<< Back to MAIN notebook

5 Image filtering

The following subsections aim to describe various image filtering techniques which can be used to enhance images for velocimetry purposes. The order of their presentation will be from the implementationally simple to more complex, provided with reusable functions. For each individual filtering method, en example will be provided which illustrates its purpose, strengths and weaknesses.

All of the presented filtering methods are also available in the preprocessing tool **SSIMS** which has a user-friendly graphical interface (GUI) and can be downloaded and used free of charge.

Contents

In [1]:
```
# Necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import cv2
import random
import glob
import scipy.stats as stats

from skimage.metrics import structural_similarity as ssim
from axes_tiein import on_lims_change, cmap

# Set default colormap, defined in axes_tiein.py
plt.rcParams['image.cmap'] = cmap

# Use [%matplotlib widget] inside JupyterLab,
# and [%matplotlib notebook] for Jupyter Notebook
%matplotlib widget
```

5.1 Image negative

Negative of an image is obtained by "reversing" the intensity of its colors, which is often implemented by subtracting its pixel values from the maximal pixel intensity - usually 255 (for an 8bit image). The

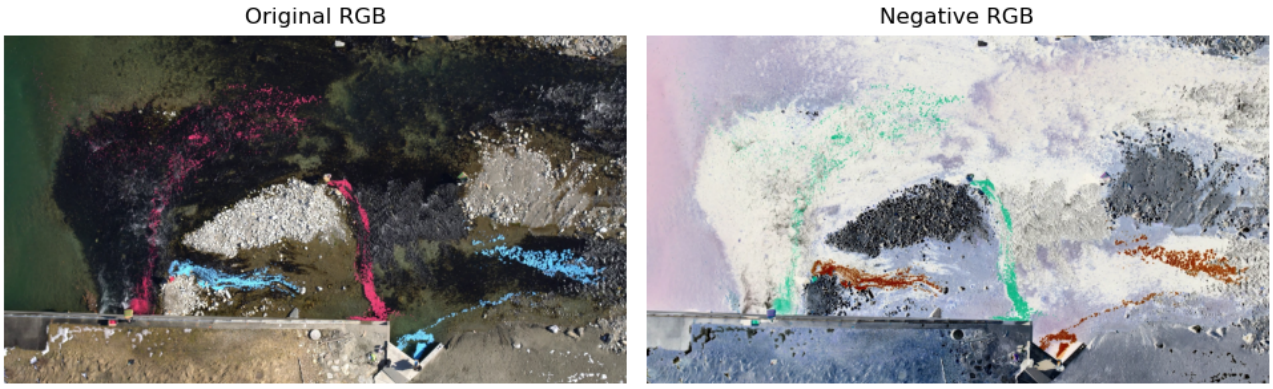
negative of an image can be obtained both for a single-channel and multi-channel image.

While the image negative itself does not enhance the image (image features are equally defined in both original and negative image) it can serve as a helper function for other filtering methods, as will be demonstrated later on.

> A more elegant way of producing an image negative is using negation (not) operator `~`.

In [2]:
```
img_path = './1080p/4.jpg'
img_bgr = cv2.imread(img_path)
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)

# Alternative cv2.subtract(img_bgr, 255) or (255 - img_bgr) for 8bit image
img_bgr_negative = ~img_bgr
img_rgb_negative = cv2.cvtColor(img_bgr_negative, cv2.COLOR_BGR2RGB)

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(9.8, 3.2))

# Don't forget to use RGB for matplotlib
ax[0].imshow(img_rgb)
ax[0].set_title('Original RGB')

ax[1].imshow(img_rgb_negative)
ax[1].set_title('Negative RGB')

[a.axis('off') for a in ax.reshape(-1)]
[a.callbacks.connect('ylim_changed', on_lims_change) for a in ax.reshape(-1)]

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

Figure



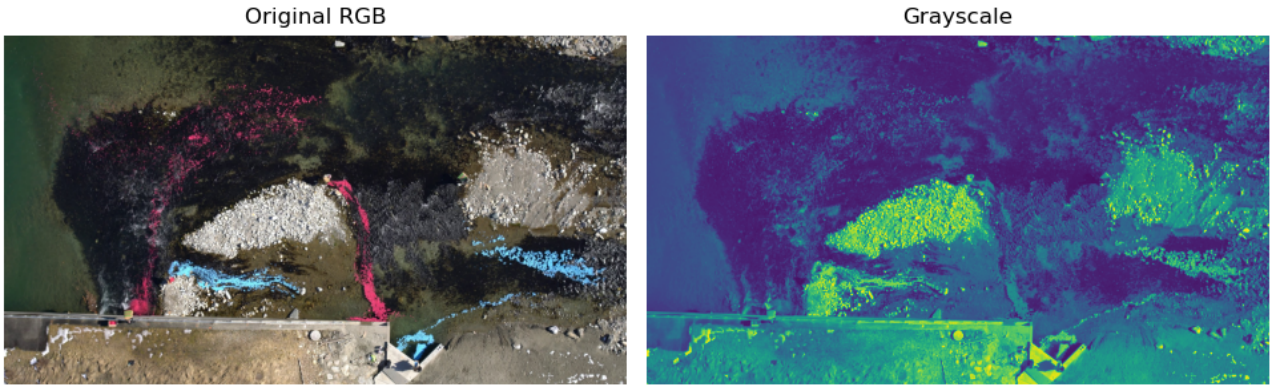
5.2 Conversion to grayscale

Grayscale colorspace has been discussed in the notebook Image formats. Here, we will just show the two methods of converting an image to grayscale colorspace.

In [3]:
```
# Converting to grayscale on image read using grayscale flag = 0
img_gray = cv2.imread(img_path, 0)
# Alternative to the above is loading BGR image and converting:
# img_gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)
```

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(9.8, 3.2))

ax[0].imshow(img_rgb)
ax[0].set_title('Original RGB')

ax[1].imshow(img_gray)
ax[1].set_title('Grayscale')

[a.axis('off') for a in ax.reshape(-1)]
[a.callbacks.connect('ylim_changed', on_lims_change) for a in ax.reshape(-1)]

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

Figure



5.3 Adjustment of brightness and contrast

In the example of **Image 4** we can notice that certain image areas, especially some regions of water surface are dark and there is not enough contrast between the surface and the tracer particles. UAV cameras, when set to record images/videos with automatic mode, will try to balance the image exposure (amount of light reaching the camera sensor), which can make certain regions too bright or too dark. In the example image, the islands and the riverbank are significantly brighter than the water surface, but for our velocimetry analysis we are only interested in the water surface and the tracer particles.

The simplest way to accentuate the tracer particles in such cases is to adjust brightness and contrast using a linear transformation:

$$Result = \alpha Original + \beta,$$

where α and β are coefficients. Using $\alpha = 1$ and $\beta = 0$ will not affect the original image.

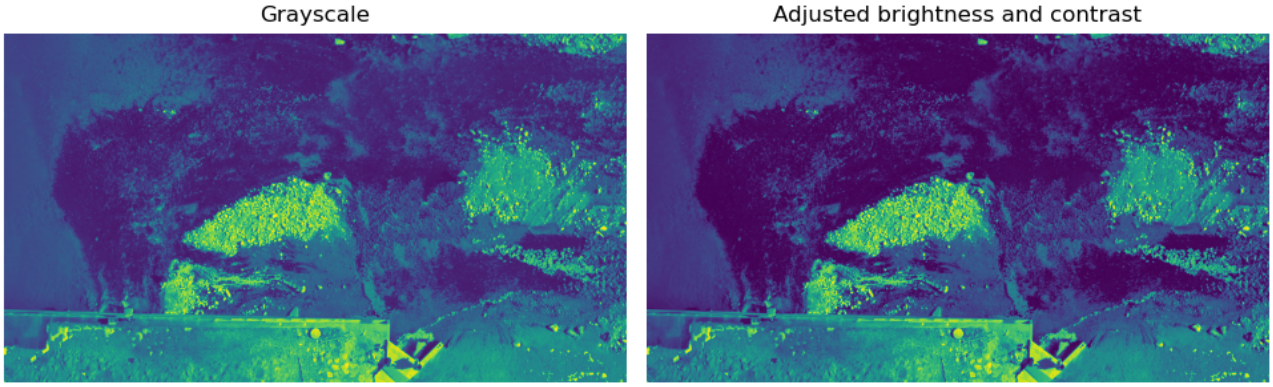
> Instead of performing the transformation directly, it is advised to use the inbuilt OpenCV finction `cv2.convertScaleAbs()` to avoid overflow problems.

We can test this filtering method using the grayscale image presented in the previous section:

In [4]:
```
# Parameters of the method
alpha = 0.8
beta = -20

# Linear transformation
img_bc_adj = cv2.convertScaleAbs(img_gray, alpha=alpha, beta=beta)
```

```
fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(9.8, 3.2))

ax[0].imshow(img_gray)
ax[0].set_title('Grayscale')

ax[1].imshow(img_bc_adj)
ax[1].set_title('Adjusted brightness and contrast')

[a.axis('off') for a in ax.reshape(-1)]
[a.callbacks.connect('ylim_changed', on_lims_change) for a in ax.reshape(-1)]

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

Figure



5.5 Gamma adjustment

Adjustment of brightness and contrast indiscriminately changes the value of each image pixel, i.e., the same transformation is applied to each pixel. Other transformation methods might prove more effective in certain cases. For example, Gamma adjustment involves the use of an exponential function, to target the contrast change in low or high pixel values in the image (where Y_{max} is the maximal pixel value, usually 255 for an 8bit image):

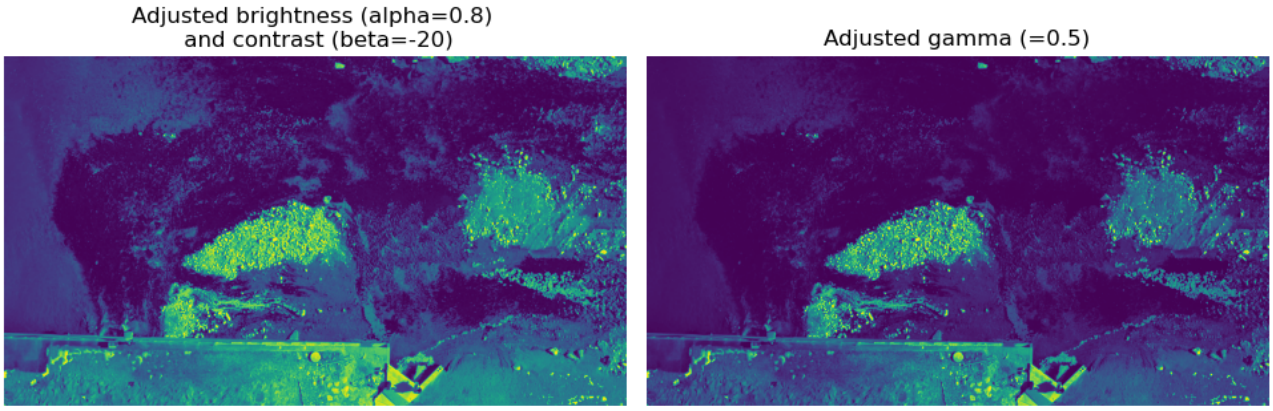
$$Result = (Original/Y_{max})^{1/\gamma} \times Y_{max},$$

where γ coefficient defines whether the image is transformed to increase the contrast in the low-end ($\gamma > 1.0$) or high-end ($\gamma < 1.0$) of image pixel values:

In [5]:
```
def gamma_adjustment(img, gamma=1.0):
    # Create lookup table
    table = np.array([((i / 255.0) ** (1/gamma)) * 255 for i in np.arange(0, 256)]).astype("u
    # Use lookup table to transform image
    return cv2.LUT(img, table)


# Parameter of the method
gamma = 0.5
# Perform transformation
img_gamma = gamma_adjustment(img_gray, gamma=gamma)

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(9.8, 3.4))
```

```
ax[0].imshow(img_bc_adj)
ax[0].set_title('Adjusted brightness (alpha={:.1f}) \n and contrast (beta={})'.format(alpha,

ax[1].imshow(img_gamma)
ax[1].set_title('Adjusted gamma (={})'.format(gamma))

[a.axis('off') for a in ax.reshape(-1)]
[a.callbacks.connect('ylim_changed', on_lims_change) for a in ax.reshape(-1)]

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

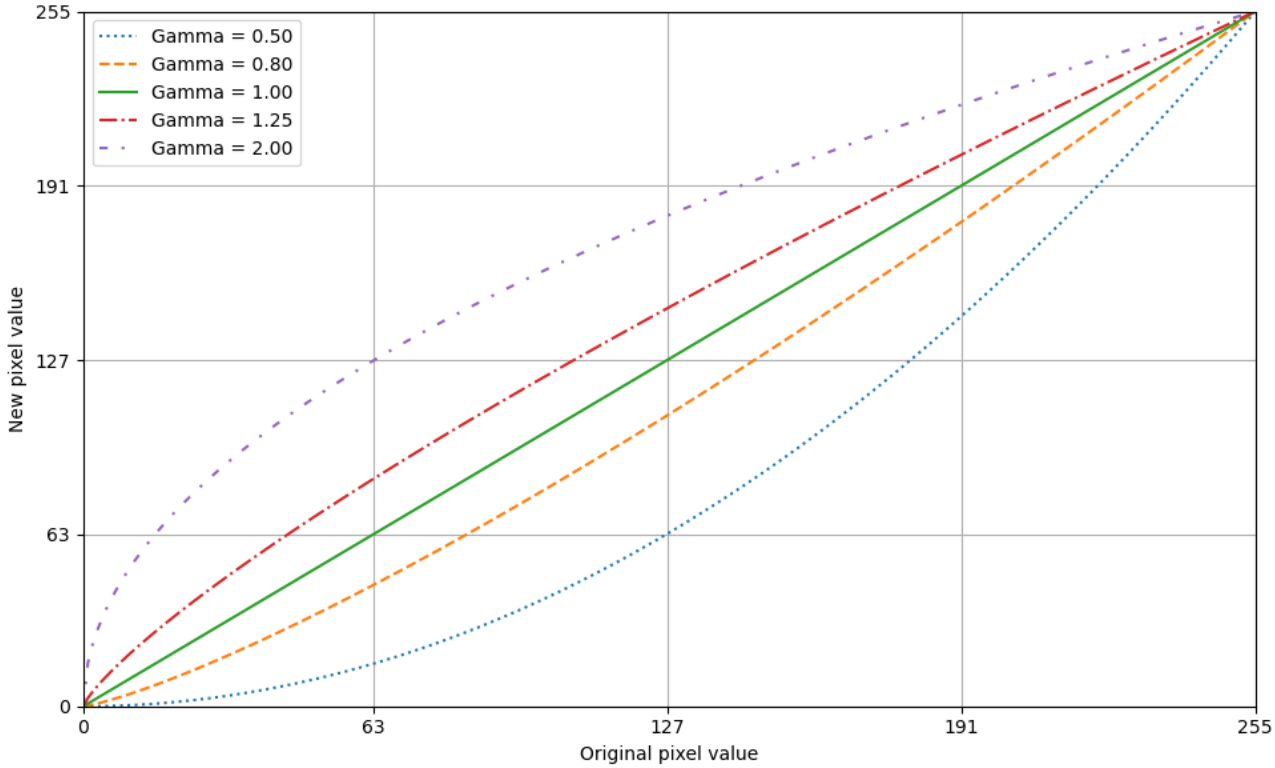
Figure



The effect of Gamma adjustment can be explained graphically with the following plot:

In [6]:
```
gamma_list = [0.5, 0.8, 1.0, 1.25, 2.0]
ls = [':', '--', '-', '-.', (0, (3, 5, 1, 5))]

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(9.8, 6))

for i, g in enumerate(gamma_list):
    table = [((i / 255.0) ** (1/g)) * 255 for i in np.arange(0, 256)]
    ax.plot(table, linestyle=ls[i])

ax.set_xlabel('Original pixel value')
ax.set_ylabel('New pixel value')
ax.set_xlim(0, 255)
ax.set_ylim(0, 255)
ax.set_xticks([0, 63, 127, 191, 255])
ax.set_yticks(ax.get_xticks())
ax.legend(['Gamma = {:.2f}'.format(g) for g in gamma_list])
ax.grid()

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

Figure



Brightness, contrast and gamma adjustment can also be combined in a single expression (for any image depth, where Y_{max} is the maximal pixel value):

$$Result = \alpha \times \left(\frac{Original}{Y_{max}} \right)^{1/\gamma} \times Y_{max} + \beta,$$

or

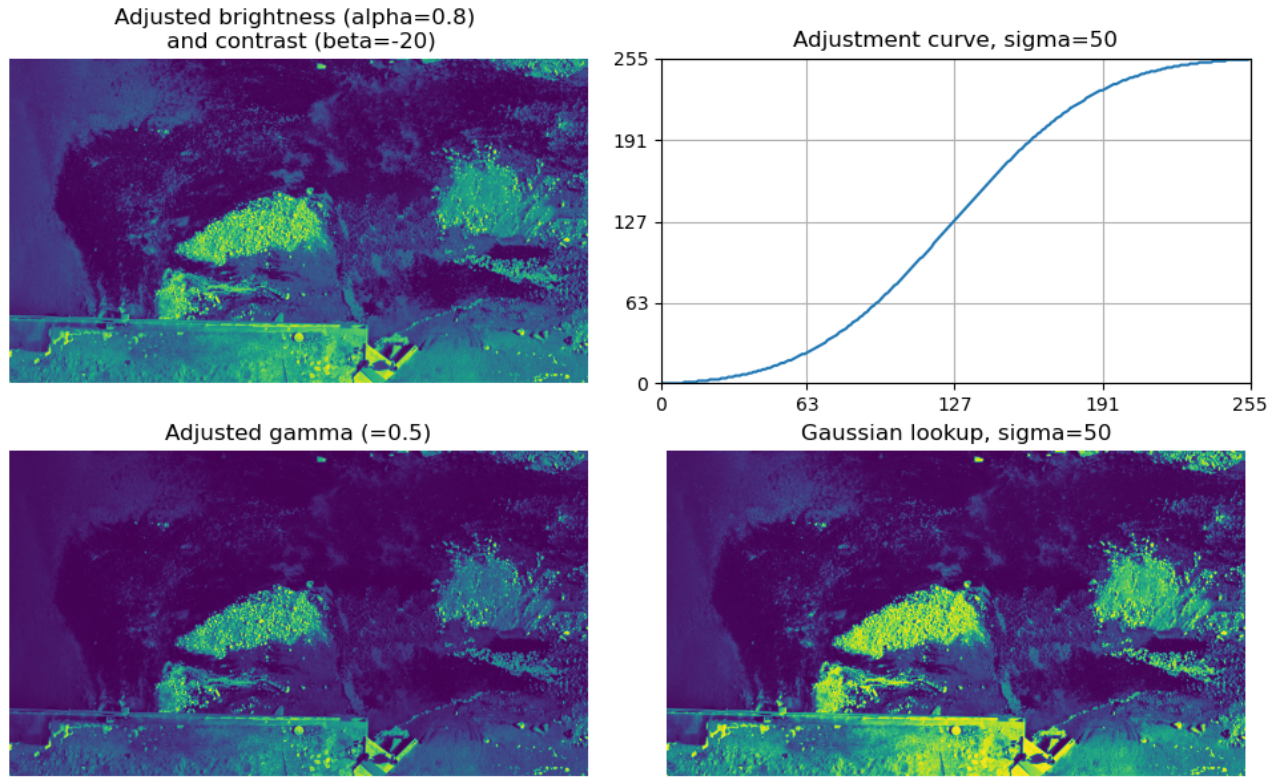
$$Result = \left(\frac{\alpha \times Original + \beta}{Y_{max}} \right)^{1/\gamma} \times Y_{max}.$$

Now that the basic image transformations have been presented, we can even go ahead and create our own transformation expressions. For example, we can use a Gaussian (normal) distribution function as a lookup table to increase image contrast. The idea behind such approach is to stretch the pixel intensities in the middle of the range and consequently between lighter tracer particles and the darker background.

In [7]:
```
def gaussian_lookup(img, sigma):
    # Original values
    x = np.arange(0, 256)
    # Probability density function (PDF)
    pdf = stats.norm.pdf(x, 127, sigma)
    # Cumulative distribution function (CDF)
    cdf = np.cumsum(pdf)
    # Normalize CDF to range 0-255
    cdf_norm = np.array([(x - np.min(cdf))/(np.max(cdf) - np.min(cdf)) * 255 for x in cdf]).a

    # x and cdf_norm returned just for the sake of
    # plotting them later, can be removed
    return x, cdf_norm, cv2.LUT(img, cdf_norm)
```

```
# Parameter of the method
sigma = 50

# Perform lookup using Gaussian CDF
x, cdf, img_gauss = gaussian_lookup(img_gray, sigma)

fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(9.8, 6))

ax[0][0].imshow(img_bc_adj)
ax[0][0].set_title('Adjusted brightness (alpha={:.1f}) \n and contrast (beta={})'.format(alph

ax[0][1].plot(x, cdf)
ax[0][1].set_title('Adjustment curve, sigma={}'.format(sigma))
ax[0][1].set_xlim(0, 255)
ax[0][1].set_ylim(0, 255)
ax[0][1].set_xticks([0, 63, 127, 191, 255])
ax[0][1].set_yticks(ax[0][1].get_xticks())
ax[0][1].grid()

ax[1][0].imshow(img_gamma)
ax[1][0].set_title('Adjusted gamma (={})'.format(gamma))

ax[1][1].imshow(img_gauss)
ax[1][1].set_title('Gaussian lookup, sigma={}'.format(sigma))

[a.axis('off') for a in ax.reshape(-1)]
ax[0][1].axis('on')
[a.callbacks.connect('ylim_changed', on_lims_change) for a in ax.reshape(-1)]

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

Figure



If we compare the image with adjusted brightness/contrast, the gamma adjusted image, and the Gaussian lookup method, we can see that in certain areas - such as around (800, 220) - the tracer

particles are far more pronounced relative to the background than in any of the previous methods. We can vary the `sigma` parameter (standard deviation of the Gaussian distribution) to achieve suitable results. Reducing the `sigma` will "tighten" the results, i.e., the darker areas will become more darker and vice verse; increasing the `sigma` will have the opposite effect.

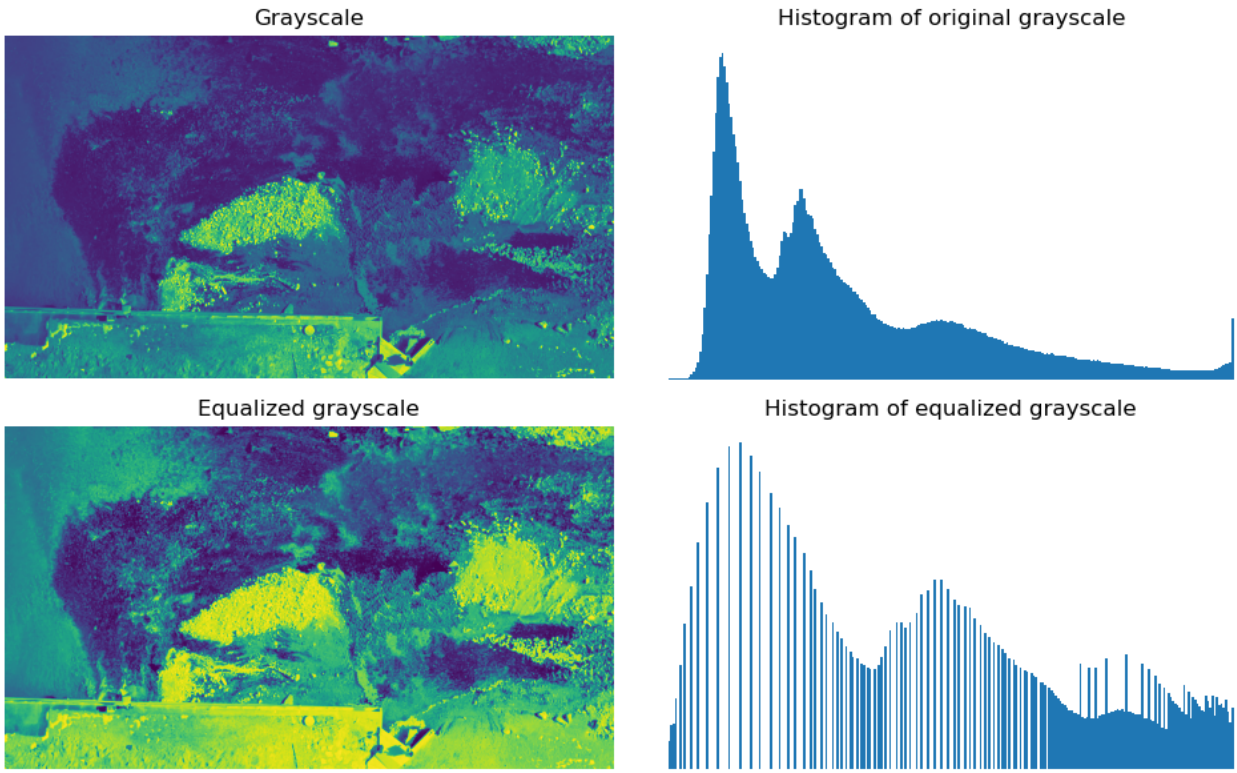
5.5 Histogram equalization

Another method for increasing image contrast is based on the manipulation of image histogram. For a single-channel image, such as a grayscale image, histogram counts occurrences of the individual pixel values, and, in a sense, is similar to the probability density function. Examining image histogram is often useful because it reveals which pixel (tonal) values are more represented in the image, and which ones are less.

A method called the histogram equalization aims to "flatten" the image histogram which in turn increases the overall image contrast:

In [8]:
```
img_histeq = cv2.equalizeHist(img_gray)

fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(9.8, 6))

ax[0][0].imshow(img_gray)
ax[0][0].set_title('Grayscale')

ax[1][0].imshow(img_histeq)
ax[1][0].set_title('Equalized grayscale')

# Flatten the image to 1D array using .reshape(-1) and create a histogram of pixel values
ax[0][1].hist(img_gray.reshape(-1), 256, [0, 256])
ax[0][1].set_title('Histogram of original grayscale')

ax[1][1].hist(img_histeq.reshape(-1), 256, [0, 256])
ax[1][1].set_title('Histogram of equalized grayscale')

[a.axis('off') for a in ax.reshape(-1)]
[ax[i][0].callbacks.connect('ylim_changed', on_lims_change) for i in range(2)]

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

Figure



As evident from the histograms of original and equalized images, the resulting image has a more even tonal distribution (more contrast) but some values in the histogram are likely to be missing after the the histogram equalization process. An important downside of this method is that it indiscriminately analyses the entire image, which usually contains areas that will not be targeted by the image velocimetry, but which will affect the histogram equalization process. Potential way of dealing with this issue is to mask out regions which will be of no interest to us later on.

Presented method is **RARELY RECOMMENDED** because of its indiscriminate approach. In vase majority of cases, the method is likely to amplify undesirable image features - riverbed, reflections, color differences, etc. There exists an extension of this method, called "Contrast-Limited Adaptive Histogram Equalization" (CLAHE), which will also be explored in the following section.

5.6 Contrast-limited adaptive histogram equalization (CLAHE)

In order to improve the previous method, we can do two thing:

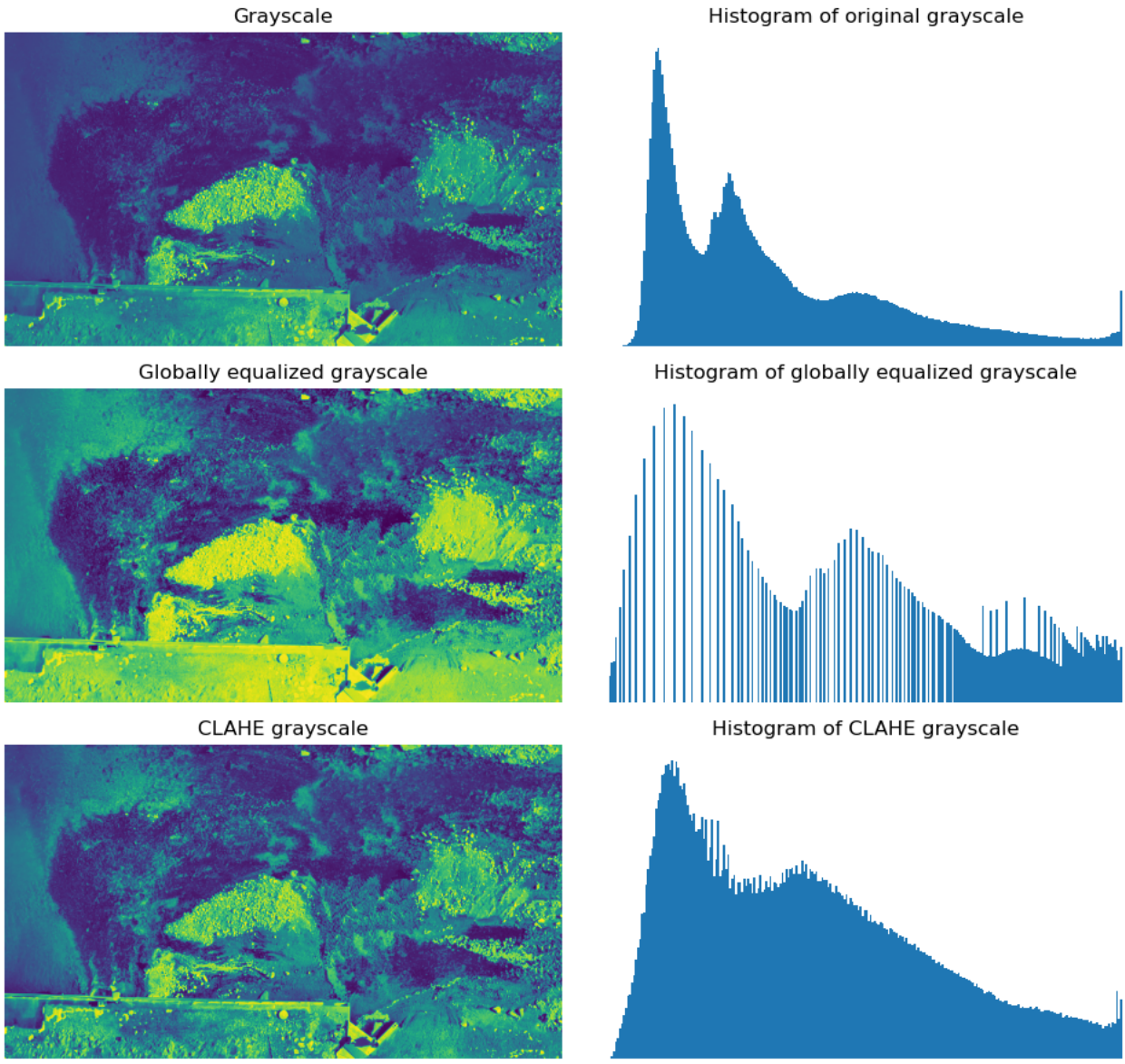
1. Instead of equalizing the histogram of an entire image, we can traverse the image using a smaller window (tile) and equalize the histogram in each window separately, which would increase the local contrast and allow for easier detection of features inside.
2. Limit the image contrast range by removing the peaks of the histogram.

The method that offers such improvements is called Contrast-Limited Adaptive Histogram Equalization (CLAHE). Unlike global equalization approach, CLAHE operates locally and the resulting histogram is clipped as not to oversaturate the image with certain pixel values. As such, CLAHE requires two parameters - `clip` value and `tile` size. Often cited range of usable `clip` values is between 2 and 5,

while the window/ `tile` size depends mostly on the size of the features we wish to accentuate. A good starting point for `tile` size is two-to-three times the average size of tracer particles.

In [9]:
```
# Tile = size of the adaptive window
# Clip = contrast limit (clip) withing a window
def clahe(img, clip, tile):
    clahe = cv2.createCLAHE(clipLimit=clip, tileGridSize=(int(tile), int(tile)))
    return clahe.apply(cv2.cvtColor(img, cv2.COLOR_BGR2GRAY))

# Parameters of the method
clip = 2.0
tile = 8

# Perform CLAHE
img_clahe = clahe(img_bgr, clip=clip, tile=tile)

fig, ax = plt.subplots(nrows=3, ncols=2, figsize=(9.8, 9))

ax[0][0].imshow(img_gray)
ax[0][0].set_title('Grayscale')

ax[1][0].imshow(img_histeq)
ax[1][0].set_title('Globally equalized grayscale')

ax[2][0].imshow(img_clahe)
ax[2][0].set_title('CLAHE grayscale')

ax[0][1].hist(img_gray.reshape(-1), 256, [0, 256])
ax[0][1].set_title('Histogram of original grayscale')

ax[1][1].hist(img_histeq.reshape(-1), 256, [0, 256])
ax[1][1].set_title('Histogram of globally equalized grayscale')

ax[2][1].hist(img_clahe.reshape(-1), 256, [0, 256])
ax[2][1].set_title('Histogram of CLAHE grayscale')

[a.axis('off') for a in ax.reshape(-1)]
[ax[i][0].callbacks.connect('ylim_changed', on_lims_change) for i in range(3)]

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

Figure



The advantage of CLAHE is visible in the figure above, as the resulting histogram is more evenly distributed than the original and there are no missing pixel values. Like with global histogram equalization, some cases are quite "resistant" to such enhancement, and the results can be worse than the original images. An example of this is **Image 5** which has a rather "centralized" histogram because of which the contrast cannot be increased without simultaneously accentuating other undesired features.

As with the global histogram equalization, this filter has proven to rarely produce desirable results.

5.7 Highpass filter

In signal processing there is often the need for filtering out undesired frequencies. Such filters can be done in either spatial of frequency domain, but the general idea is to remove frequencies above or below a certain threshold - high or low frequencies - using so-called **highpass** and **lowpass** filters. For image data, high frequencies are presented by "rapid" changes in pixel intensity, often associated with image noise, while low frequencies are "slower" visual changes. If our goal is to accentuate tracer particles, we can look into the workings of the **highpass** filter.

> We have all likely encountered highpass and lowpass filters in our everyday lives - sound reproduction devices use hardware based filters called **audio crossover circuits** which split the audio signal into low frequencies (which are sent to the bass speakers), mid frequencies (which are usually sent to the driver speakers), and high frequencies (which are sent to so-called tweeters).

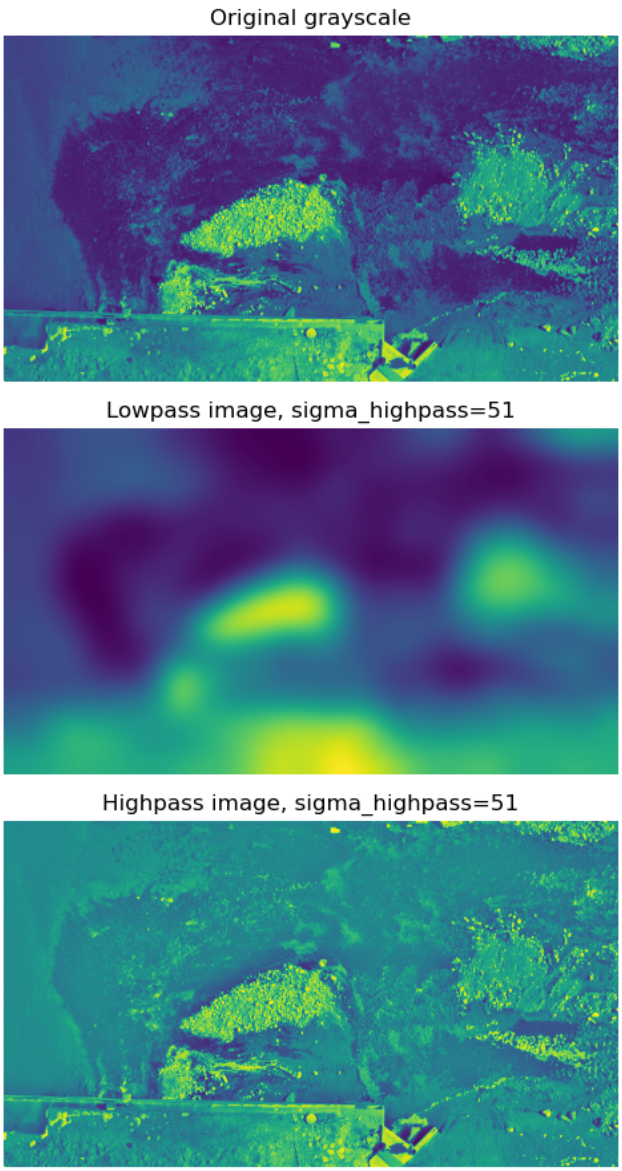
The problem with software implementations of highpass filters is that the frequency domain of a signal is always unbound on the right side towards $+\infty$ values, but limited on the left side with 0 - the range of frequencies we wish to "pass" is therefore unbound. A neat way of surpassing this limitation is to identify low frequencies and simply subtract those from the original signal - which leaves us with the high frequency content:

$$High\_frequency\_content = Original - Low\_frequency\_content.$$

The simplest way of applying a lowpass filter is by using some sort of an image blurring algorithm such as Gaussian blur. The method is then performed in spatial domain, and only has a single parameter - the blur filter strength described by the `sigma_highpass`:

In [10]:
```
# Function returns both low and high frequencies
def lowpass_highpass(img, sigma):
    # Make sure :sigma: is an odd number
    if sigma % 2 == 1:
        sigma += 1

    # Gaussian kernel size auto computed from :sigma:
    blur = cv2.GaussianBlur(img, (0, 0), int(sigma))

    return blur, ~cv2.subtract(cv2.add(blur, 127), img)

# Parameter of the method
sigma_highpass = 51

# Get both lowpassed and highpassed data
lowpass, highpass = lowpass_highpass(img_gray, sigma=sigma_highpass)

fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(9.8, 9))

ax[0].imshow(img_gray)
ax[0].set_title('Original grayscale')

ax[1].imshow(lowpass)
ax[1].set_title('Lowpass image, sigma_highpass={}'.format(sigma_highpass))

ax[2].imshow(highpass)
ax[2].set_title('Highpass image, sigma_highpass={}'.format(sigma_highpass))

[a.axis('off') for a in ax.reshape(-1)]
[a.callbacks.connect('ylim_changed', on_lims_change) for a in ax.reshape(-1)]

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

Figure



In the figure (with **Image 4**) above we can identify large image features in the lowpass image - riverbank, islands, changes is riverbed material, etc. The resulting highpass image is more-or-less devoid of such features which leaves it with only smaller features such as tracer particles, but also some amount of light reflections, waves, and other local features. There are no general recommendations on the size of the `sigma`, and has to be tailored to the specific case, but it's generally related to the size of the low frequency features we wish to remove - the larger the feature, the higher `sigma` value should be.

> The highpass filter implemented above will work for both singlechannel and multichannel images.

5.8 Intensity capping

If the tracer particles are consistently lighter or brighter than the water surface, we can make the background more uniform by "capping" (limiting) its tonal range. This can be done by replacing the pixel values Y with a capping value C:

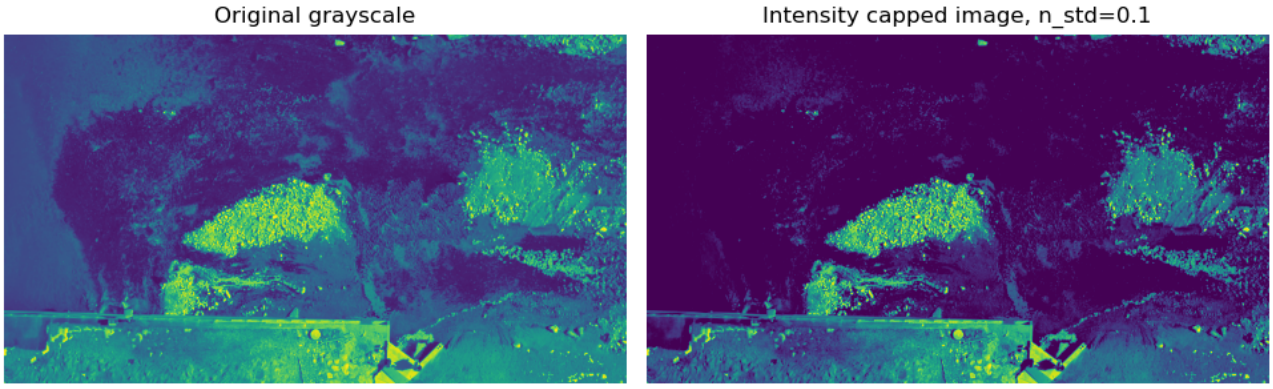
$$Y = \begin{cases} C = N_{cap} \times Y_{median} & \text{if } Y > C \\ Y & \text{else,} \end{cases}$$

where N_{cap} defines the capping value as the number of standard deviations of image pixel values from the median, and Y_{median} is the median image pixel value. Keep in mind that the equation above limits pixel values higher than a threshold, i.e., it is implicitly assumed that the tracer particles are darker than the background. If this is not the case, we can still use the same equation if we first convert the image to its negative. Additionally, the N_{cap} does not have to be an integer or even a positive number:

In [11]:
```
# Helper function which "stretches" the pixel values between given limits,
# while lower and higher pixel values are clipped to lower and upper limit
def normalize_image(img, lower=None, upper=None):
    if lower is None:
        lower = np.min(img)
    if upper is None:
        upper = np.max(img)

    img_c = img.astype(int)

    img_c = ((img_c - np.min(img_c)) / (np.max(img_c) - np.min(img_c)) * 255).astype('uint8')

    return img_c


# Mode:
#     LoD = light tracers on darker background
#     DoL = darl tracers on lighter background
# Set :manual: if you want to define the capping value
def intensity_capping(img, n_std, mode='LoD', manual=None):
    assert mode in ['LoD', 'DoL']
    img_c = ~img.copy() if mode == 'LoD' else img.copy()

    if not manual:
        median = np.median(img_c)
        stdev = np.std(img_c)
        cap = median - n_std * stdev
    else:
        assert type(manual) in [int, float]
        cap = manual

    img_c[img_c > cap] = cap
    # Stretch the pixel values between 0 and 255
    img_c = normalize_image(img_c, cap, np.max(img_c))

    if mode == 'LoD':
        img_c = ~img_c

    return img_c


# Parameter of the method
n_cap = 0.1

# Set :mode='LoD': to indicate Light (particles) on Dark (surface)
# You can manually set the capping value with parameter :manual:
img_capped = intensity_capping(img_gray, n_std=n_cap, mode='LoD')

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(9.8, 3.2))
```

```
ax[0].imshow(img_gray)
ax[0].set_title('Original grayscale')

ax[1].imshow(img_capped)
ax[1].set_title('Intensity capped image, n_std={:.1f}'.format(n_cap))

[a.axis('off') for a in ax.reshape(-1)]
[a.callbacks.connect('ylim_changed', on_lims_change) for a in ax.reshape(-1)]

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

Figure



The resulting image has a far more uniform background while the light-colored tracer particles are not affected. A good starting point for finding the optimal capping value is by starting from `n_cap=0.0` and varying it in steps of +/- 0.1.

5.9 Denoising

In the section about the highpass filter we've discussed how the Gaussian filter can remove high frequency image content. Depending on the size of the Gaussian kernel and its strength (`sigma`), we can control which high frequencies are filtered - if we're not careful we will also filter out our tracer particles. The Gaussian blur method is one of many ways of eliminating "image noise", and there are some far more sophisticated.

Why should we even opt for removing high frequency content from an image? Well, this is only useful in a handful of cases. One of such is when the UAV camera was forced to operate with a high ISO value (high sensor sensitivity) in low light conditions (recording in the morning or at dusk), which can produce granular visual noise - often referred to as *salt-and-pepper* noise.

In absence of such example images, we can create our own artificial noisy images:

In [12]:
```
# A function to generate "noise" in image
def sp_noise(img, prob):
    output = np.zeros(img.shape[:2], dtype='uint8')
    thr = 1 - prob

    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            rdn = random.random()
```

```
            if rdn < prob:
                output[i][j] = random.randint(0, 127)
            elif rdn > thr:
                output[i][j] = random.randint(127, 255)
            else:
                output[i][j] = img[i][j]

    return output

img_sp = cv2.imread('./1080p/1.jpg', 0)
noise_prob_list = [0, 0.05, 0.1, 0.2]

fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(9.8, 6))

for i, prob in enumerate(noise_prob_list):
    ax[i//2][i%2].imshow(sp_noise(img_sp, prob))
    ax[i//2][i%2].set_title('Noise level = {:.2f}'.format(prob))

[a.axis('off') for a in ax.reshape(-1)]
[a.callbacks.connect('ylim_changed', on_lims_change) for a in ax.reshape(-1)]

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

Figure

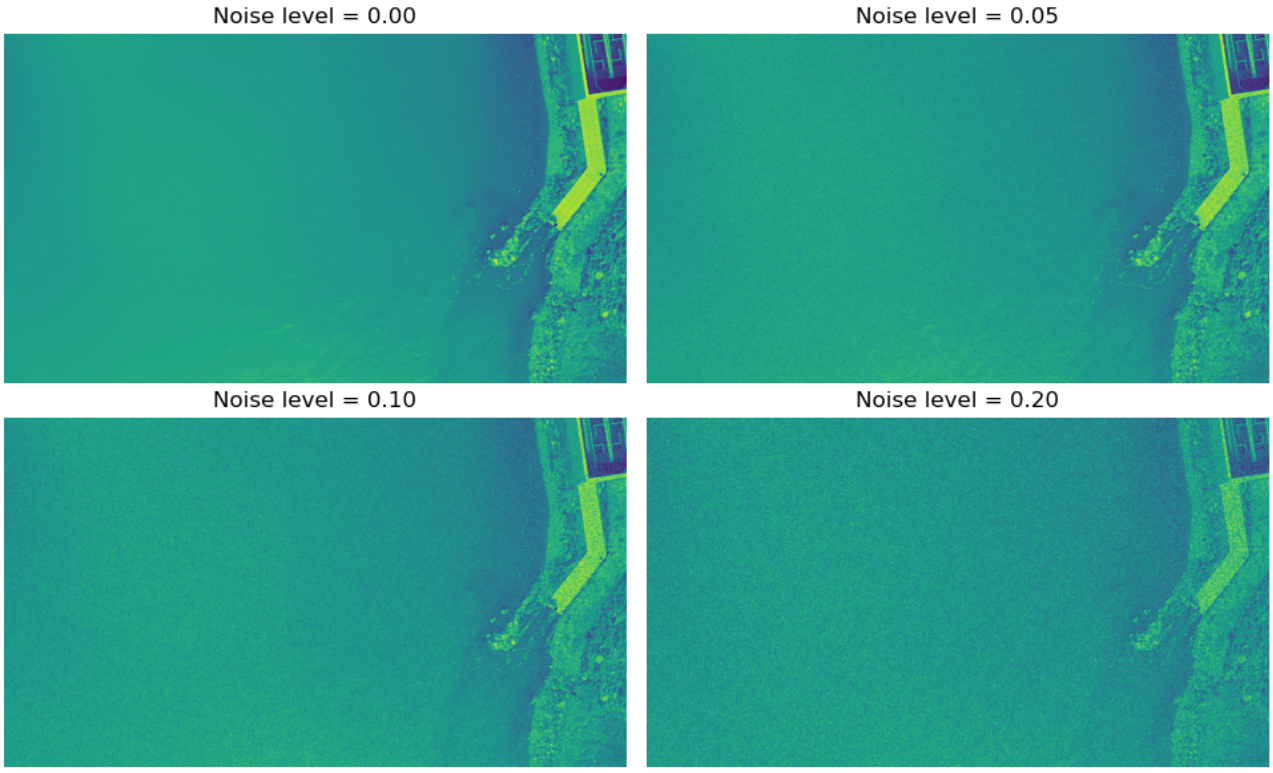


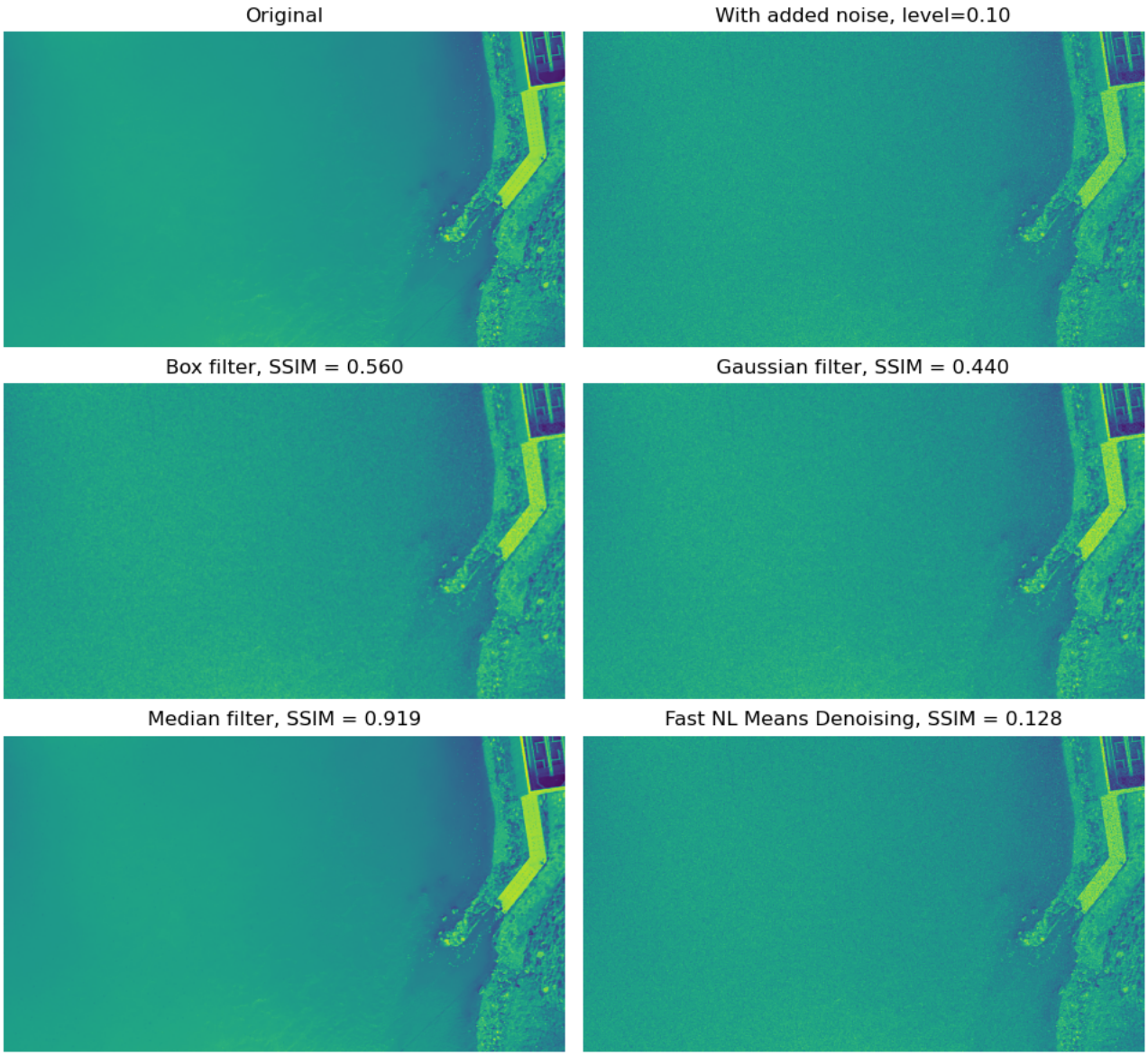
Figure above is meant to simulate what high camera ISO value (overly sensitive camera sensor) does to captured images. Now we can explore several noise removal methods:

1. **Box filter**, which uses a unitary kernel (filled with ones),
2. **Gaussian filter** which uses a kernel based on 2D Gaussian distribution,
3. **Median filter** which replaces the pixel value with the median of its neighborhood, and
4. **Fast non-local means denoising** method, implemented in OpenCV.

In order to test different denoising methods, we can use one of the images from the previous figure. Testing the effectiveness of the methods will be done using Structural Similarity Index (SSIM) from the `skimage` library, which can compare the post-denoising image with the original image without artificial noise (first image in the figure above):

In [13]:
```
# Noise level = 0.10
img_sp_ex = sp_noise(img_sp, 0.1)

# Apply filters
img_dn_box = cv2.boxFilter(img_sp_ex, cv2.CV_8U, (5, 5))
img_dn_gauss = cv2.GaussianBlur(img_sp_ex, (0, 0), 1)
img_dn_median = cv2.medianBlur(img_sp_ex, 3)
img_dn_fastnl = cv2.fastNlMeansDenoising(img_sp_ex, None, 15, 15, 45)

# Calculate SSIM scores
ssim_box = ssim(img_sp, img_dn_box, data_range=img_dn_box.max() - img_dn_box.min())
ssim_gauss = ssim(img_sp, img_dn_gauss, data_range=img_dn_gauss.max() - img_dn_gauss.min())
ssim_median = ssim(img_sp, img_dn_median, data_range=img_dn_median.max() - img_dn_median.min(
ssim_fastnl = ssim(img_sp, img_dn_fastnl, data_range=img_dn_fastnl.max() - img_dn_fastnl.min(

fig, ax = plt.subplots(nrows=3, ncols=2, figsize=(9.8, 9))

ax[0][0].imshow(img_sp)
ax[0][0].set_title('Original')

ax[0][1].imshow(img_sp_ex)
ax[0][1].set_title('With added noise, level=0.10')

ax[1][0].imshow(img_dn_box)
ax[1][0].set_title('Box filter, SSIM = {:.3f}'.format(ssim_box))

ax[1][1].imshow(img_dn_gauss)
ax[1][1].set_title('Gaussian filter, SSIM = {:.3f}'.format(ssim_gauss))

ax[2][0].imshow(img_dn_median)
ax[2][0].set_title('Median filter, SSIM = {:.3f}'.format(ssim_median))

ax[2][1].imshow(img_dn_fastnl)
ax[2][1].set_title('Fast NL Means Denoising, SSIM = {:.3f}'.format(ssim_fastnl))

[a.axis('off') for a in ax.reshape(-1)]
[a.callbacks.connect('ylim_changed', on_lims_change) for a in ax.reshape(-1)]

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

Figure



The ideal SSIM score, which would indicate identical images, equals 1. From the results in the previous figure, the best filtering method for salt-and-pepper noise is the median filter. By inspecting the tracers near the riverbank, it becomes clear that no denoising method, except the median filter, was able to remove noise without sacrificing the tracer particles as well. The most sophisticated of the analyzed solutions - Fast NL Means Denoising method, which has also proven to be quite slow - had the lowest SSIM score. This goes to show that the simplest of solutions are sometimes the most effective.

5.10 Removal of image background

Proper and continuous identification of tracer particles is a crucial step in the image velocimetry workflow, as they are the carriers of the information about surface flow velocities. In an ideal case, everything but the tracer particles would remain stationary and not affect the final results. All of the image filtering methods presented so far have been focused on helping us identify the movable parts the image - the image foreground.

An alternative approach, instead, could be to first identify the static, non-movable parts of the image (background) and identify the tracer particles (foreground) as a difference between the given image and

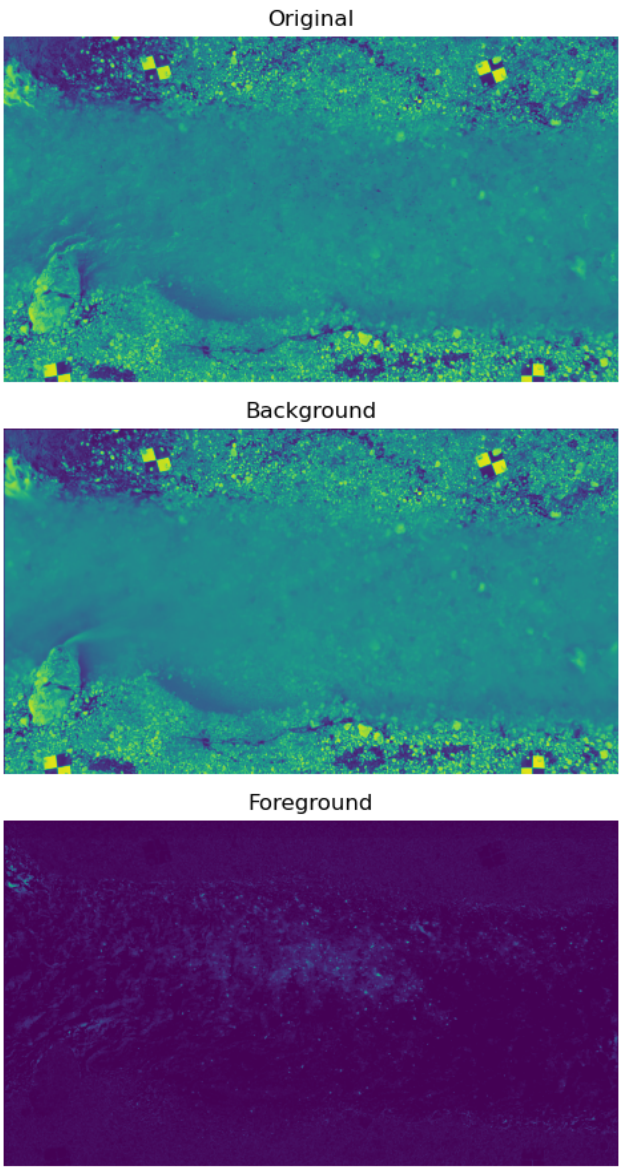
the obtained background. **In order for this to be possible, objects that are static in the real world have to be static in the images, i.e., the image stabilization has to be completed before this procedure.**

We can determine the image background using median values of individual pixels in a sequence of stabilized images. For multichannel images, this process can be repeated for each channel.

In [14]:
```
# Number of frames to estimate background
num_frames_background = 100

# This block will work if :img_folder: is provided
try:
    img_folder = r'Path to frames\' folder'
    img_path_list = glob.glob('{}/*.jpg'.format(img_folder))
    num_frames_total = len(img_path_list)

    height, width = cv2.imread(img_path_list[0], 0).shape

    # Don't use only the images from the beginning of the sequence,
    # but distribute the sample evenly across the entire sequence
    frame_step = num_frames_total // num_frames_background

    # Start stacking images one on top of the other
    stack = np.ndarray([height, width, num_frames_background], dtype='uint8')

    # Stack the individual frames one on to of the other
    for i in range(num_frames_background):
        stack[:, :, i] = cv2.imread(img_path_list[i*frame_step], 0)

    # Go through axis 2 and find the median pixel values
    background = np.median(stack, axis=2).astype('uint8')

    # Uncomment if you want to save the background to file
    cv2.imwrite('./background.jpg', background)

    # Calculate the foreground
    original = cv2.imread(img_path_list[0], 0)
    foreground = cv2.subtract(background, original)

# If :img_folder: is NOT provided, try to read from file
except IndexError:
    # Load background from file
    background = cv2.imread('./background.jpg', 0)

    # Calculate the foreground
    original = cv2.imread('./1080p/5.jpg', 0)
    foreground = cv2.subtract(background, original)

fig, ax = plt.subplots(nrows=3, ncols=1, figsize=(9.8, 9.0))

ax[0].imshow(original)
ax[0].set_title('Original')

ax[1].imshow(background)
ax[1].set_title('Background')

ax[2].imshow(foreground)
ax[2].set_title('Foreground')
```

```
[a.axis('off') for a in ax.reshape(-1)]
[a.callbacks.connect('ylim_changed', on_lims_change) for a in ax.reshape(-1)]

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

Figure



Don't forget to zoom in on different regions using interactive controls and inspect the details.

5.11 Conclusions on image filtering

This notebook provides an overview of different image filtering techniques which can potentially be applied for enhancement of UAV images for velocimetry purposes. However, not all of the presented methods will prove to be useful, or even adequate. For that reason, the effort of this report is also directed towards "what not to do", and to provide an overview of common practical pitfalls.

In order to fully explore the potential of different filtering methods, we will explore them in combination with different colorspace models and their channels, and in combination with other filtering methods, i.e., creating stacks of different filters in order to obtain optimal results. These procedures will be explained in the following notebook.

Continue to next chapter: Filter stacking >>

or

<< Back to MAIN notebook