plot of $S_L(x, y)$ versus $S_R(x, y)$. This means, start with a plot with horizontal and vertical axes, plot a point at location $(S_L(x, y), S_R(x, y))$, with the value of $S_L(x, y)$ and $S_R(x, y)$ on the horizontal and vertical axes respectively, for each pixel (x, y) in images $S_L(x, y)$ and $S_R(x, y)$. Compare your plot with one in Figure 1, and see if they look similar.

In [26]:
```
shifted_left = (left - np.mean(left)).flatten()
shifted_right = (right - np.mean(right)).flatten()

corr = correlation_coefficient(shifted_left, shifted_right)

plt.scatter(shifted_left, shifted_right, s = 0.1)
plt.axhline(y = 0, color = "black", linewidth = 0.5)
plt.axvline(x = 0, color = "black", linewidth = 0.5)
plt.xlabel("S left")
plt.ylabel("S right")
plt.title(f"Correlation: {round(corr, 2)}")
plt.show()
```



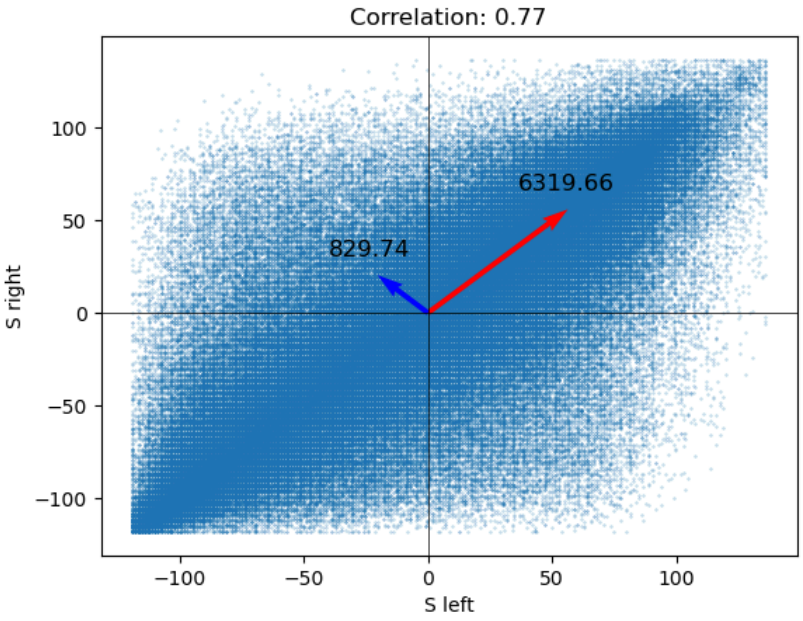
(L) Eigenvalues and eigenvectors

Calculate the eigenvalues and eigenvectors of the 2×2 matrix R^S. Plot each eigenvector as a vector in the scatter plot you obtained above in (K), and observe how each eigenvector is related to the character of this scatter plot of data, and observe how each eigenvalue is related to the variance of the data projected onto each eigenvector.

In [27]:
```
eigenvalues, eigenvectors = np.linalg.eig(r)
```

In [28]:
```
plt.scatter(shifted_left, shifted_right, s = 0.1, alpha = 0.5)
plt.axhline(y = 0, color = "black", linewidth = 0.5)
plt.axvline(x = 0, color = "black", linewidth = 0.5)
plt.xlabel("S left")
plt.ylabel("S right")
plt.title(f"Correlation: {round(corr, 2)}")

for i in range(len(eigenvalues)):
```

```
    eigen_x = np.sqrt(eigenvalues)[i] * eigenvectors[0, i]
    eigen_y = np.sqrt(eigenvalues)[i] * eigenvectors[1, i]

    plt.quiver(0, 0, eigen_x, eigen_y,
               angles='xy', scale_units='xy', scale=1, color=['r', 'b'][i])
    plt.annotate(text = round(eigenvalues[i], 2), xy = (eigen_x - 20, eigen_y + 10), size = 12)

plt.show()
```



(M) Image decorrelation

Define

$$S_+(x,y) = \frac{1}{\sqrt{2}}(S_L(x,y) + S_R(x,y))$$

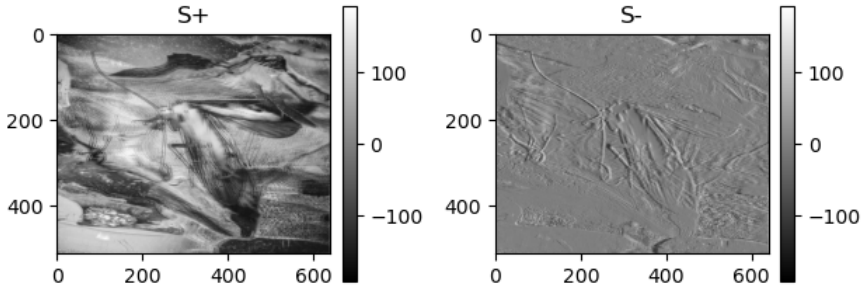
$$S_-(x,y) = \frac{1}{\sqrt{2}}(-S_L(x,y) + S_R(x,y))$$

Now $S_+(x,y)$ and $S_-(x,y)$ are two new images. Plot them out. Their pixel values are the projections of the original data $(S_L(x,y), S_R(x,y))$ onto the eigenvectors, or they are the principal components of the data.

In [29]:
```
left = normalize(left_original)
left = left - np.mean(left)
right = normalize(right_original)
right = right - np.mean(right)
```

In [30]:
```
s_plus = 1 / math.sqrt(2) * (left + right)
s_minus = 1 / math.sqrt(2) * (left - right)
```

In [31]:
```
plot_left_and_right(s_plus, s_minus, apply_lim = True, title_left = "S+", title_right = "S-")
```



(N) Correlation of de-correlated images

Calculate the 2×2 correlation matrix with elements

$$R^S_{ij} \equiv \langle S_i S_j \rangle$$

with $i = +$ or $-$ and $j = +$ or $-$. Write out the correlation matrix

$$R^S = \begin{pmatrix} R^S_{++} & R^S_{+-} \\ R^S_{-+} & R^S_{--} \end{pmatrix}$$

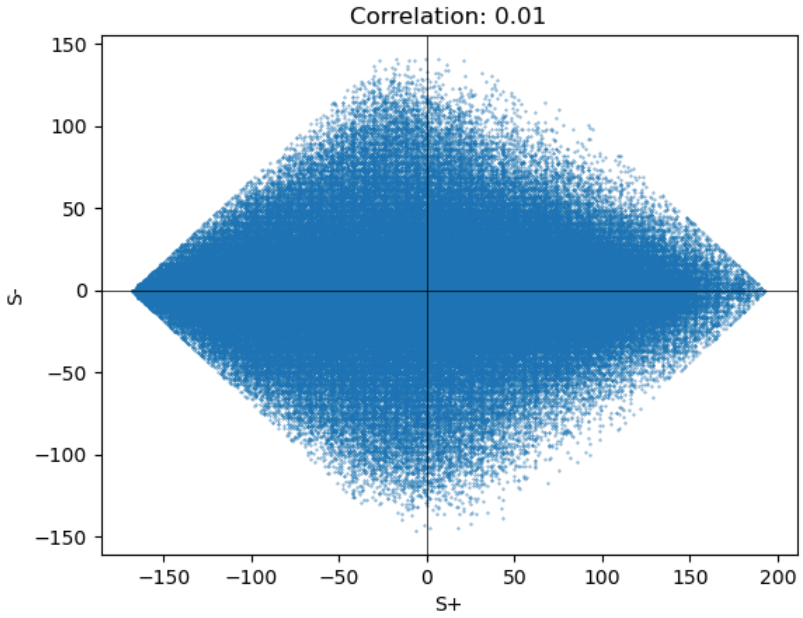
and verify that the off diagonal elements $R^S_{+-} = R^S_{-+} \approx 0$ in comparison to the diagonal elements. Reflect on what this means. Do a scatter plot of data points $(S_+(x,y), S_-(x,y))$ using all pixels (x,y) and observe how the matrix R^S reflects the character of the data in the scatter plot. Observe the variance of $S_+(x,y)$ versus the variance of $S_-(x,y)$. Which one has a larger variance?

In [32]:
```
print("Covariance: ")
print(covariance(s_plus.flatten(), s_minus.flatten()))

print("Correlation: ")
print(correlation(s_plus.flatten(), s_minus.flatten()))
```

```
Covariance:
[[6319.58539275   20.08577828]
 [  20.08577828  829.81012335]]
Correlation:
[[1.         0.00877112]
 [0.00877112 1.        ]]
```

The diagonal elements are the covariance between the two principal components of the data. The elements R^S_{++} and R^S_{--} reflect the variance. As expected, the S_+ has a larger variance compared to S_-.

In [33]:
```
shifted_s_plus = (s_plus - np.mean(s_plus)).flatten()
shifted_s_minus = (s_minus - np.mean(s_minus)).flatten()

corr = correlation_coefficient(shifted_s_plus, shifted_s_minus)

plt.scatter(shifted_s_plus, shifted_s_minus, s = 0.1)
plt.axhline(y = 0, color = "black", linewidth = 0.5)
plt.axvline(x = 0, color = "black", linewidth = 0.5)
plt.xlabel("S+")
plt.ylabel("S-")
plt.title(f"Correlation: {round(corr, 2)}")
plt.show()
```



(O) Gain control

Give gains g_+ and g_- to $S_+(x,y)$ and $S_-(x,y)$, respectively, to create the gain controlled images

$$O_+(x,y) = g_+ S_+(x,y)$$

$$O_-(x,y) = g_- S_-(x,y)$$

Let the gain values be

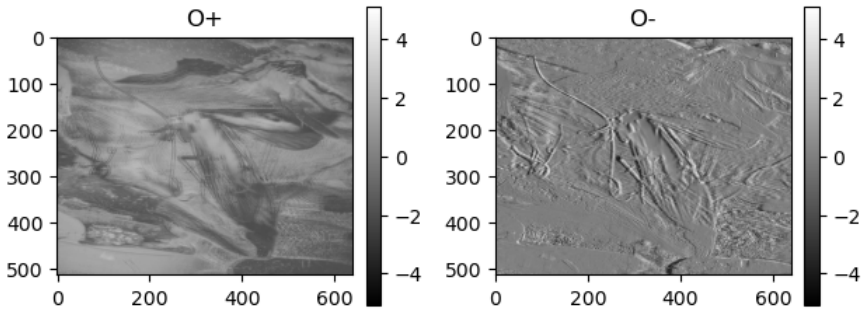
$$g_+ = \frac{1}{\sqrt{R^S_{++}}},\ g_- = \frac{1}{\sqrt{R^S_{--}}}$$

Plot the two images $O_+(x,y)$ and $O_-(x,y)$.

In [34]:
```
r = covariance(s_plus.flatten(), s_minus.flatten())
```

```
g_plus = 1 / math.sqrt(r[0][0])
g_minus = 1 / math.sqrt(r[1][1])

o_plus = g_plus * s_plus
o_minus = g_minus * s_minus
```

In [35]:
```
plot_left_and_right(o_plus, o_minus, apply_lim = True, title_left = "O+", title_right = "O-")
```



(P) Correlation of gain controlled images

Do a scatter plot of data $(O_+(x,y), O_-(x,y))$, and calculate the correlation matrix R^O with matrix element

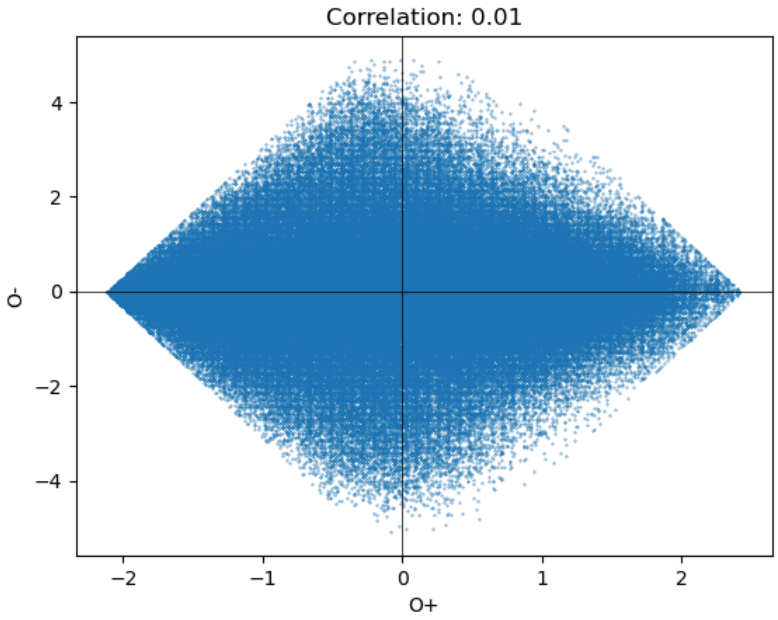
$$R^O_{ij} \equiv \langle O_i O_j \rangle$$

with $i = +$ or $-$ and $j = +$ or $-$. Write out the correlation matrix

$$R^O = \begin{pmatrix} R^O_{++} & R^O_{+-} \\ R^O_{-+} & R^O_{--} \end{pmatrix}$$

You should see that $O_+(x,y)$ and $O_-(x,y)$ are not correlated with each other, but have roughly the same variance. This is because we used the gains g_+ and g_- which are for whitening.

In [36]:
```
shifted_o_plus = (o_plus - np.mean(o_plus)).flatten()
shifted_o_minus = (o_minus - np.mean(o_minus)).flatten()

corr = correlation_coefficient(shifted_o_plus, shifted_o_minus)

plt.scatter(shifted_o_plus, shifted_o_minus, s = 0.1)
plt.axhline(y = 0, color = "black", linewidth = 0.5)
plt.axvline(x = 0, color = "black", linewidth = 0.5)
plt.xlabel("O+")
plt.ylabel("O-")
plt.title(f"Correlation: {round(corr, 2)}")
plt.show()
```



In [37]:
```
r = correlation(o_plus, o_minus)
np.round(r, 2)
```

Out[37]:
```
array([[1.  , 0.01],
       [0.01, 1.  ]])
```

(Q) Output images

Construct two new images $O_1(x, y)$ and $O_2(x, y)$ from $O_\pm(x, y)$ as follows

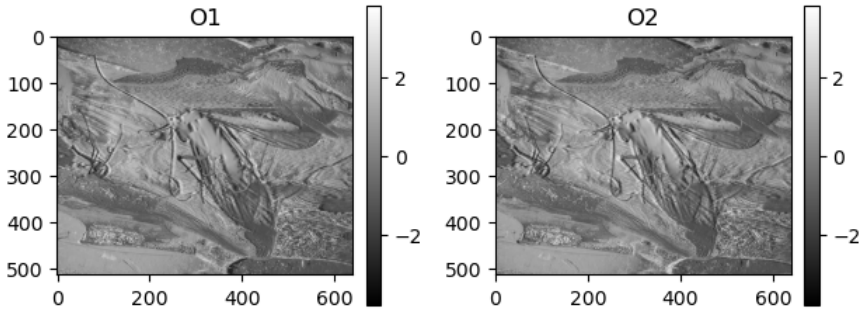
$$O_1(x, y) = \frac{1}{\sqrt{2}}(O_+(x, y) + O_-(x, y))$$

$$O_2(x, y) = \frac{1}{\sqrt{2}}(O_+(x, y) - O_-(x, y))$$

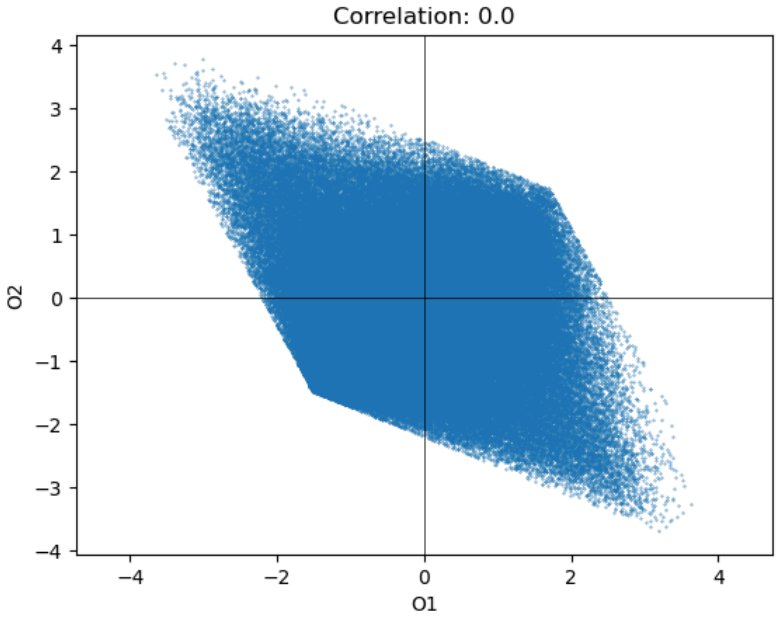
and plot them out. Also, do a scatter plot of data $(O_1(x, y), O_2(x, y))$. You should see that $O_1(x, y)$ and $O_2(x, y)$ are not correlated with each other, but have roughly the same variance.

In [38]:
```
o1 = 1 / math.sqrt(2) * (o_plus + o_minus)
o2 = 1 / math.sqrt(2) * (o_plus - o_minus)
```

In [39]:
```
plot_left_and_right(o1, o2, apply_lim = True, title_left = "O1", title_right = "O2")
```



In [40]:
```
shifted_o1 = o1 - np.mean(o1)
shifted_o2 = o2 - np.mean(o2)

corr = np.corrcoef(shifted_o1.flatten(), shifted_o2.flatten())[1][0]
lim_max = np.max([np.max(np.abs(shifted_o1)), np.max(np.abs(shifted_o2))]) * 1.25

plt.scatter(shifted_o1, shifted_o2, s = 0.1)
plt.axhline(y = 0, color = "black", linewidth = 0.5)
plt.axvline(x = 0, color = "black", linewidth = 0.5)
plt.xlabel("O1")
plt.ylabel("O2")
plt.title(f"Correlation: {round(corr, 2)}")
plt.xlim(-lim_max, lim_max)
plt.show()
```



(R) Output correlation

Construct the 2×2 correlation matrix with elements

$$R^O_{ij} \equiv \langle O_i O_j \rangle$$

with $i = 1$ or 2 and $j = 1$ or 2, for the matrix

$$R^O = \begin{pmatrix} R^O_{11} & R^O_{12} \\ R^O_{21} & R^O_{22} \end{pmatrix}$$

Relate this matrix with the scatter plot in (Q).

In [41]:
```
r = correlation(o1, o2)
np.round(r, 2)
```

Out[41]:
```
array([[1., 0.],
       [0., 1.]])
```

The elements $R^O_{21} = R^O_{12} = 0$ show that the outputs are not correlated with each other.

In [41]:

In [1]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import uniform
from matplotlib.image import imread
```

Homework 2 (4)

(S) Introducing noise

In step (J), you have the images $S_L(x,y)$ and $S_R(x,y)$. Add noise $N_L(x,y)$ and $N_R(x,y)$ to them, such that

$$S_L(x,y) \to S_L(x,y) + N_L(x,y)$$

$$S_R(x,y) \to S_R(x,y) + N_R(x,y)$$

At each pixel location (x,y), the noise $N_L(x,y)$ and $N_R(x,y)$ are independent of each other, and is a random number uniformly distributed in the range between $-N_{max}$ and N_{max}, i.e., $-N_{max} \le N_L(x,y) \le N_{max}$ and $-N_{max} \le N_R(x,y) \le N_{max}$. Noise in different pixels should be independent random numbers in this range. Choose $N_{max} = \sqrt{(R^S_{11} + R^S_{22})/2}$ (or you can choose a larger or smaller N). Plot out the noisy images $S_L(x,y)$ and $S_R(x,y)$. With these noisy images, repeat steps (B)-(R), and observe what happens and reflect on the result. You can play with the value of N_{max} and see different effects by different N_{max} values, and try to reflect why.

In [2]:
```
def noise(nmax = 1, n = 1):
    return uniform.rvs(loc = -nmax, scale = nmax, size = n)
```

In [3]:
```
def noisy_image(original, nmax = 1):
    noise_data = np.reshape(noise(nmax, n = np.size(original)), newshape = original.shape)
    return original + noise_data
```

In [4]:
```
def covariance(x, y):
    _x = x - np.mean(x)
    _y = y - np.mean(y)
    N = np.size(x)

    return np.array([
        [np.sum(_x * _x) / N, np.sum(_x * _y) / N],
        [np.sum(_y * _x) / N, np.sum(_y * _y) / N]
    ])
```

In [5]:
```
def correlation_coefficient(x, y):
    _x = x - np.mean(x)
    _y = y - np.mean(y)

    return np.sum(_x * _y) / np.sqrt(np.sum(_x * _x) * np.sum(_y * _y))
```

In [6]:
```
def correlation(x, y):
    return np.array([
        [correlation_coefficient(x, x), correlation_coefficient(x, y)],
        [correlation_coefficient(y, x), correlation_coefficient(y, y)]
    ])
```

In [7]:
```
left_original = imread("resources/left.png")
right_original = imread("resources/right.png")
```

In [8]:
```
r = correlation(left_original, right_original)
```

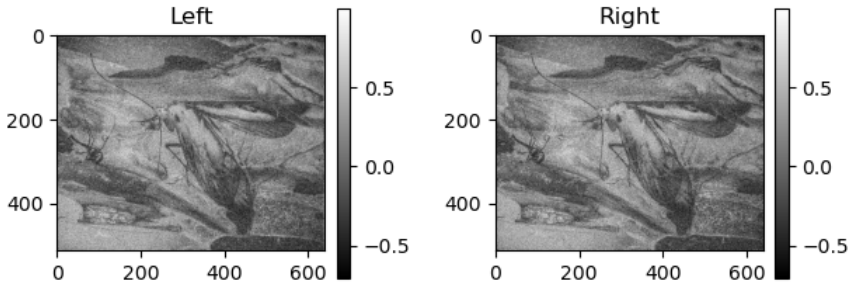
In [9]:
```
nmax = math.sqrt(r[0][0] + r[1][1]) / 2

print(f"Using nmax = {nmax}")

left_original = noisy_image(left_original, nmax = nmax)
right_original = noisy_image(right_original, nmax = nmax)
```

```
Using nmax = 0.7071067811865476
```

In [10]:
```
def plot_left_and_right(left, right, apply_lim = False, title_left = "Left", title_right = "Right"):
    max_lim = np.max(np.abs([left, right]))

    plt.subplot(1, 2, 1)
    if apply_lim:
        plt.imshow(left, cmap = "gray", vmin = -max_lim, vmax = max_lim)
    else:
        plt.imshow(left, cmap = "gray")
    plt.colorbar(fraction = 0.046, pad = 0.04)
```

```
    if title_left != "":
        plt.title(title_left)

    plt.subplot(1, 2, 2)
    if apply_lim:
        plt.imshow(right, cmap = "gray", vmin = -max_lim, vmax = max_lim)
    else:
        plt.imshow(right, cmap = "gray")
    plt.colorbar(fraction = 0.046, pad = 0.04)

    if title_right != "":
        plt.title(title_right)

    plt.tight_layout()
    plt.show()
```

In [11]:
```
plot_left_and_right(left_original, right_original)
```



(B) Normalizing the images

Please normalize each image as follows. For $S_i(x, y)$, with $i = L$ or $i = R$, find S_i^{min} and S_i^{max} as the mininum and maximum of $S_i(x, y)$ across all pixel locations (x, y). Then, for $\hat{S} = 255$, do

$$ S_i(x, y) \rightarrow \hat{S} \frac{S_i(x, y) - S_i^{min}}{S_i^{max} - S_i^{min}} $$

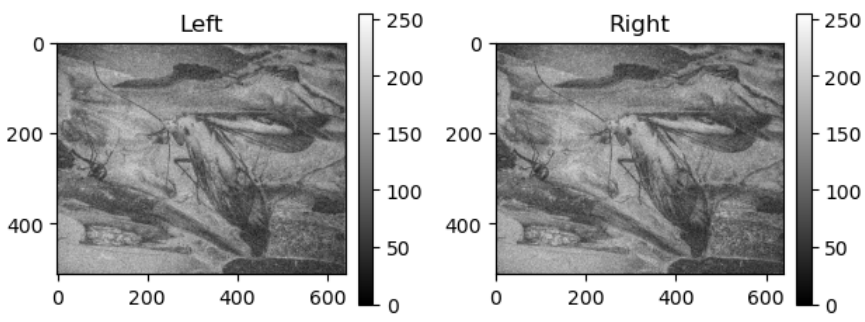
Now $0 \leq S_i(x, y) \leq \hat{S}$. Round each $S_i(x, y)$ into an integer value so that $S_i(x, y)$ is an integer between 0 and $\hat{S}$.

In [12]:
```
def normalize(image, s_hat = 255):
    s_min = np.min(image)
    s_max = np.max(image)

    image = s_hat * (image - s_min) / (s_max - s_min)
    image = np.round(image).astype(np.uint8)

    return image
```

In [13]:
```
left = normalize(left_original)
right = normalize(right_original)

print(f"Before normalization left: min = {np.min(left_original)}, max = {np.max(left_original)}")
print(f"After normalization left: min = {np.min(left)}, max = {np.max(left)}")

print(f"Before normalization right: min = {np.min(right_original)}, max = {np.max(right_original)}")
print(f"After normalization right: min = {np.min(right)}, max = {np.max(right)}")
```

```
Before normalization left: min = -0.7055875859343761, max = 0.9990245389659125
After normalization left: min = 0, max = 255
Before normalization right: min = -0.7047442708259074, max = 0.9985928893377534
After normalization right: min = 0, max = 255
```

In [14]:
```
plot_left_and_right(left, right)
```

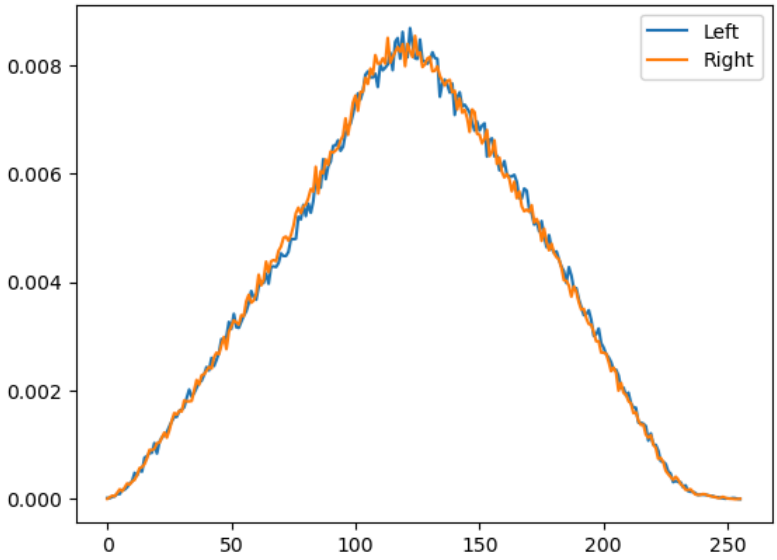


(C) Signal probability

For each $S_i(x, y)$, with $i = L$ or $i = R$, calculate the probability $P(S)$ for $S_i(x, y) = S$ across all (x, y). This means, for each $S = 0, 1, 2, \ldots, \hat{S}$, let $n(S)$ be the number of pixels (x, y) for which $S_i(x, y) = S$, and let N be the total number of pixels in S_i, then $P(S) = \frac{n(S)}{N}$. Plot out $P(S)$ vs S for each image S_i.

In [15]:
```
def probability(x, min_val = 0, max_val = 255):
    bins = np.linspace(min_val, max_val + 1, num = max_val + 2)
    return [bins[:-1], np.histogram(x, bins = bins)[0] / np.size(x)]
```

In [16]:
```
s, p = probability(left)
plt.plot(s, p, label = "Left")

s, p = probability(right)
plt.plot(s, p, label = "Right")

plt.legend()
plt.show()
```



(D) Signal entropy

For each $S_i(x, y)$, with $i = L$ or $i = R$, calculate the pixel entropy

$$H(S_i) = -\sum_{S_i} P(S_i)\ log_2\ P(S_i)$$

Please note that, if for some values S you have $P(S) = 0$, in such a case $P(S)\ log_2\ P(S) = 0$. Your computer will complain if you try to calculate $log_2\ P(S)$ when $P(S) = 0$. So omit these S values with zero $P(S)$ when doing the sum above.

Write out what $H(S)$ is for each image S_i.

In [17]:
```
def entropy(probs):
    return -np.sum([p * np.log2(p) if p > 0 else 0 for p in probs])
```

In [18]:
```
_, p = probability(left)
print(f"Entropy for i = L: {round(entropy(p), 2)}")

_, p = probability(right)
print(f"Entropy for i = R: {round(entropy(p), 2)}")
```

```
Entropy for i = L: 7.56
Entropy for i = R: 7.56
```

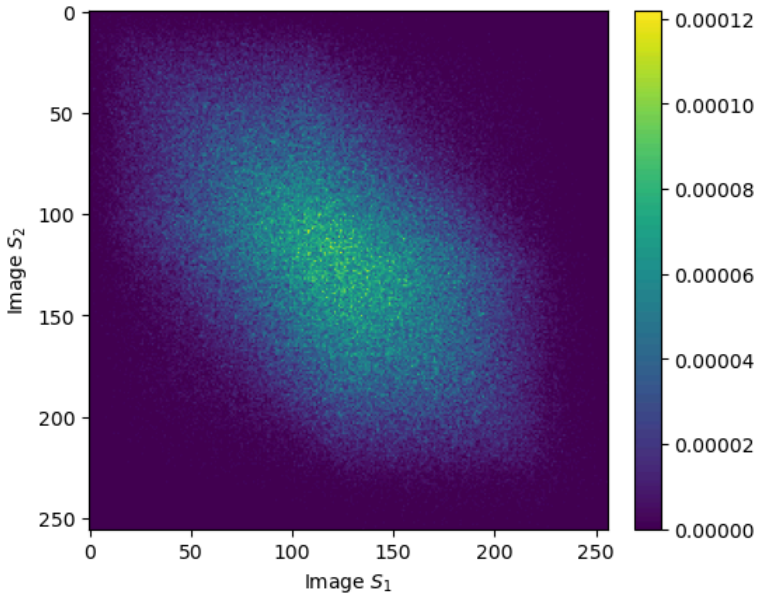
(E) Joint probability

Calculate the joint probability $P(S_1, S_2)$ of $S_L(x, y) = S_1$ and $S_R(x, y) = S_2$ for $S_1 = 0, 1, 2, \ldots, \hat{S}$ and $S_2 = 0, 1, 2, \ldots, \hat{S}$. This means, for each possible pair of values (S_1, S_2), go across all pixels (x, y) to find $n(S_1, S_2)$ as the number of pixels satisfying $S_L(x, y) = S_1$ and $S_R(x, y) = S_2$. Then $P(S_1, S_2) = \frac{n(S_1, S_2)}{N}$. Plot out $P(S_1, S_2)$ (which is the joint probability distribution function) versus S_1 and S_2.

In [19]:
```
def joint_probability(x, y, min_val = 0, max_val = 255):
    counts, _, _ = np.histogram2d(x.flatten(), y.flatten(), bins = max_val + 1)

    return counts / np.size(x)
```

In [20]:
```
probs = joint_probability(left, right)
```

In [21]:
```
plt.imshow(probs)
plt.xlabel(r'Image $S_1$')
plt.ylabel(r'Image $S_2$')
plt.colorbar(fraction = 0.046, pad = 0.04)
plt.show()
```



(F) Joint entropy

Calculate joint entropy as

$$H(S_1, S_2) = -\sum_{S_1, S_2} P(S_1, S_2)\ log_2\ P(S_1, S_2)$$

Write out the value for H(S_1, S_2).

In [22]:
```
print(f"Joint entropy: {round(entropy(probs.flatten()), 2)}")
```

```
Joint entropy: 14.85
```

(G) Mutual information

Calculate mutual information between corresponding pixels in the two images as

$$I(S_1, S_2) = \sum_{S_1, S_2} P(S_1, S_2)\ log_2\ \frac{P(S_1, S_2)}{P(S_1)P(S_2)} = H(S_1) + H(S_2) - H(S_1, S_2)$$

Write out the value for $I(S_1, S_2)$.

In [23]:
```
def mutual_information(x, y, min_val = 0, max_val = 255):
    _, p = probability(x, min_val = min_val, max_val = max_val)
    entropy_x = entropy(p)

    _, p = probability(y, min_val = min_val, max_val = max_val)
    entropy_y = entropy(p)

    p = joint_probability(x, y, min_val = min_val, max_val = max_val)
    entropy_xy = entropy(p.flatten())

    return entropy_x + entropy_y - entropy_xy
```

In [24]:
```
print(f"Mutual information: {round(mutual_information(left, right), 2)}")
```

```
Mutual information: 0.27
```

(H) Redundancy

Calculate the redundancy between the left and right eye images as

$$\text{Redundancy} = \frac{H(S_1) + H(S_2)}{H(S_1, S_2)} - 1$$

In [25]:
```
def redundancy(x, y, min_val = 0, max_val = 255):
    _, p = probability(x, min_val = min_val, max_val = max_val)
    entropy_x = entropy(p)

    _, p = probability(y, min_val = min_val, max_val = max_val)
    entropy_y = entropy(p)

    p = joint_probability(x, y, min_val = min_val, max_val = max_val)
    entropy_xy = entropy(p.flatten())

    return ((entropy_x + entropy_y) / entropy_xy) - 1
```

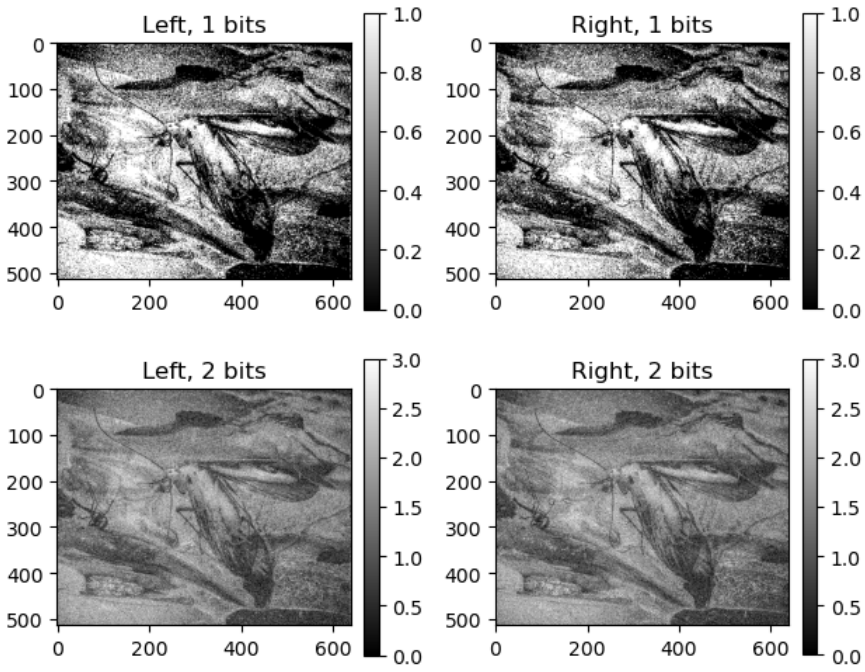
In [26]:
```
print(f"Redundancy: {round(redundancy(left, right), 2)}")
```

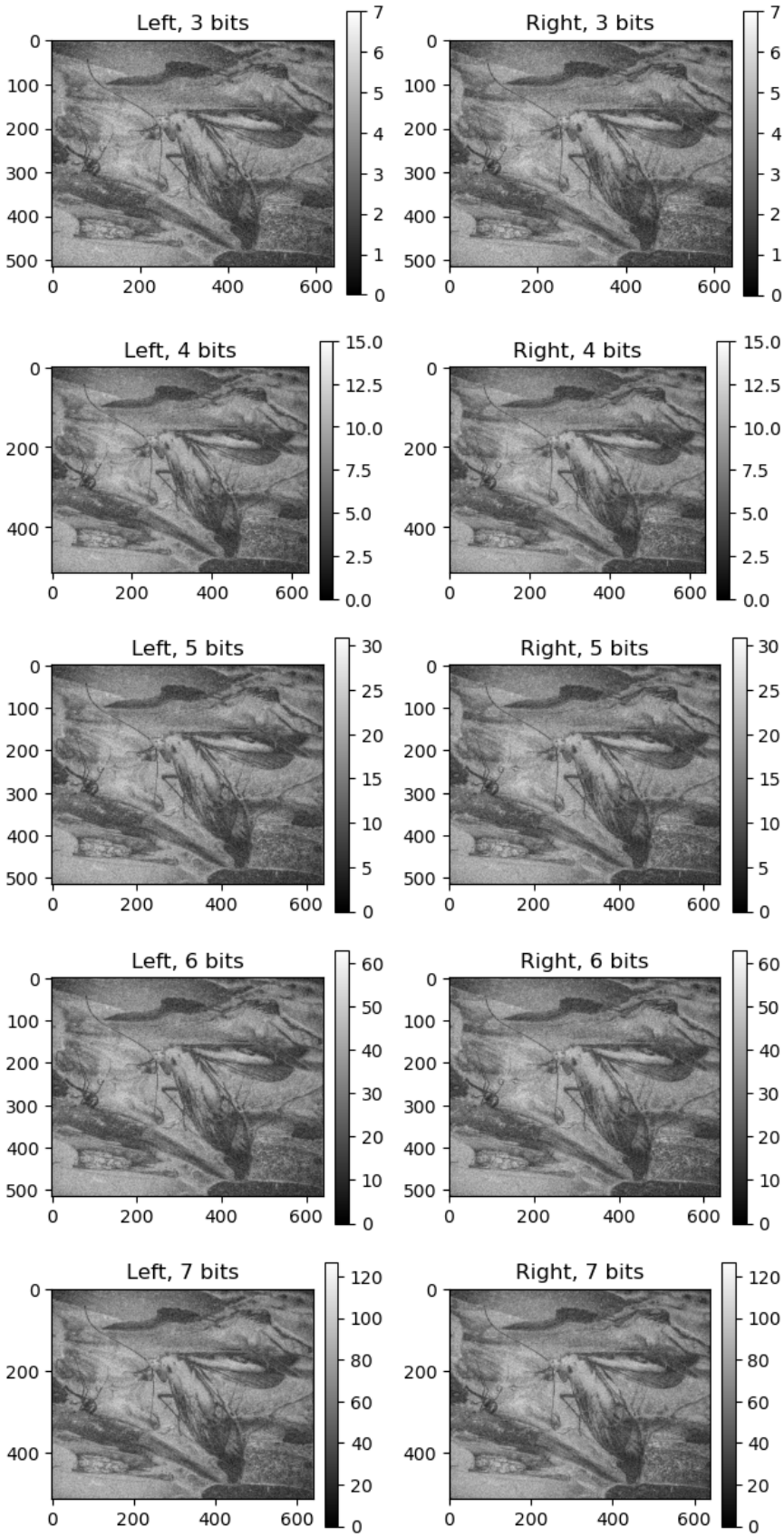
```
Redundancy: 0.02
```

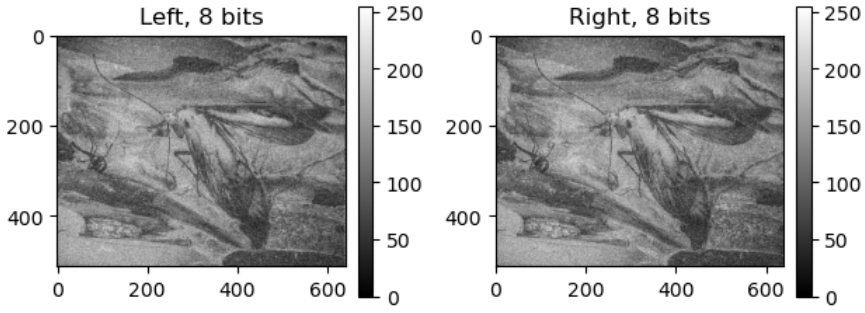
(I) Using n bits to present each image pixel

So far, you have done (B)-(H) when the highest pixel value is $\hat{S} = 255$. Now, repeat (B)-(H) for $\hat{S} = 127, 63, 31, 15, 7, 3, 1$. In other words, you can take $\hat{S} = 2^n - 1$ for $n = 1, 2, 3, \ldots, 8$ (so that $\hat{S} = 255$ when $n = 8$), so that you use n bits to present each image pixel. Plot $H(S_i)$, $H(S_1, S_2)$, $I(S_1, S_2)$ and Redundancy as functions of n. Also, please plot out the two images for each n value, and see if they make sense.

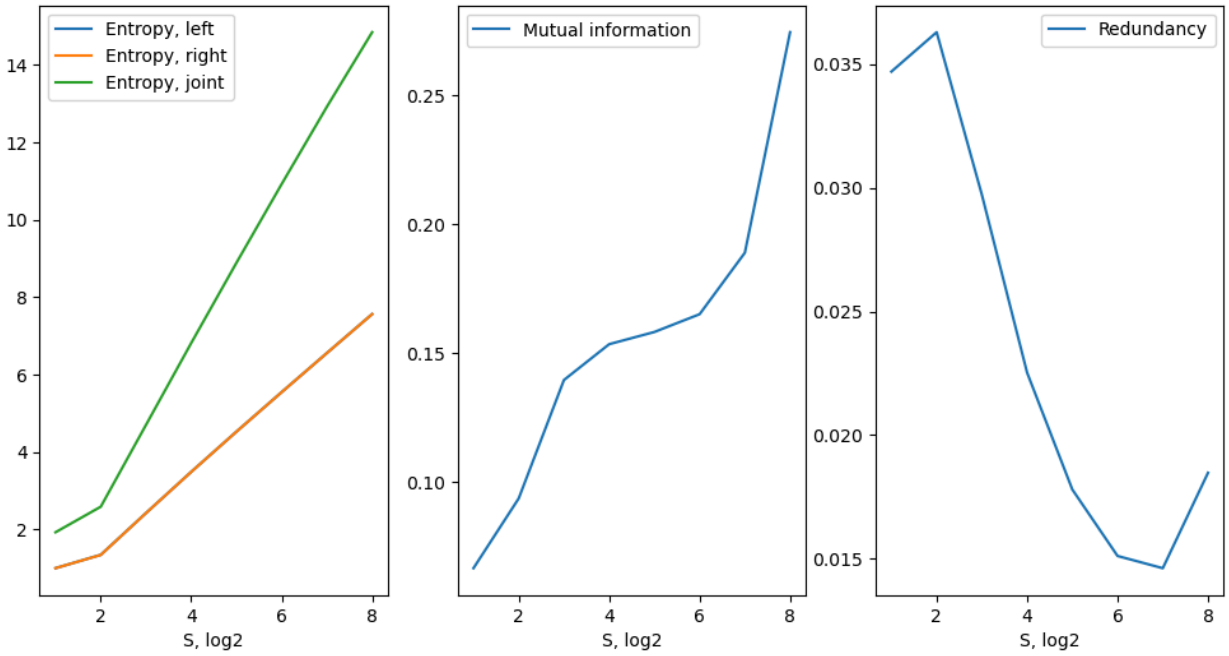
In [27]:
```
h_l = []
h_r = []
h_rl = []
i_rl = []
red = []
n_bits = []

for n_bit in range(1, 9):
    s_hat = round(math.pow(2, n_bit) - 1)
    n_bits.append(n_bit)

    _left = normalize(left_original, s_hat = s_hat)
    _right = normalize(right_original, s_hat = s_hat)

    plot_left_and_right(_left, _right, title_left = f"Left, {n_bit} bits", title_right = f"Right, {n_bit} bits"

    _, p = probability(_left, max_val = s_hat)
    h_l.append(entropy(p))

    _, p = probability(_right, max_val = s_hat)
    h_r.append(entropy(p))

    p = joint_probability(_left, _right, max_val = s_hat)
    h_rl.append(entropy(p.flatten()))

    i_rl.append(mutual_information(_left, _right))

    red.append(redundancy(_left, _right, max_val = s_hat))
```




Left, 3 bits
Right, 3 bits
Left, 4 bits
Right, 4 bits
Left, 5 bits
Right, 5 bits
Left, 6 bits
Right, 6 bits
Left, 7 bits
Right, 7 bits



In [28]:
```
plt.figure(figsize = (12, 6))

plt.subplot(1, 3, 1)
plt.plot(n_bits, h_l, label = "Entropy, left")
plt.plot(n_bits, h_r, label = "Entropy, right")
plt.plot(n_bits, h_rl, label = "Entropy, joint")
plt.legend()
plt.xlabel("S, log2")

plt.subplot(1, 3, 2)
plt.plot(n_bits, i_rl, label = "Mutual information")
plt.legend()
plt.xlabel("S, log2")

plt.subplot(1, 3, 3)
plt.plot(n_bits, red, label = "Redundancy")
plt.legend()
plt.xlabel("S, log2")
plt.show()
```



(J) Correlation

Let us go back to take $\hat{S} = 255$, so that each $S(x, y)$ pixel is represented by 8 bits. For each image $S(x, y)$, let

$$\bar{S}_i = \sum_{x,y} \frac{S_i(x, y)}{N}$$

be the average value of $S(x, y)$ across all the image pixels. Then shift the pixel value

$$S_i(x, y) \to S_i(x, y) - \bar{S}_i$$

so that each image should now have a zero mean value.

Now, the correlation between $S_i(x, y)$ and $S_j(x, y)$ across pixels is

$$
\begin{aligned}
R^S_{ij} &= \langle S_i S_j \rangle \\
&= \frac{\sum_{x,y} S_i(x,y) S_j(x,y)}{\sum_{x,y} 1} \\
&= \frac{\sum_{x,y} S_i(x,y) S_j(x,y)}{N}
\end{aligned}
$$

So you can get a 2×2 matrix R^S with elements R^S_{ij} for $i = 1, 2$ and $j = 1, 2$.

$$
R^S = \begin{pmatrix} R^S_{11} & R^S_{12} \\ R^S_{21} & R^S_{22} \end{pmatrix}
$$

The diagonal element, R^S_{11} and R^S_{22} of this matrix are the variance of pixel values in each monocular image, and the off-diagonal values are the covariance between the two monocular images. Please write out this matrix value.

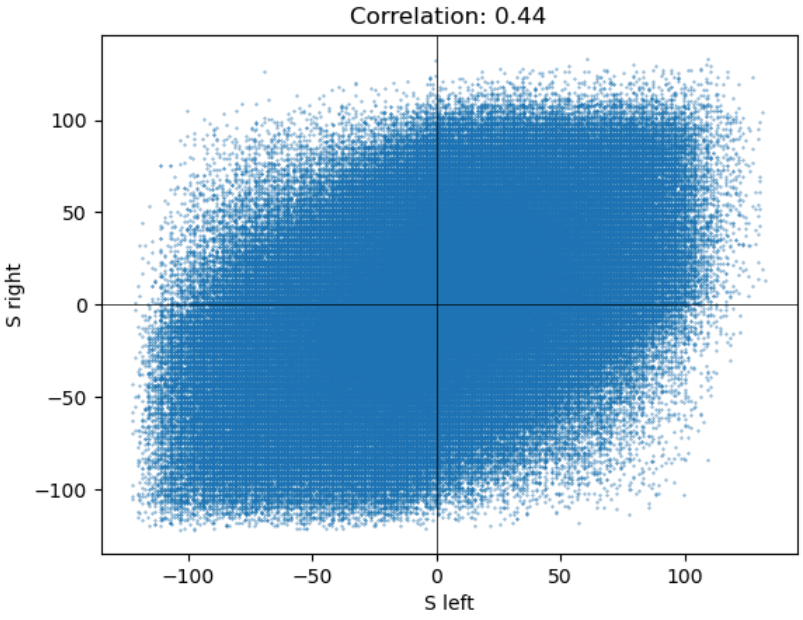
In [29]:
```
r = covariance(left, right)

print("Covariance:")
print(np.round(r, 2))

print("Correlation:")
print(np.round(correlation(left.flatten(), right.flatten()), 2))
```

```
Covariance:
[[2165.65  947.39]
 [ 947.39 2154.8 ]]
Correlation:
[[1.   0.44]
 [0.44 1.  ]]
```

(K) Scatter plot

Give a scatter plot of $S_L(x,y)$ versus $S_R(x,y)$. This means, start with a plot with horizontal and vertical axes, plot a point at location $(S_L(x,y), S_R(x,y))$, with the value of $S_L(x,y)$ and $S_R(x,y)$ on the horizontal and vertical axes respectively, for each pixel (x,y) in images $S_L(x,y)$ and $S_R(x,y)$. Compare your plot with one in Figure 1, and see if they look similar.

In [30]:
```
shifted_left = (left - np.mean(left)).flatten()
shifted_right = (right - np.mean(right)).flatten()

corr = correlation_coefficient(shifted_left, shifted_right)

plt.scatter(shifted_left, shifted_right, s = 0.1)
plt.axhline(y = 0, color = "black", linewidth = 0.5)
plt.axvline(x = 0, color = "black", linewidth = 0.5)
plt.xlabel("S left")
plt.ylabel("S right")
plt.title(f"Correlation: {round(corr, 2)}")
plt.show()
```

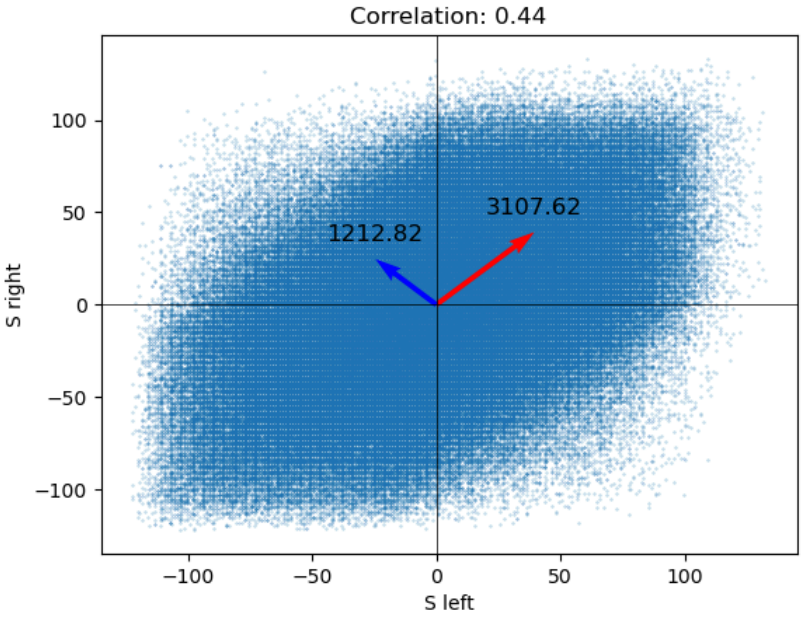


(L) Eigenvalues and eigenvectors

Calculate the eigenvalues and eigenvectors of the 2×2 matrix R^S. Plot each eigenvector as a vector in the scatter plot you obtained above in (K), and observe how each eigenvector is related to the character of this scatter plot of data, and observe how each eigenvalue is related to the variance of the data projected onto each eigenvector.

In [31]:
```
eigenvalues, eigenvectors = np.linalg.eig(r)
```

In [32]:
```
plt.scatter(shifted_left, shifted_right, s = 0.1, alpha = 0.5)
plt.axhline(y = 0, color = "black", linewidth = 0.5)
plt.axvline(x = 0, color = "black", linewidth = 0.5)
plt.xlabel("S left")
plt.ylabel("S right")
plt.title(f"Correlation: {round(corr, 2)}")

for i in range(len(eigenvalues)):
    eigen_x = np.sqrt(eigenvalues)[i] * eigenvectors[0, i]
    eigen_y = np.sqrt(eigenvalues)[i] * eigenvectors[1, i]

    plt.quiver(0, 0, eigen_x, eigen_y,
               angles='xy', scale_units='xy', scale=1, color=['r', 'b'][i])
    plt.annotate(text = round(eigenvalues[i], 2), xy = (eigen_x - 20, eigen_y + 10), size = 12)

plt.show()
```



(M) Image decorrelation

Define

$$S_+(x,y) = \frac{1}{\sqrt{2}}(S_L(x,y) + S_R(x,y))$$

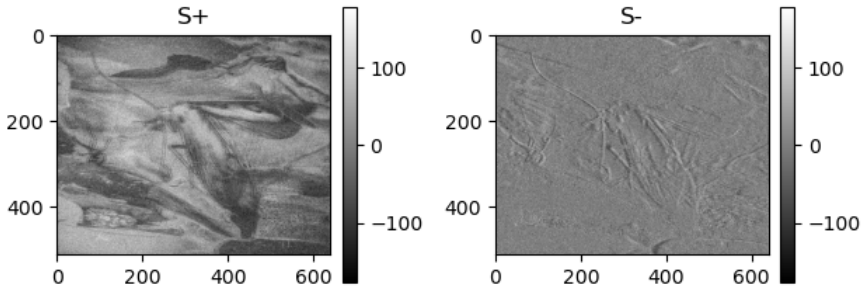
$$S_-(x,y) = \frac{1}{\sqrt{2}}(-S_L(x,y) + S_R(x,y))$$

Now $S_+(x,y)$ and $S_-(x,y)$ are two new images. Plot them out. Their pixel values are the projections of the original data $(S_L(x,y), S_R(x,y))$ onto the eigenvectors, or they are the principal components of the data.

In [33]:
```
left = normalize(left_original)
left = left - np.mean(left)
right = normalize(right_original)
right = right - np.mean(right)
```

In [34]:
```
s_plus = 1 / math.sqrt(2) * (left + right)
s_minus = 1 / math.sqrt(2) * (left - right)
```

In [35]:
```
plot_left_and_right(s_plus, s_minus, apply_lim = True, title_left = "S+", title_right = "S-")
```



(N) Correlation of de-correlated images

Calculate the 2×2 correlation matrix with elements

$$R^S_{ij} \equiv \langle S_i S_j \rangle$$

with $i = +$ or $-$ and $j = +$ or $-$. Write out the correlation matrix

$$R^S = \begin{pmatrix} R^S_{++} & R^S_{+-} \\ R^S_{-+} & R^S_{--} \end{pmatrix}$$

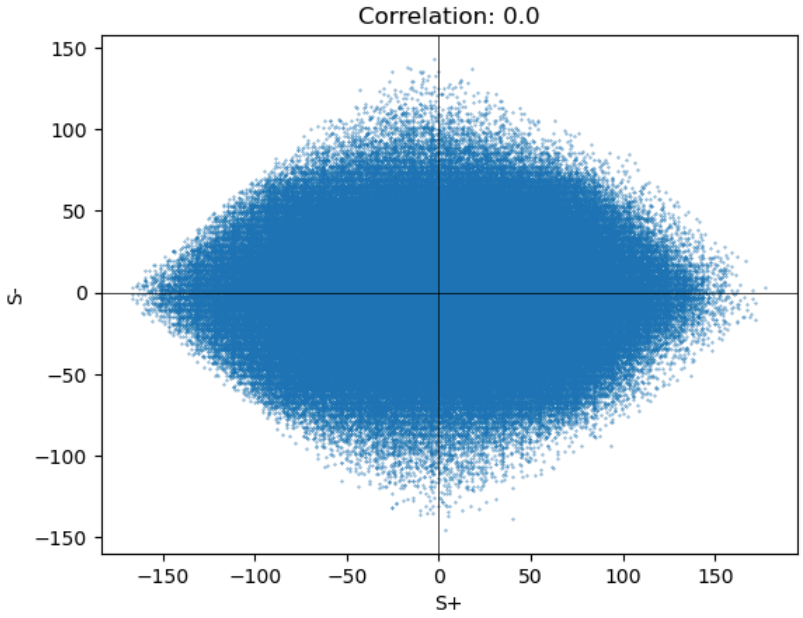
and verify that the off diagonal elements $R^S_{+-} = R^S_{-+} \approx 0$ in comparison to the diagonal elements. Reflect on what this means. Do a scatter plot of data points $(S_+(x,y), S_-(x,y))$ using all pixels (x,y) and observe how the matrix R^S reflects the character of the data in the scatter plot. Observe the variance of $S_+(x,y)$ versus the variance of $S_-(x,y)$. Which one has a larger variance?

In [36]:
```
print("Covariance: ")
print(covariance(s_plus.flatten(), s_minus.flatten()))

print("Correlation: ")
print(correlation(s_plus.flatten(), s_minus.flatten()))
```

```
Covariance:
[[3107.60713373    5.42638652]
 [   5.42638652 1212.83690231]]
Correlation:
[[1.         0.00279509]
 [0.00279509 1.        ]]
```

The diagonal elements are the covariance between the two principal components of the data. The elements R^S_{++} and R^S_{--} reflect the variance. As expected, the S_+ has a larger variance compared to S_-.

In [37]:
```
shifted_s_plus = (s_plus - np.mean(s_plus)).flatten()
shifted_s_minus = (s_minus - np.mean(s_minus)).flatten()

corr = correlation_coefficient(shifted_s_plus, shifted_s_minus)

plt.scatter(shifted_s_plus, shifted_s_minus, s = 0.1)
plt.axhline(y = 0, color = "black", linewidth = 0.5)
plt.axvline(x = 0, color = "black", linewidth = 0.5)
plt.xlabel("S+")
plt.ylabel("S-")
plt.title(f"Correlation: {round(corr, 2)}")
plt.show()
```



(O) Gain control

Give gains g_+ and g_- to $S_+(x,y)$ and $S_-(x,y)$, respectively, to create the gain controlled images

$$O_+(x,y) = g_+ S_+(x,y)$$

$$O_-(x,y) = g_- S_-(x,y)$$

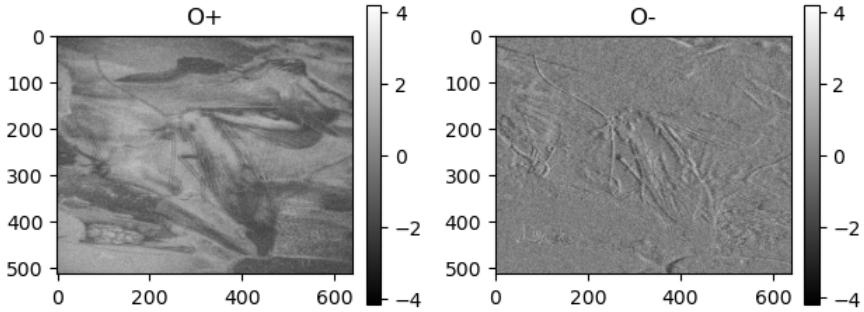
Let the gain values be

$$g_+ = \frac{1}{\sqrt{R^S_{++}}},\ g_- = \frac{1}{\sqrt{R^S_{--}}}$$

Plot the two images $O_+(x,y)$ and $O_-(x,y)$.

In [38]:
```
r = covariance(s_plus.flatten(), s_minus.flatten())

g_plus = 1 / math.sqrt(r[0][0])
g_minus = 1 / math.sqrt(r[1][1])

o_plus = g_plus * s_plus
o_minus = g_minus * s_minus
```

In [39]:
```
plot_left_and_right(o_plus, o_minus, apply_lim = True, title_left = "O+", title_right = "O-")
```



(P) Correlation of gain controlled images

Do a scatter plot of data $(O_+(x,y), O_-(x,y))$, and calculate the correlation matrix R^O with matrix element

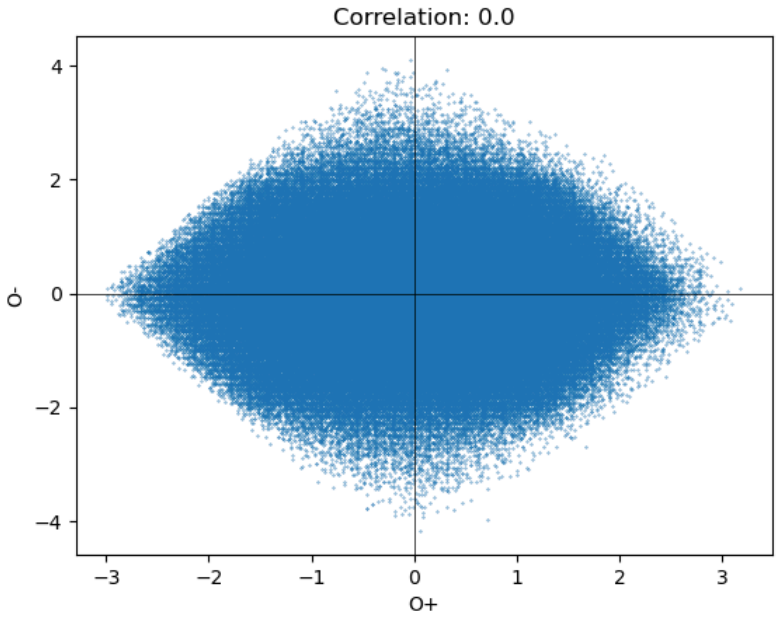
$$R^O_{ij} \equiv \langle O_i O_j \rangle$$

with $i = +$ or $-$ and $j = +$ or $-$. Write out the correlation matrix

$$R^O = \begin{pmatrix} R^O_{++} & R^O_{+-} \\ R^O_{-+} & R^O_{--} \end{pmatrix}$$

You should see that $O_+(x,y)$ and $O_-(x,y)$ are not correlated with each other, but have roughly the same variance. This is because we used the gains g_+ and g_- which are for whitening.

In [40]:
```
shifted_o_plus = (o_plus - np.mean(o_plus)).flatten()
shifted_o_minus = (o_minus - np.mean(o_minus)).flatten()

corr = correlation_coefficient(shifted_o_plus, shifted_o_minus)

plt.scatter(shifted_o_plus, shifted_o_minus, s = 0.1)
plt.axhline(y = 0, color = "black", linewidth = 0.5)
plt.axvline(x = 0, color = "black", linewidth = 0.5)
plt.xlabel("O+")
plt.ylabel("O-")
plt.title(f"Correlation: {round(corr, 2)}")
plt.show()
```



In [41]:
```
r = correlation(o_plus, o_minus)
np.round(r, 2)
```

Out[41]:
```
array([[1., 0.],
       [0., 1.]])
```

(Q) Output images

Construct two new images $O_1(x,y)$ and $O_2(x,y)$ from $O_\pm(x,y)$ as follows

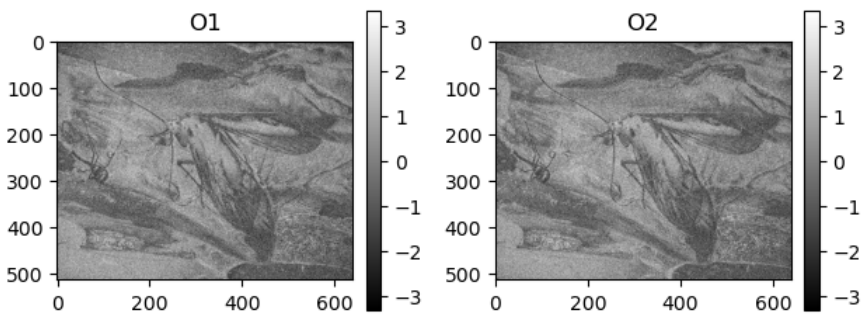
$$O_1(x,y) = \frac{1}{\sqrt{2}}(O_+(x,y) + O_-(x,y))$$

$$O_2(x,y) = \frac{1}{\sqrt{2}}(O_+(x,y) - O_-(x,y))$$

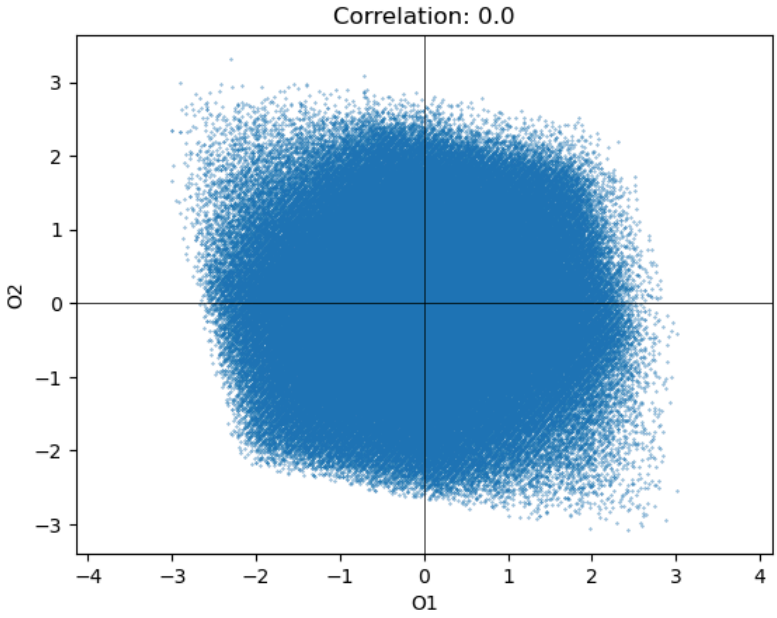
and plot them out. Also, do a scatter plot of data $(O_1(x,y), O_2(x,y))$. You should see that $O_1(x,y)$ and $O_2(x,y)$ are not correlated with each other, but have roughly the same variance.

In [42]:
```
o1 = 1 / math.sqrt(2) * (o_plus + o_minus)
o2 = 1 / math.sqrt(2) * (o_plus - o_minus)
```

In [43]:
```
plot_left_and_right(o1, o2, apply_lim = True, title_left = "O1", title_right = "O2")
```



In [44]:
```
shifted_o1 = o1 - np.mean(o1)
shifted_o2 = o2 - np.mean(o2)

corr = np.corrcoef(shifted_o1.flatten(), shifted_o2.flatten())[1][0]
lim_max = np.max([np.max(np.abs(shifted_o1)), np.max(np.abs(shifted_o2))]) * 1.25

plt.scatter(shifted_o1, shifted_o2, s = 0.1)
plt.axhline(y = 0, color = "black", linewidth = 0.5)
plt.axvline(x = 0, color = "black", linewidth = 0.5)
plt.xlabel("O1")
plt.ylabel("O2")
plt.title(f"Correlation: {round(corr, 2)}")
plt.xlim(-lim_max, lim_max)
plt.show()
```



(R) Output correlation

Construct the 2×2 correlation matrix with elements

$$R^O_{ij} \equiv \langle O_i O_j \rangle$$

with $i = 1$ or 2 and $j = 1$ or 2, for the matrix

$$R^O = \begin{pmatrix} R^O_{11} & R^O_{12} \\ R^O_{21} & R^O_{22} \end{pmatrix}$$

Relate this matrix with the scatter plot in (Q).

In [45]:
```
r = correlation(o1, o2)
np.round(r, 2)
```

Out[45]:
```
array([[1., 0.],
       [0., 1.]])
```

The elements $R^O_{21} = R^O_{12} = 0$ show that the outputs are not correlated with each other.

In [45]:

In [1]:
```
import math
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.image import imread
from scipy.stats import uniform
```

Homework 2 (5)

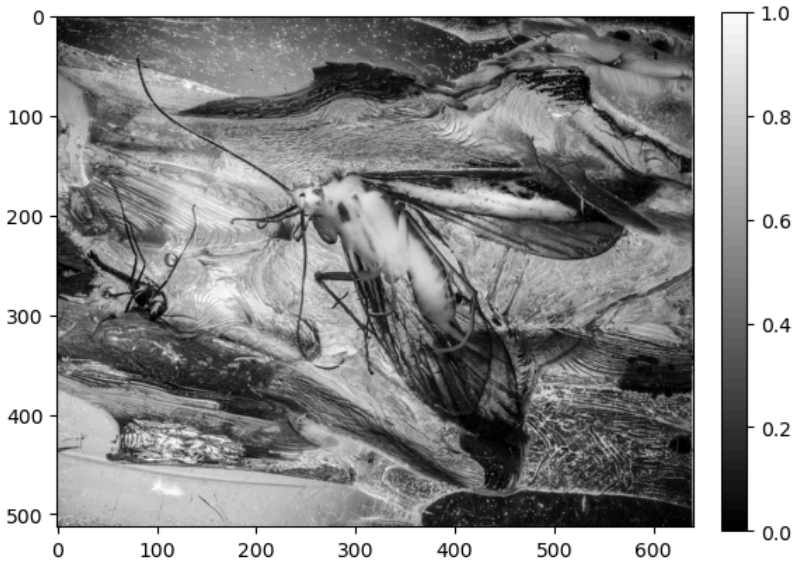
(A) Image selection

Take a photograph of a natural environment. This image is denoted as $S(x, y)$. Here x and y are the coordinates of the image pixel locations. For example $x = 1, 2, \ldots, 256$ and $y = 1, 2, \ldots, 256$. You may find some examples online, or one of your own photographs. An example is the image of peppers in Figure 2. If the original image is colored, remove the color. Plot out the image as a luminance image.

In [2]:
```
def plot_image(image):
    plt.imshow(image, cmap = "gray")
    plt.colorbar(fraction = 0.046, pad = 0.04)
    plt.show()
```

In [3]:
```
insect = imread("resources/left.png")
```

In [4]:
```
plot_image(insect)
```



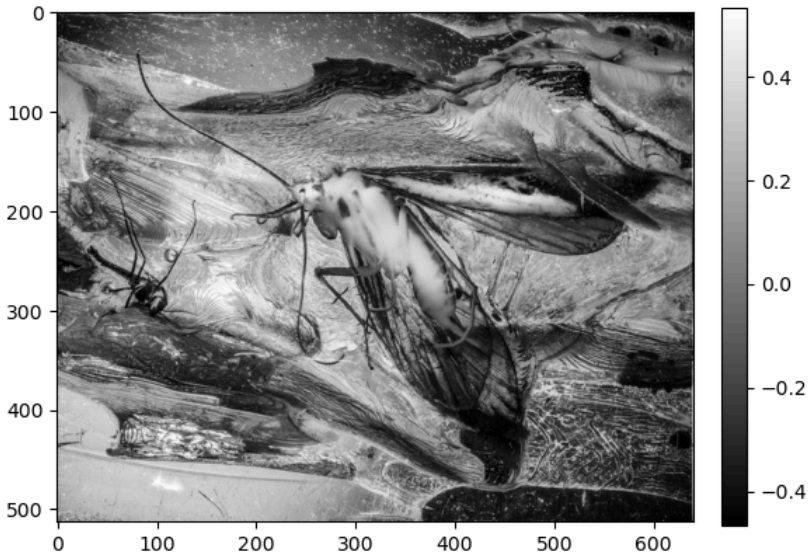
(B) Rescale the image

Transform the image to make it zero mean for convenience. Hence, get the pixel value average $\hat{S} = \sum_{x,y} \frac{S(x,y)}{N}$, where N is the total number of pixels in this image. Then make $S(x, y) \to S(x, y) - \hat{S}$. Now $S(x, y)$ has zero mean value across the image.

In [5]:
```
insect = insect - np.mean(insect)

print(f"Mean: {round(np.mean(insect), 5)}")
```

```
Mean: 0.0
```

In [6]:
```
plot_image(insect)
```



(C) Correlation

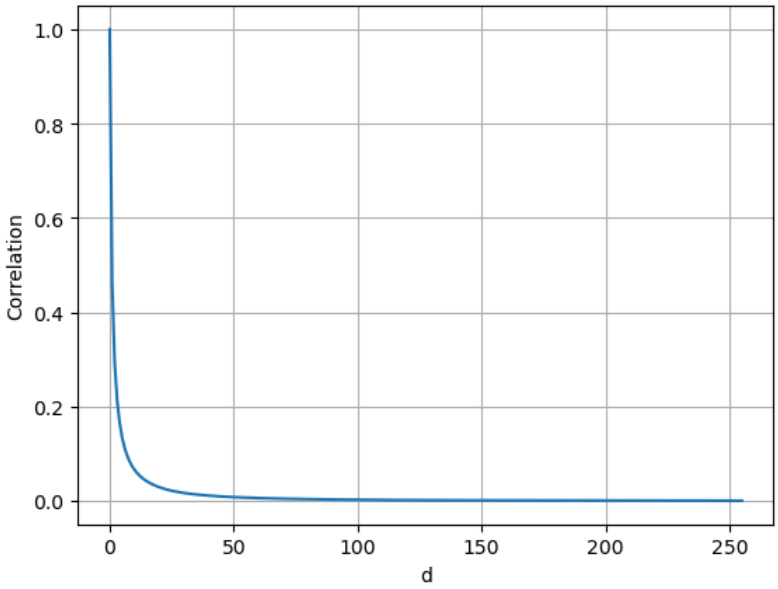
Observe how neighboring pixels are correlated with each other in their luminance values. For example, if you have one pixel at (x, y) and another at $(x + d, y)$, these two pixels are horizontally displaced by a distance d. Calculate

$$R^S(d) = \sum_{x,y} \frac{S(x, y)S(x + d, y)}{N'}$$

where N' the total number of pairs of pixels that you include in your sum $\sum_{x,y}$. Note that some pixels (x, y) are too close to the edge of the images that $(x + d, y)$ in not included in the image. This $R^S(d)$ approximates the correlation as a function of d. Try to calculate for a number of different d values, and see how $R^S(d)$ decays with d. Take a moment to consider what this entails for efficient coding of such images. (Many programming languages have routines for calculating correlations, e.g., in Matlab, you have xcorr().)

In [7]:
```
def pixel_correlation(image, max_d):
    height, width = image.shape
    corr = np.zeros(max_d)
    N_prime = 0

    for d in range(1, max_d + 1):
        sum_corr = np.sum(image[:, :-d] * image[:, d:])
        num_pairs = (height * (width - d))

        N_prime += num_pairs
        corr[d - 1] = sum_corr / N_prime

    # Scale
    corr = (corr - np.min(corr)) / (np.max(corr) - np.min(corr))
    return corr
```

In [8]:
```
corr = pixel_correlation(insect, max_d = round(insect.shape[0] / 2))

plt.plot(corr)
plt.xlabel("d")
plt.ylabel("Correlation")
plt.grid()
plt.show()
```



(D) Fourier power spectrum

If you have not yet done it in your previous exercise, calculate a Fourier power spectrum. Let us practice just a simplified version. Let L be the range of x values of your image pixels (x, y). Then, take Fourier frequency k as one of these discrete values,

$$k = \frac{n \cdot 2\pi}{L}$$

using $(L + 1)$ integer values $n = 0, 1, 2, \ldots, L$, (this k is in the unit of radian/pixel) and calculate

$$S_c(k) = \sum_{x,y} S(x, y) cos(kx)$$

$$S_s(k) = \sum_{x,y} S(x, y) sin(kx)$$

Then the signal power for this Fourier frequency k is (up to a scale constant)

$$|S(k)|^2 = [S_c(k)]^2 + [S_s(k)]^2$$

Plot out $|S(k)|^2$ as a function of k. For typical photographs of natural scenes, $|S(k)|^2$ decays with k as $1/k^2$. To see the results clearly, use log scale on both the vertical and horizontal axes, like in the lower left panel in Figure 2.

You may like to average $|S(k)|^2$ over multiple images (make all images the same size for convenience) to see the trend of $|S(k)|^2$ versus k more clearly.

In [9]:
```
def g_c(kernel, k):
    res = 0

    for i in range(kernel.shape[0]):
        for j in range(kernel.shape[1]):
            pos = j
            res += kernel[i, j] * np.cos(k * pos)

    return res
```

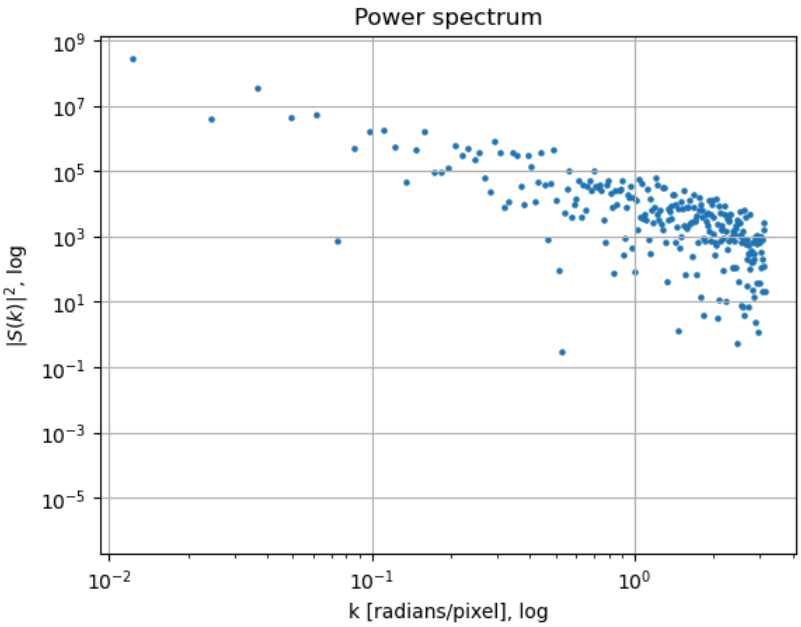
In [10]:
```
def g_s(kernel, k):
    res = 0

    for i in range(kernel.shape[0]):
        for j in range(kernel.shape[1]):
            pos = j
            res += kernel[i, j] * np.cos(k * pos)

    return res
```

In [11]:
```
def power_spectrum(image, k):
    g_k = []
    for _k in k:
        gs = math.pow(g_s(image, _k), 2)
        gc = math.pow(g_c(image, _k), 2)
        g_k.append(gs + gc)

    return g_k
```

In [12]:
```
def k_step(pixels):
    return [ n * 2 * math.pi / pixels for n in range(0, int(pixels / 2)) ]
```

In [13]:
```
ks = k_step(insect.shape[0])
sk = power_spectrum(insect, ks)
```

In [14]:
```
plt.scatter(ks, sk, s = 5)
plt.grid()
plt.xlabel("k [radians/pixel], log")
plt.ylabel(r"$|S(k)|^2$, log")
plt.xscale("log")
plt.yscale("log")
plt.title("Power spectrum")
plt.show()
```

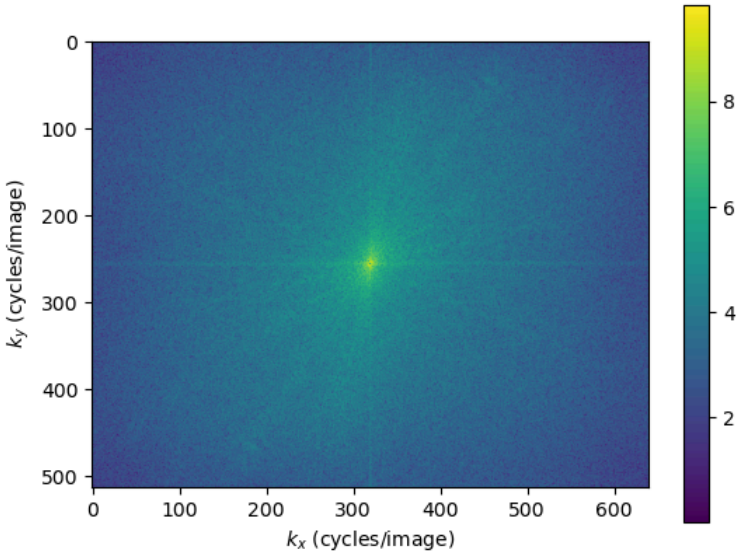


(E) Fourier transform

The steps in (D) only looked at Fourier frequencies for a Fourier wave that is vertical, so the frequency k is along the horizontal or x direction. However, the Fourier wave can be in any direction. So the frequency k should be a vector with a horizontal component k_x and a vertical component k_y, so that the magnitude $k = \sqrt{k_x^2 + k_y^2}$. Then, for each image $S(x, y)$, which span horizontally in x and vertically in y, one can obtain its Fourier component $S(k)$ for vector frequencies k. Usually, your programming language has a routine for Fourier transform, try to use it to get $S(k)$.

In [15]:
```
fourier_transform = np.fft.fft2(insect)
centered_spectrum = np.fft.fftshift(fourier_transform)
magnitude_spectrum = np.abs(centered_spectrum)
inverse_transform = np.fft.ifft2(fourier_transform)

image_height, image_width = insect.shape

frequencies_x = np.fft.fftfreq(image_width) * image_width
frequencies_y = np.fft.fftfreq(image_height) * image_height

shifted_frequencies_x = np.fft.fftshift(frequencies_x)
shifted_frequencies_y = np.fft.fftshift(frequencies_y)
```

In [16]:
```
plt.imshow(np.log1p(magnitude_spectrum))
plt.xlabel(r"$k_x$ (cycles/image)")
plt.ylabel("$k_y$ (cycles/image)")
plt.colorbar()
plt.show()
```



(F) Fourier power spectrum of a noisy image

In (D) and (E), you may find that your power spectrum, the $|S(k)^2|$ versus k, does not decay with k in a way expected when k is very large. This may be because your image is noisy, so that $|S(k)^2|$ include the power of the white noise. However, if your image is not noisy enough, or to practice seeing the effect of noise on $|S(k)^2|$, let us make the image noisy by adding noise as follows

$$S(x,y) \to S(x,y) + N(x,y)$$

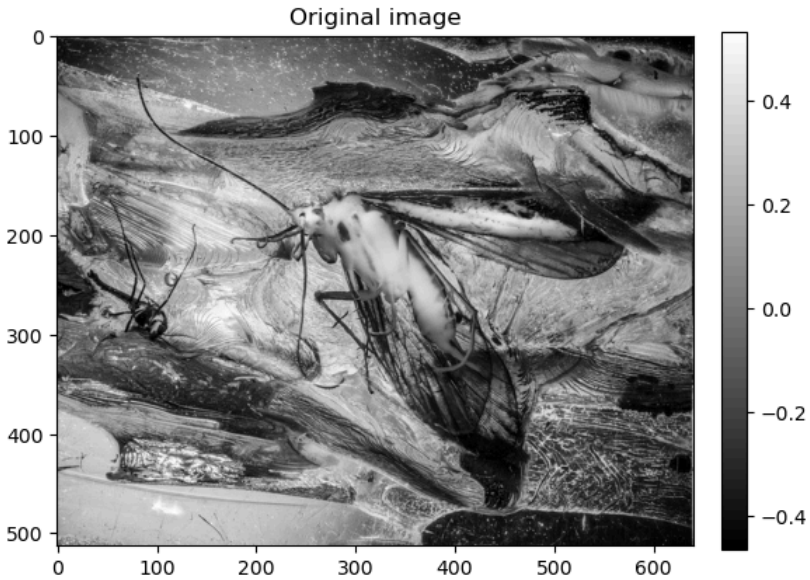
in which $N(x,y)$ is a noise image. For each pixel (x,y), make $N(x,y)$ a zero-mean random nubmer $-N_{max} \leq N(x,y) \leq N_{max}$ with a maximum magnitude N_{max}. Make N_{max} large enough so that the noise is obvious. Plot out this noisy image. You can see an example of such a noisy image in Figure 4A. Once you get this noisy image, repeat (D) to plot $|S(k)^2|$ versus k, and see whether $|S(k)^2|$ versus k behaves like the blue curve in Figure 4B, and try to understand the result.

In [17]:
```
def noise(nmax = 1, n = 1):
    return uniform.rvs(loc = -nmax, scale = nmax, size = n)
```

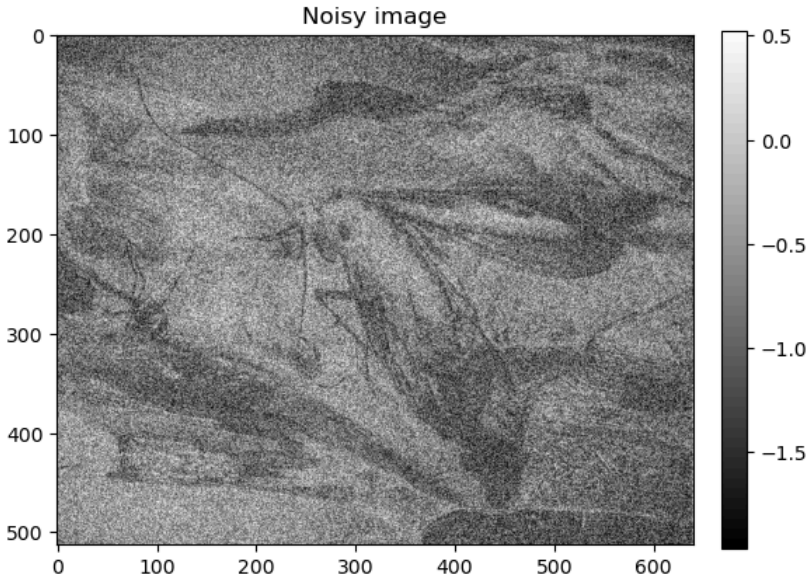
In [18]:
```
def noisy_image(original, nmax = 1):
    noise_data = np.reshape(noise(nmax, n = np.size(original)), newshape = original.shape)
    return original + noise_data
```

In [19]:
```
noisy_insect = noisy_image(insect, nmax = 1.5)
```

In [20]:
```
plt.title("Original image")
plot_image(insect)
```

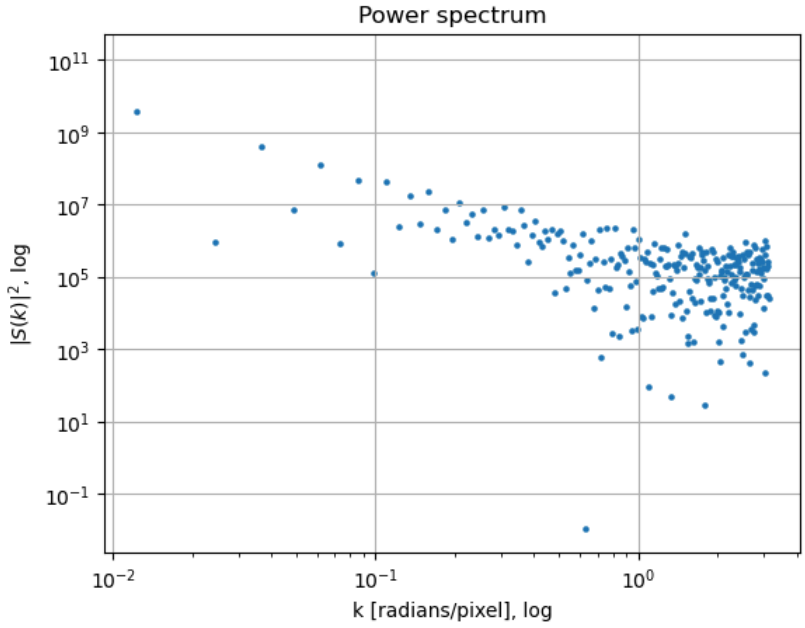


In [21]:
```
plt.title("Noisy image")
plot_image(noisy_insect)
```



In [22]:
```
ks = k_step(noisy_insect.shape[0])
sk = power_spectrum(noisy_insect, ks)
```

In [23]:
```
plt.scatter(ks, sk, s = 5)
plt.grid()
plt.xlabel("k [radians/pixel], log")
plt.ylabel(r"$|S(k)|^2$, log")
plt.xscale("log")
plt.yscale("log")
plt.title("Power spectrum")
plt.show()
```



(G) Gain control of the Fourier components

Going from the image S to its Fourier components $S(k)$ is the principal component transform on the image. For efficient coding, we need to give gain control $g(k)$ to each component $S(k)$ as the second step in the efficient coding recipe. This gives

$$O(k) = g(k)S(k)$$

This $g(k)$ should be

$$g(k) = (|k| + |k_o|) \times exp(-\frac{|k|^2}{k_{low}^2})$$

in which k_o and k_{low} are parameters. The first factor $(|k| + |k_o|)$ is a whitening filter, useful for low input noise conditions. Typically, k_o is very small, you could make $k_o = 0$ or just non-zero small number. The second factor $exp(-\frac{|k|^2}{k_{low}^2})$ is a low-pass low filter, to

avoid sending too much noise when signal-to-noise is too low for large $|k|$. Try to make k_{low} at a $|k|$ value where the signal and noise power are comparable, this is where $|S(k)|^2$ stop the trend of decaying with $|k|$.

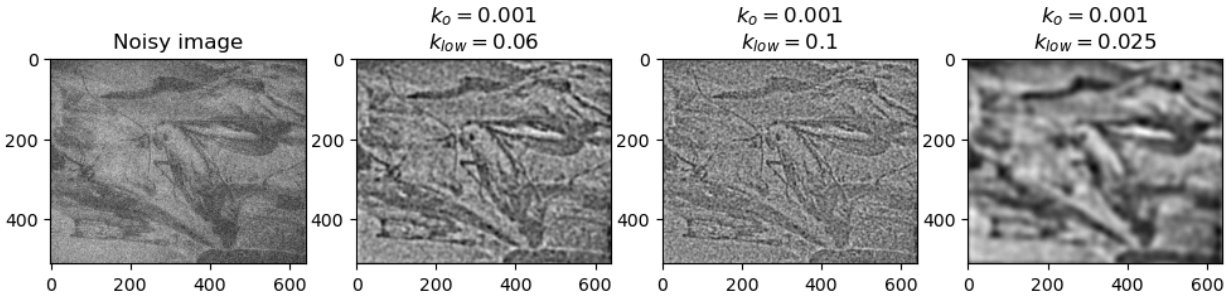
Then, you do the inverse Fourier transform on $O(k)$ and plot the results. Play with different values for k_{low} and see what you get as the results. Compare with the plots in Figure 4CDE.

In [24]:
```
def gain_filter(image_shape, k_o, k_low):
    height, width = image_shape
    freq_x, freq_y = np.meshgrid(np.fft.fftfreq(width), np.fft.fftfreq(height))
    frequency_magnitude = np.sqrt(freq_x**2 + freq_y**2)
    return (np.abs(frequency_magnitude) + np.abs(k_o)) * \
        np.exp(-frequency_magnitude**2 / k_low**2)
```

In [25]:
```
fourier_transform = np.fft.fft2(noisy_insect)
```

In [26]:
```
k_o = 1e-3
k_low_values = [0.06, 0.1, 0.025]
```

In [27]:
```
plt.figure(figsize = (12, 6))
plt.subplot(1, 4, 1)
plt.imshow(noisy_insect, cmap = "gray")
plt.title("Noisy image")

for i, k_low in enumerate(k_low_values):
    g_k = gain_filter(noisy_insect.shape, k_o, k_low)
    O_k = g_k * fourier_transform

    filtered_image = np.fft.ifft2(O_k)
    filtered_image = np.real(filtered_image)

    plt.subplot(1, 4, i + 2)
    plt.imshow(filtered_image, cmap = "gray")
    plt.title(r"$k_o = $" + f"{k_o}" + "\n" + r"$k_{low} = $" + f"{k_low}")

plt.show()
```



In [27]: