Overview of image enhancement techniques for UAV image velocimetry

Abstract

This report is a deliverable of the **COST Action CA16219** - "Harmonization of UAS techniques for agricultural and natural ecosystems monitoring" (**www.costharmonious.eu**), and describes image enhancement workflow for image velocimetry (removal of visual noise, accentuation of tracer particles), image format handling, the use of different image colorspaces, etc.

The aim of the investigation is to explore which image enhancement methods are most viable for application in UAV image velocimetry workflow, especially when natural or artificial tracer particles are used. Throughout a series of Jupyter notebooks, most important aspects of image quality, formats, colorspace models, filtering methods, and the combinations of the mentioned, will be explored and described.

The code in the notebooks can be used by anyone to improve the accuracy of their image velocimetry. Finally, a "no-code" workflow will be described with the use of the SSIMS tool.

Contents

Disclaimer: The code presented in this and connected notebooks is free to use, adapt, and distribute. The author of the notebooks offers no guarantees that the code will work correctly on every device and hold no liability for any damages that might occur due to its use.

1 Examples

Code in this report is written in Python programming language and prepared for presentation in a series of Jupyter notebooks, although static PDF versions are also available in the `./PDFs` folder. Some of the figures in the notebooks are interactive, and can be manipulated using the icons below or next to them (such as zoom and pan). Image handling will be done using OpenCV library which is by far the most popular C++/Python library for programmatic interaction with image data, including image processing and computer vision.

Libraries necessary for the presentation in this notebook can be imported as:

In [1]:
```
# Necessary libraries
import matplotlib.pyplot as plt
import cv2

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

The report will explore various image enhancement procedures using five images taken during various field campaigns. These images (individual video frames) are located in the `./1080p` and `./4K` folders in two different resolutions (1920-by-1080 px and 3840-by-2160 px). The notebooks use 1080p images,

but you can change the folders to `./4K` if you wish to obtain more detailed view. Images used can be examined using the following code:

In [2]:
```
fig, ax = plt.subplots(nrows=3, ncols=2, figsize=(9.8, 9))

for i in range(5):
    # Using 1080p images for higher loading speed
    img_path = './1080p/{}.jpg'.format(i+1)

    # OpenCV library defaults to Blue-Green-Red (BGR) channel order,
    # which is the order of information in storage/memory (0x00bbggrr)
    img_bgr = cv2.imread(img_path)
    # However, Matplotlib library defaults to a more common
    # Red-Green-Blue (RGB) channel order
    img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)

    ax[i//2][i%2].imshow(img_rgb)
    ax[i//2][i%2].set_title('Image {}'.format(i+1))


ax[2][1].set_visible(False)
[a.axis('off') for a in ax.reshape(-1)]


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

Figure

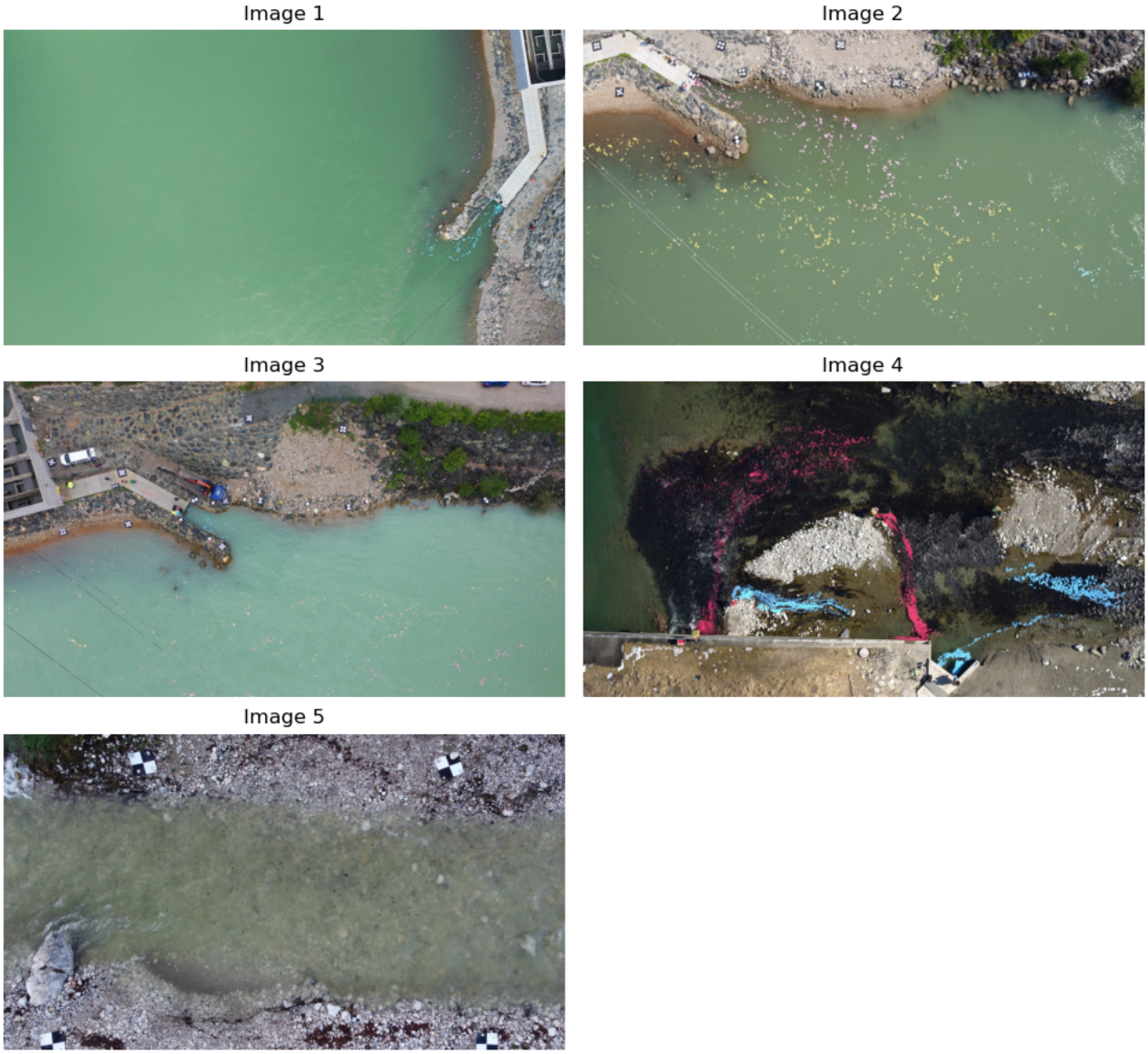


Image 1 presents a case with pink- and cyan-colored artificial tracer particles at the right-hand image side and close to the riverbank. There are some minor light reflections and waves on the water surface as well as some barely visible riverbed features close to the riverbank.

Image 2 presents a case with pink-, yellow-, cyan-, and white-colored tracer particles scattered across a uniform water surface with no light reflections and/or surface waves.

Image 3 is similar to Image 2, except with surface waves that disturb the perception of tracer particles. This image was also taken with a UAV at a higher flight altitude.

Images 1 through 3 are taken at different times on the same site.

Image 4 presents a drastically more challenging case of shallow, clear water, with water surface perturbations caused by shallow flow, and islands of deposited sediment (gravel) in the river itself. Magenta- and cyan-colored artificial tracer particles were used with high seeding density.

Image 5 presents another challenging case with significant surface perturbations and light reflections. This is also a case of a shallow, supercritical flow with relatively clear water where riverbed features are quite prominent. Dark-colored dried leaves were used as low-density seeding (tracer) particles.

2 Image velocimetry workflow

In order to understand the role of the image enhancement for image velocimetry, one should have a solid understanding of the entire workflow from video/image acquisition of the water surface flow to the estimation of surface velocities. The general approach is often involves the following operations, performed in given order:

1. Video/image acquisition,
2. Removal of geometric distortions of the camera,
3. Stabilization of the video,
4. Orthorectification,
5. **Image enhancement**,
6. Velocimetry analyses.

For explanation about steps 1 through 4, the reader can consult the HARMONIOUS video series on the **project's YouTube channel** or the relevant publications listed on the **project's website**. The guiding idea of those steps is to accurately provide a stable frame of reference for the entire video, and to estimate the camera position relative to the real-world with minimal image distortions.

While in some cases velocimetry analyses can begin immediately after step 4, certain surface flow and ground conditions may prove to be unfavorable for a direct application of image velocimetry algorithms - there exist some reflections of light off the water surface, tracer particles are not well-defined relative to the water surface, riverbed is visible and interferes with tracer particle detection, etc. Such conditions call for an application of **image enhancement** algorithms, which allow for a targeted accentuation of tracer particles (features), with the goal of improving the accuracy (i.e., reducing the measurement uncertainty) of velocity estimation.

Alternatively, one could advocate for **image enhancement** to precede steps 2-4 in order to facilitate the remainder of the workflow. However, as it will be shown later on, many of the image enhancement algorithms accentuate certain image features at the expense of deteriorating other (for velocimetry less important) features. Many of such non-tracer features can be used in various steps of the video stabilization and orthorectification, and their deterioration could reduce the reliability and accuracy of said procedures. Additionally, some stabilization and orthorectification methods use certain image enhancement algorithms in their respective implementations, so premature image enhancement can even be detrimental to their overall goals.

Continue to next chapter: Image formats >>

<< Back to MAIN notebook

3. Image formats

There can be some differences between the image data loaded to your code and the same data once it is saved to your storage drive. Image data loaded to your code is kept in RAM memory as an N-dimensional array of pixel values (usually 3-dimensional array of 8bit unsigned integers with pixel values between 0 and 255, or 64bit floats with values between 0.0 to 1.0).

However, image formats used for storing the image data on your storage drive (HDD/SSD) rarely save raw pixel values as that could require unreasonable amounts of storage space. Instead, a compression (i.e., size-reduction) algorithm is applied to the raw data to reduce the storage space requirements. Compression algorithms are generally divided into two groups:

1. **Lossless compression** - no information from the image is lost when it is stored to drive. Once stored, image can be retrieved from drive to reconstruct the identical initial N-dimensional array, i.e., original image.
2. **Lossy compression** - a certain amount of information is lost when image is stored in order to reduce its size in storage. This compression type is commonly used with photographic data such as UAV imagery.

In the following chapters, a number of lossless and lossy formats will be presented along with their pros and cons, to allow the user to consider the balance between image quality and computational complexity of applying image velocimetry workflow, including image enhancement algorithms.

> File size comparisons in this notebook are made by using the 4K resolution images found in the `./4K` folder. Other notebooks use reduced size images (1080p) for efficiency reasons.

Contents

In [1]:
```
# Necessary libraries
import matplotlib.pyplot as plt
import cv2

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

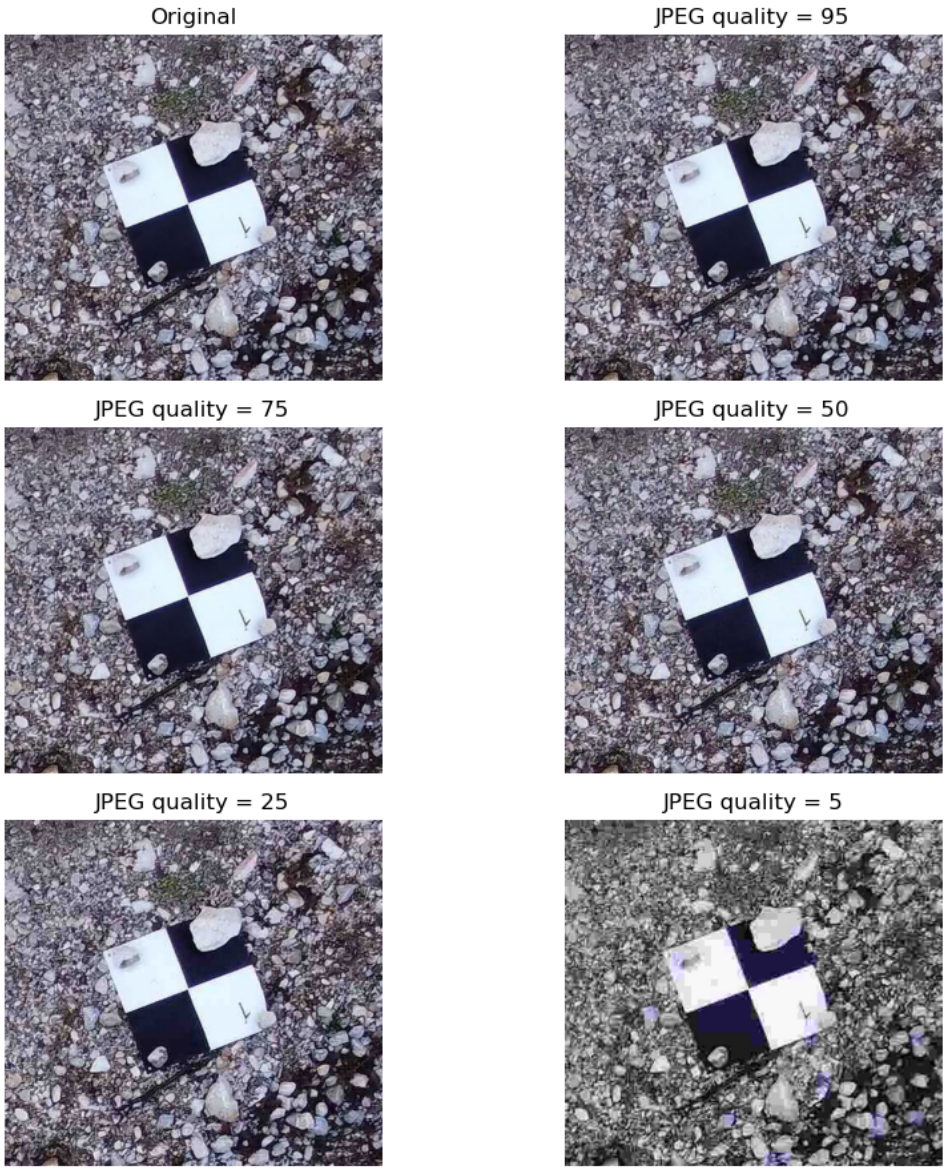
3.1 JP(E)G

JPEG or JPG abbreviates "Joint Photographic Experts Group", and is the most widely used image format. The compression algorithm is **lossy** but can achieve high compression ratio (easily around 10:1). However, there are some variants of JPEG standard which achieve lossless compression, but are rarely used relative to other modern lossless formats.

Consider the following example of a ground control point #1 (top-left) from Image 5. We can vary the JPEG image quality and compare the resulting images (default JPEG quality in OpenCV library is 95):

In [2]:
```
# Read image and crop to center of marker
img_jpeg = cv2.imread('./figures/Fig02.jpg')

# JPEG quality values to plot
image_qual = [95, 75, 50, 25, 5]

# Initiate a 3x2 plot
fig, ax = plt.subplots(nrows=3, ncols=2, figsize=(9.8, 9.0))

# Original image
ax[0][0].imshow(cv2.cvtColor(img_jpeg, cv2.COLOR_BGR2RGB))
ax[0][0].set_title('Original')
ax[0][0].axis('off')

for i, qual in enumerate(image_qual):
    # Encode using JPEG in memory, then decode to numpy array
    _, img_jpeg_enc = cv2.imencode('.jpg', img_jpeg, [int(cv2.IMWRITE_JPEG_QUALITY), qual])
    img_jpeg_dec = cv2.imdecode(img_jpeg_enc, 1)

    # Plot in 2-column figure
    ax[(i+1)//2][(i+1)%2].imshow(cv2.cvtColor(img_jpeg_dec, cv2.COLOR_BGR2RGB))
    ax[(i+1)//2][(i+1)%2].set_title('JPEG quality = {}'.format(qual))

# Hide all figure axes
[a.axis('off') for a in ax.reshape(-1)]

# Show the image
plt.tight_layout()
plt.show()
```

Figure



Images with quality values between 95 and 25 appear to be quite the same until we zoom in on the marker center (using the "zoom to window" option in the interactive figure above). At that zoom level, the differences in colors and edge aliasing degrade noticeably with decreasing JPEG quality. More specifically, there is a noticeable visual noise with lower JPEG quality. If we write the compressed images to disk, we can compare their file sizes:

JPEG quality [-]	File size [KB]	Relative size [%]
100	159.0	100
95	113.0	71.1
90	86.9	54.7
75	60.9	38.3
50	48.3	30.4

JPEG quality [-]	File size [KB]	Relative size [%]
25	**31.5**	**19.8**
5	**11.5**	**7.2**

JPEG format was developed specifically for handling photographic images, such as the ones that are used in UAV velocimetry. For that reason, it is the most commonly used format for such purposes, but the user should be aware that repeated compression of image data has a cumulative effect - the data loss will keep increasing if we:

1. Compress the images when we extract them from the video,
2. Compress the images again when we apply geometric correction techniques,
3. Compress the images again when we stabilize the image sequence,
4. Compress the images again when we orthorectify images,
5. ...

The aforementioned workflow should be avoided **as much as possible** to preserve as much original visual information and avoid adding noise to our images. Some preprocessing tools for UAV velocimetry, such as the **SSIMS tool** algorithmically bundle certain steps together (steps 2-4) to reduce the total number of image compression iterations.

If JPEG format is used, it is advised to keep the JPEG quality at most at 95 (default in OpenCV library) since higher values produce significantly larger file sizes with no noticeable quality improvements. In fact, some lossless image formats may even produce smaller file sizes than JPEG with quality of 100. Lower boundary of acceptable JPEG quality is often cited to be between 75 and 80.

> OpenCV uses a well established `libjpeg` library for JPEG compression. However, some libraries such as the `MozJPEG` have been proven to be more efficient for JPEG compression, but are not implemented in the OpenCV and are not used as often.

3.2 PNG

Second most popular image format is PNG, which stands for "Portable Network Graphics". PNG is a lossless format, meaning no information is lost when saving an image to storage drive. Additionally, PNG supports transparency using an additional Alpha channel (4th channel apart from the usual RGB/BGR).

Unlike with JPEG format, when using OpenCV library to save a PNG image, user can only select the **compression level**. While both higher and lower compression level retain all image information, higher compression level will result in a smaller file size but longer time needed to decode the image from storage drive to memory (and vice versa).

Keep in mind that PNG images have larger file sizes than JPEG with little to no noticeable effect of in image quality, so this format is rarely used for UAV velocimetry purposes. To provide a perspective, the PNG file size of the original image from the previous subsection is **557KB with compression level 0**

(350% increase relative to JPEG with quality of 100) and **342KB with compression level 9** (215% increase relative to JPEG with quality of 100).

> Interestingly, the lowest file size is not always achieved with the highest PNG compression level - the same image saved with **level 1 has a file size of 335KB**.

3.3 WebP

The youngest of the image formats described in this report is WebP developed by Google for web images and graphics. Unlike JPEG or PNG, it can be both lossless and lossy depending on the selected compression strength. As claimed by Google, "*[lossless WebP images are] 26% smaller in size compared to PNGs*" and "*[lossy WebP images are] 25-34% smaller than comparable JPEG images at equivalent SSIM quality index*". Their use for computer vision tasks such as velocimetry is rare, possibly due to the late introduction of WebP support on PCs. OpenCV library supports WebP since 2013, but the initial support was rather buggy until versions 4.x.

However, comparing lossless WebP with PNG and lossy WebP with JPEG reveals a significant advantage of this new format for general purpose computing. Lossless WebP version of the original image from subsection 3.1 with image quality set to 100 (default in OpenCV) yields a file size of 123KB which is only 36.7% of the PNG image with compression level 1. Furthermore, this is a smaller file size than with JPEG image with quality set to 100 (by 22.6%), and is only 8.8% larger than JPEG with quality of 95 (default in OpenCV).

We can also compare lossy WebP compression with the popular JPEG format with the code below. It is noticeable that for similar file sizes (WebP file sizes are actually somewhat lower than JPEG for the same quality parameter, up until the value of around 35), lossy WebP produces significantly lower amounts of compression artifacts even for quality of 5 which produces a file size of just 19KB.

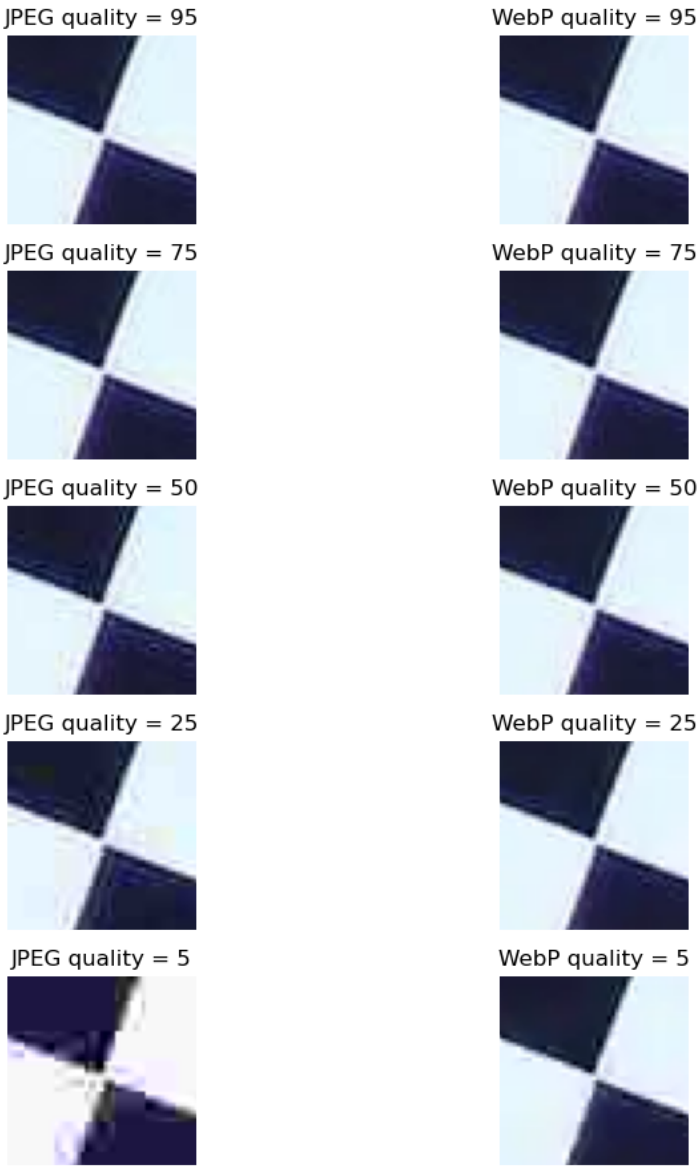
All the above makes the WebP format a strong contender for replacing JPEG as the go-to format for image velocimetry, assuming that necessary libraries for handling such files are installed.

> Keep in mind that not all image viewing software supports WebP format.

In [3]:
```
# WebP image quality to plot
webp_qual = [95, 75, 50, 25, 5]

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

for i, qual in enumerate(webp_qual):
    # Encode using JPEG in memory, then decode to numpy array
    _, img_jpeg_enc = cv2.imencode('.jpg', img_jpeg, [int(cv2.IMWRITE_JPEG_QUALITY), qual])
    img_jpeg_dec = cv2.imdecode(img_jpeg_enc, 1)

    # Encode using WebP in memory, then decode to numpy array
    _, img_webp_enc = cv2.imencode('.webp', img_jpeg, [int(cv2.IMWRITE_WEBP_QUALITY), qual])
    img_webp_dec = cv2.imdecode(img_webp_enc, 1)

    ax[i][0].imshow(cv2.cvtColor(img_jpeg_dec, cv2.COLOR_BGR2RGB))
    ax[i][0].set_title('JPEG quality = {}'.format(qual))
    ax[i][0].axis('off')
```

```
    ax[i][1].imshow(cv2.cvtColor(img_webp_dec, cv2.COLOR_BGR2RGB))
    ax[i][1].set_title('WebP quality = {}'.format(qual))
    ax[i][1].axis('off')

# Zoom to GCP marker center for a better view
for a in ax.reshape(-1):
    a.set_xlim([195, 245])
    a.set_ylim([180, 230])

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

Figure



3.4 TIF(F)

TIFF is an abbreviation of "Tagged Image File Format", and is less commonly used for image velocimetry. While TIFF format can be either lossless or lossy, in practice it is almost exclusively used for lossless image storage without compression, which results in very large file sizes - there is an option of LZW (Lempel–Ziv–Welch) compression for TIFF but it is not supported on all devices. For example,

uncompressed TIFF file size of Image 3 is 23.8MB but LZW-compressed TIFF is only 11.6MB (48.7% reduction). However, the same image as compressed JPEG (quality 95) is just 2.2MB (1072% and 523% reduction, respectively). Lossless PNG file size, on the other hand is close to LZW TIFF with 10.7MB.

> LZW is the default compression method for TIFF in OpenCV library.

For the aforementioned reasons, TIFF is generally avoided for image velocimetry purposes, but it can rival PNG if LZW compression is used.

However, TIFF format supports various features not common with some other formats like 16bit images, image layering/multipage images, transparency, standardized metadata assignment, etc. Because of these additional capabilities, TIFF format is often used to represent results of survey/photogrammetry analyses such as digital elevation models (DEMs).

3.5 BMP

BMP or bitmap (or "bump") is a lossless image format commonly used on Windows devices. While such images can be compressed using a lossless algorithm, this is uncommon and they are usually found uncompressed making their file size substantial. Actually, BMP image file size should be exactly the same as of the uncompressed TIFF variant, which severely limits its usability for image velocimetry purposes.

3.6 GIF

While rarely used for color images because it only supports 256 colors per image, GIF format (or "Graphical Interchange Format") can be safely used for lossless compression and storage of 8bit grayscale images - which are commonly used for image velocimetry purposes. Like the TIFF format, it used LZW compression algorithm.

Continue to next chapter: Image colorspaces >>

or

<< Back to MAIN notebook

<< Back to MAIN notebook

4 Image colorspace

A **colorspace** is an organizational model of image data, i.e., the way visual information is handled inside the memory, as well as saved to and retrieved from storage. In general, different image formats can use different colorspace models - while JPEG images usually use RGB (Red-Green-Blue) model, they can, for example, use CMYK (Cyan-Magenta-Yellow-Key; Key=Black) model instead. As it will be shown in this section, image enhancement can benefit greatly from manipulating images in different colorspace models and accessing different kinds of information stored within.

Contents

In [1]:
```
# Necessary libraries
import itertools
import numpy as np
import matplotlib.pyplot as plt
import cv2

from os import mkdir
from os.path import exists, dirname
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
```

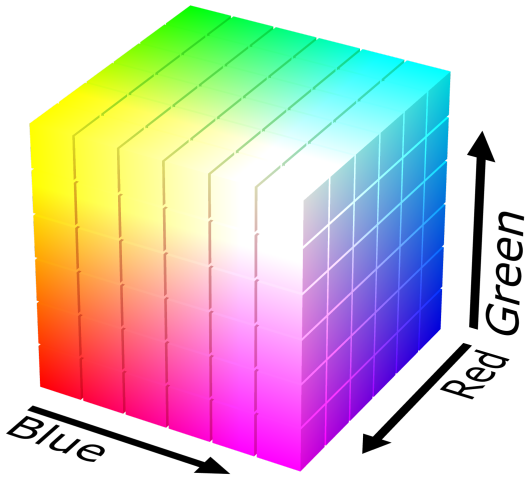
4.1 RGB/BGR

RGB and BGR are the most commonly used (so-called additive) colorspace models. RGB model contains 3 layers (channels) of information about the intensity of red, green and blue light in the image, while BGR is basically the same model with the reversed positions of the first and the third channel. All three channels are usually 8bit unsigned integers with pixel values between 0 and 255. While most image processing software/tools default to the RGB model, OpenCV library uses BGR as default colorspace.

Reasons behind this are mostly historical, but they mean that sometimes we have to switch from one to the other to correctly present our results.

> RGB and BGR sometimes also refer to different subpixel order of LEDs in monitor and TV panels. While there are some differences between these panel types, this report will only refer to them as data organizational models. Some image formats support RGBA colorspace with incorporates additional Alpha channel which defines opacity (non-transparency) of individual pixels. This channel usually contains values between 0.0 (= fully transparent pixel) and 1.0 (= fully opaque pixel). However, no image or video format used by UAV cameras supports transparency and saves to this colorspace.

The components of the RGB/BGR model are commonly presented graphically in the form of a cube, as shown below:



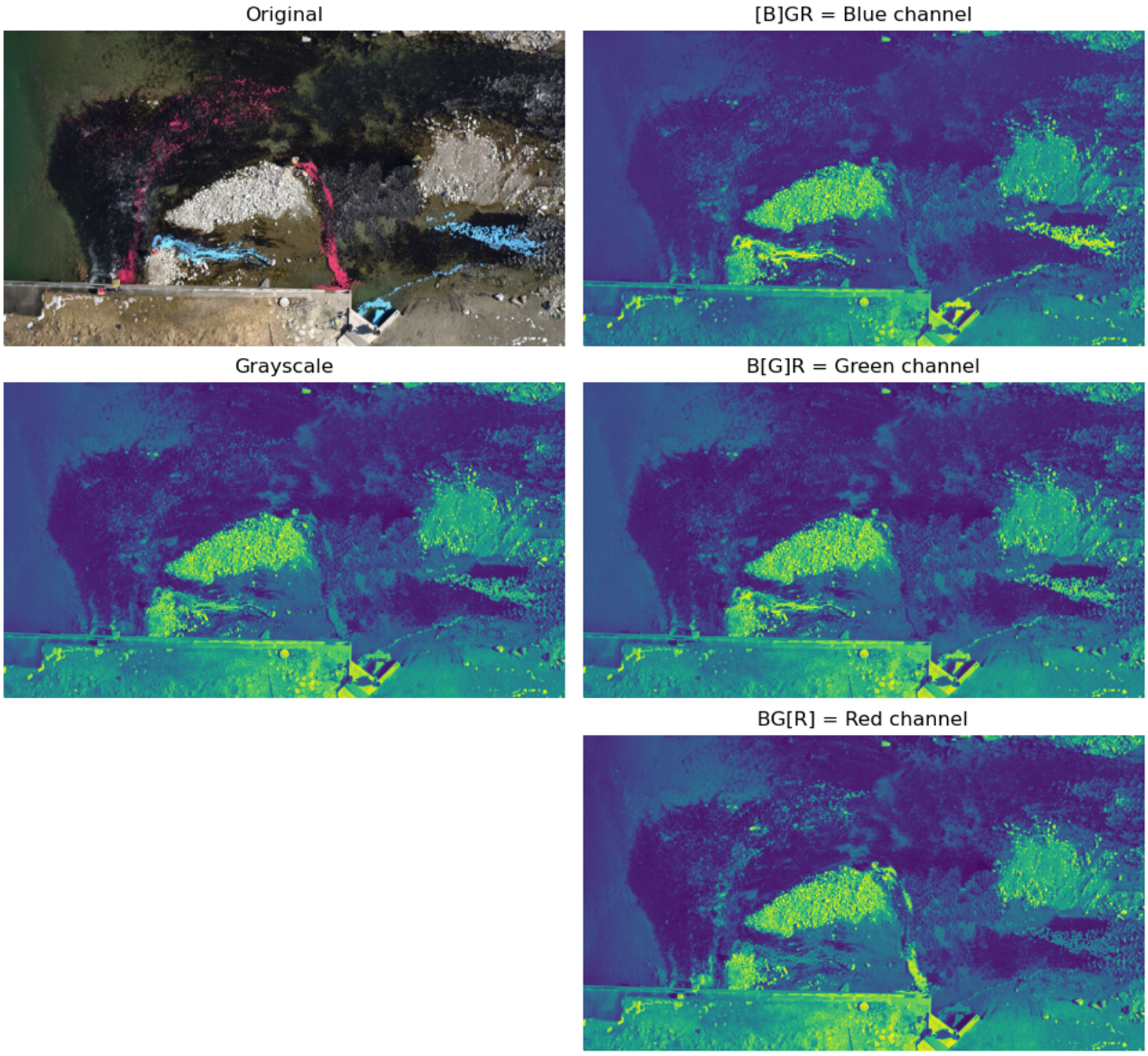
RGB/BGR colorspace model representation, source: Wikimedia

We can explore these channels by decomposing an example image, and also compare the results to a single-channel grayscale image. A suitable candidate could be **Image 4** due to the variety of tracer particle colors:

In [2]:
```
# Load image into different colorspaces
img_path = './1080p/4.jpg'

# OpenCV loads to BGR model by default
img_bgr = cv2.imread(img_path)

# Conversion to grayscale model
img_gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)

# Conversion to RGB model
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)

# Split image into channels
b, g, r = cv2.split(img_bgr)

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

```
# Matplotlib expects RGB image
ax[0][0].imshow(img_rgb)
ax[0][0].set_title('Original')

ax[1][0].imshow(img_gray)
ax[1][0].set_title('Grayscale')

ax[2][0].set_visible(False)

ax[0][1].imshow(b)
ax[0][1].set_title('[B]GR = Blue channel')

ax[1][1].imshow(g)
ax[1][1].set_title('B[G]R = Green channel')

ax[2][1].imshow(r)
ax[2][1].set_title('BG[R] = Red channel')

# Turn off axes on all images
[a.axis('off') for a in ax.reshape(-1)]

# Connect all axes to simultaneously change on zoom or pan
[a.callbacks.connect('ylim_changed', on_lims_change) for a in ax.reshape(-1)]

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

Figure



Tip: From this point onward, the brackets in colorspace names in the text and figure titles will indicate the selected channel of the mentioned colorspace model - for example R[G]B indicates the green channel of the RGB model.

Tip: Figures with axes (images) connected to `on_lims_change` function will change together when ZOOM or PAN are used on any of them.

> Individual image channels and grayscale image above appear to be colored due to the colormap set by the `matplotlib` library which allows for a better contrast. Default `matplotlib` colormap is called `viridis`. You can change the default colormap in `axes_tiein.py`.

We can now notice the differences in local pixel intensities between individual image channels, especially in areas covered by magenta- (around coordinates [900, 200]) and cyan-colored tracer particles (around

[1680, 700]). While most velocimetry software operates on a single channel image, usually grayscale (described in the following subsection), it's not hard to notice that in the selected case some tracer particles are far more accentuated in [R]GB and RG[B] channels than in grayscale. Additionally, we can notice that the water surface is the brightest in the R[G]B channel (which was to be expected from the original) and darkest and most uniform in the RG[B] channel (check area around [1020, 180]). As a rule of thumb, we want the water surface to be uniform and tracer particles to have as much contrast from it as possible.

Choosing a suitable image channel for velocimetry analysis could allow for a better quality of tracer particle detection and their motion tracking, and improve the overall velocity estimation accuracy. A candidate strategy for analyzing the video from which the **Image 4** was extracted could also be to perform the image velocimetry on [R]GB and RG[B] image channels separately, and then merge the results in the postprocessing stage.

> **Tip:** Defaulting to a grayscale representation of original color images for image velocimetry does not have to be the best option for each case. Consider using individual RGB image channels if colored tracer particles were used for surface seeding. Even with white/dark tracers, some image channels may even provide better contrast between the image background (water surface) and the particles themselves.

4.2 Grayscale

Even though not technically a colorspace, it is often used when a single channel representation of color images is required. Grayscale model aims to describe the human perception of pixel intensities of different colors, i.e., the "human-perceived achromatic intensity". For example, conversion of an image from RGB/BGR model to grayscale can be done using a linear combination of its color channels (so called NTSC model):

$$Y = 0.299R + 0.587G + 0.114B,$$

Where Y is the grayscale pixel value with R, G and B values from individual RGB channels. The previous expression implies (and correctly so) that human perception of red, green, and blue light is not uniform - human eye is more sensitive to green (or green-yellow) color than to blue, which is reflected by the transformation coefficients.

Grayscale model is the most commonly used single channel representation of images, but it is not always a good one. For example, we can create an image from RGB value of [255, 0, 0] on the left half, [0, 130, 0] in the middle, and [0, 81, 251] on the right, convert to grayscale and observe the results:

In [3]:
```
# Create images by tiling pixel values
img_left = np.tile((255, 0, 0), np.array((300, 150, 1)))
img_middle = np.tile((0, 130, 0), np.array((300, 150, 1)))
img_right = np.tile((0, 81, 251), np.array((300, 150, 1)))

# Merge red and blue image halves
img_merged = np.hstack([img_left, img_middle, img_right]).astype('uint8')
```

```
# Convert RGB to grayscale
img_merged_gray = cv2.cvtColor(img_merged, cv2.COLOR_RGB2GRAY)

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

# RGB plot
ax[0].imshow(img_merged)
ax[0].set_title('RGB')

# Grayscale plot, set colormap and color range for proper display
ax[1].imshow(img_merged_gray, cmap='gray', vmin=0, vmax=255)
ax[1].set_title('Grayscale intensity')

[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

Previous figure demonstrates the potential shortcomings of grayscale channel representation of RGB/BGR images, along with the demonstrations from the previous subsection.

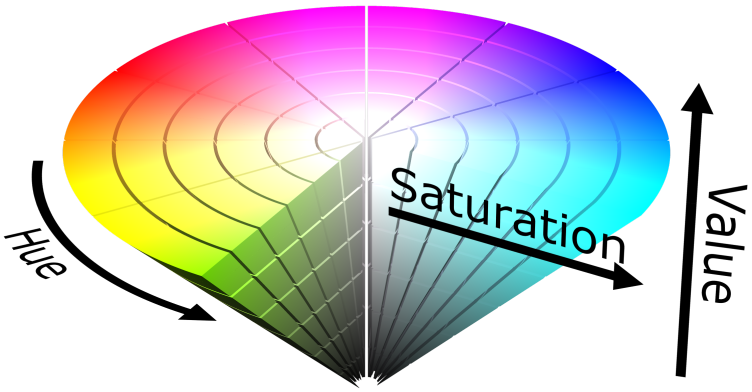
4.3 HSV

Unlike RGB/BGR model where individual colors intensities are represented in separate image channels, HSV takes a somewhat different approach of separating image data into three channels:

1. [H]SV = **Hue** channel (presented as an angle between 0 and 360 degrees) which represents the **primary (base) color** or **color tone**. Hue angle of 0deg corresponds to the red color, 90deg is yellow, 270deg is blue, etc.
2. H[S]V = **Saturation** channel (0-255 unsigned 8bit integer) which represents amount of gray in the base color, often called **chroma**. When saturation is 0, the resulting color will appear completely "faded", and when saturation is 255, the result will be the actual base color.
3. HS[V] = **Value** channel (0-255 unsigned 8bit integer) which represents the color intensity/brightness. When value is 0, the result will be a black pixel, and when value is 255, the result will be completely defined by just the hue and saturation.

> OpenCV defines the [H]SV component in range between 0 and 180deg, as opposed to the more formal definition above. Reasons for this are purely implementational - since both H[S]V and HS[V] are 8bit unsigned integers (0-255), creators of the library "squashed" the [H]SV channel range from 360 to 180 (degrees). However, OpenCV also offers a so-called `HSV_full` model, which stretches the [H]SV channel to range between 0 and 255.

Components of the HSV model are commonly graphically presented in the form of a cone, as shown below.



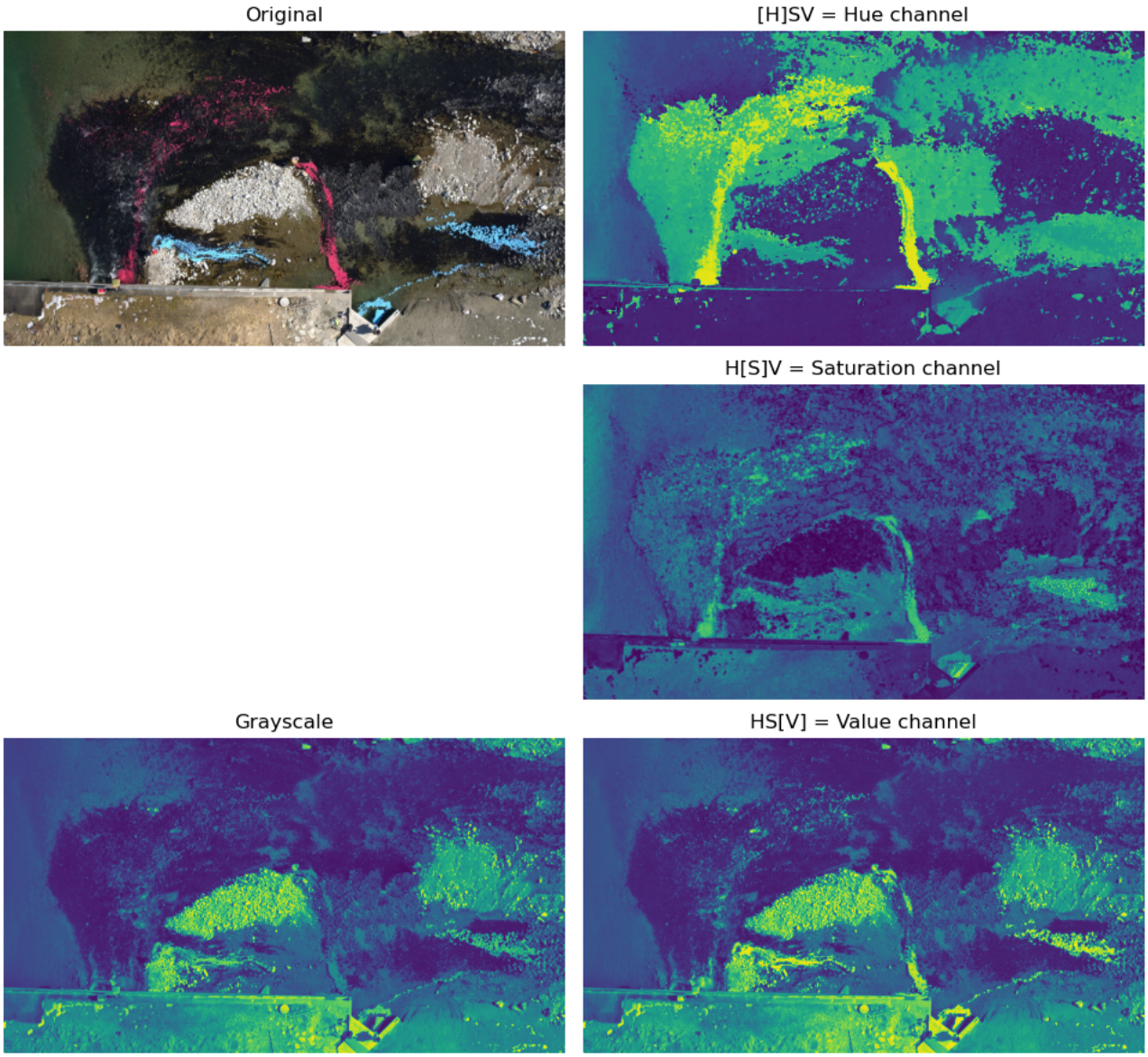
HSV colorspace model representation, source: Wikimedia

Like with the RGB/BGR model, we can explore these channels by decomposing an example image, and also compare the results to a single channel grayscale image:

In [4]:
```
# BGR image loaded in one of the previous code cells
img_hsv = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2HSV)

# Split image into channels
h, s, v = cv2.split(img_hsv)

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

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

ax[1][0].set_visible(False)

ax[2][0].imshow(img_gray)
ax[2][0].set_title('Grayscale')

ax[0][1].imshow(h)
ax[0][1].set_title('[H]SV = Hue channel')

ax[1][1].imshow(s)
ax[1][1].set_title('H[S]V = Saturation channel')

ax[2][1].imshow(v)
ax[2][1].set_title('HS[V] = Value channel')
```

```
[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



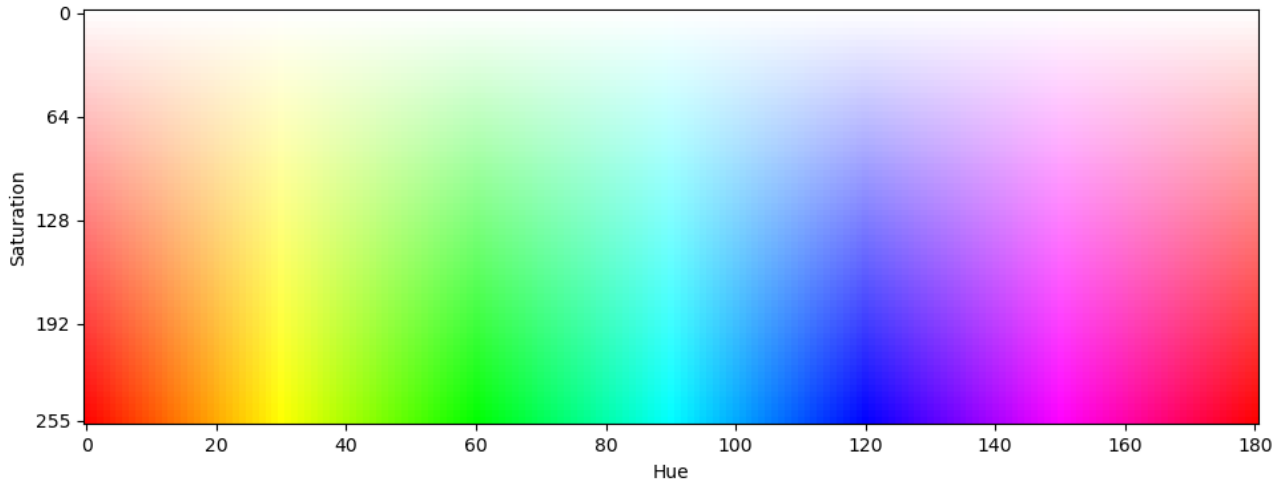
When placed next to each other, grayscale model and HS[V] channel appear alike - this is not by accident as grayscale is primarily a representation of human perception of color brightness. However, in the case of **Image 4**, tracers in the HS[V] channel appear to be significantly more pronounced than in the grayscale image, and more equally so for both magenta and cyan particles. H[S]V channel does not deliver much information and is rather noisy.

The [H]SV channel can actually serve another purpose. We can use the [H]SV channel information to target and manipulate specific colors in the image. To do so, we should first explore the [H]SV and H[S]V components by creating a graph:

In [5]:
```
# Initiate a 64x180px three channel image
```

```
height, width = 64, 181
img = np.ndarray([height, width, 3], dtype='uint8')

fig, ax = plt.subplots(figsize=(9.8, 4))

# Assign Hue and Saturation to each pixel
for i, j in itertools.product(range(height), range(width)):
    img[i, j] = (j, i*4, 255)

# Convert HSV image to RGB for plotting
ax.imshow(cv2.cvtColor(img, cv2.COLOR_HSV2RGB))
ax.set_xlabel('Hue')
ax.set_ylabel('Saturation')

plt.xticks(np.arange(0, width, step=20))
plt.yticks([0, 16, 32, 48, 63], [0, 64, 128, 192, 255])
plt.tight_layout()
plt.show()
```

Figure

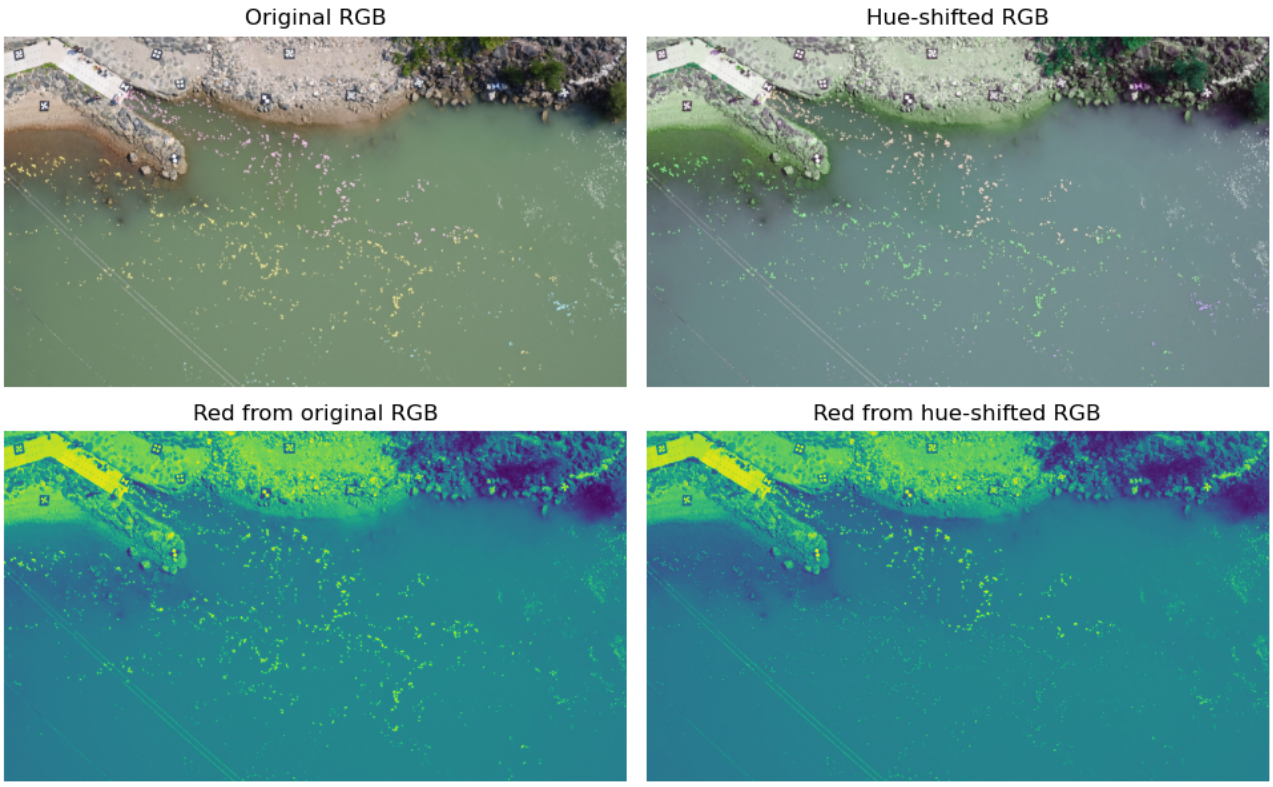


HSV model allows us to do something that RGB/BGR does not - to manipulate color hue in order to better grab certain information from RGB channels. For instance, pink-colored tracer particles from **Image 2** have a hue value of about 145 (check by hovering with mouse over these areas in figures above). By manipulating the [H]SV component, we can "shift" hue values (or "rotate" the HSV cone around its vertical axis) of all pixels in an image to move certain information to a specific channel - let's try to shift the pink tracers to the red channel:

In [6]:
```
# Load image using different colorspaces
img_path = './1080p/2.jpg'
img_bgr = cv2.imread(img_path)
img_hsv = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2HSV)
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)

# Get the first (red) channel from RGB
img_rgb_red = img_rgb[:, :, 0]

# Split image into channels
h, s, v = cv2.split(img_hsv)
```

```
# Add 35 to hue channel to move pink to red, and merge to new HSV image.
# While channel can be changed with simple addition (h+35), this should be avoided
# as it can cause an overflow (value can exceed 255 and be improperly converted).
# Function cv2.add() deals with this issue.
h_shift = cv2.add(h, 35)

# Merge the new HSV image
img_hsv_shift = cv2.merge([h_shift, s, v])

# Convert to RGB and get the red channel
img_rgb_shift = cv2.cvtColor(img_hsv_shift, cv2.COLOR_HSV2RGB)
img_rgb_shift_red = img_rgb_shift[:, :, 0]

fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(9.8, 6.2))

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

ax[0][1].imshow(img_rgb_shift)
ax[0][1].set_title('Hue-shifted RGB')

# Only show channel 0, i.e., the red channel
ax[1][0].imshow(img_rgb_red)
ax[1][0].set_title('Red from original RGB')

# Only show channel 0, i.e., the red channel
ax[1][1].imshow(img_rgb_shift_red)
ax[1][1].set_title('Red from hue-shifted 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



As the figure above demonstrates, by strategically "shifting" the hue value we have also modified the RGB contents of the image. Tracer particles that were initially pink have obtained more red color content and are now more pronounced in the [R]GB channel. Additionally, the red channel has become less sensitive to the yellow colored tracer particles, as these have also moved towards higher hue values (towards the R[G]B channel).

There are other methods of manipulating the HSV colorspace (for example by creating lookup tables) but such strategies can be considerably more complex and worthwhile only in specific cases, thus will not be described in this report.

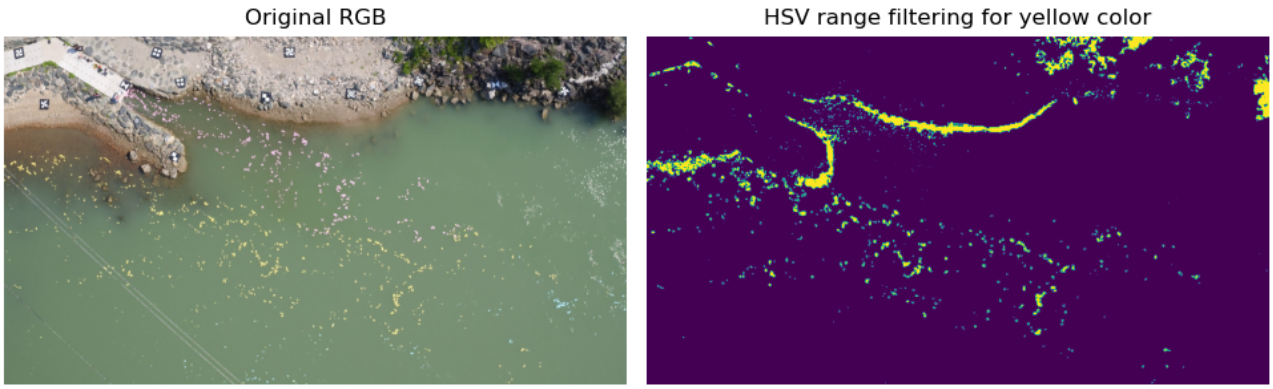
> **Tip:** By "shifting" the hue values in the HSV colorspace, we can strategically target specific colors in the image, regardless of their original RGB values.

Another thing possible in the HSV colorspace is masking certain image features by using range of hue values. For example, we can split the yellow-colored tracer particles from the background:

In [7]:
```
# Yellow hue is around 30, so we can take 30 +/- 10
# Saturation lower boundary found by trial and error
upper_limit = (40, 255, 255)
lower_limit = (20, 60, 0)

# Mark pixels in the range defined above, using original HSV
mask = cv2.inRange(img_hsv, lower_limit, upper_limit)

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(mask)
ax[1].set_title('HSV range filtering for yellow color')

[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



Warning: It is noticeable that filtering by HSV range also detects some other areas that are not strictly yellow. However, these regions usually do not move and will not be detected by the image velocimetry algorithm. The downside of this approach is that the resulting image is binarized - it contains only zeros (pixels outside the provided HSV range) and ones (pixels inside the defined range). This can lead to particles slightly changing shape between consecutive frames, which can increase the velocity estimation uncertainty.

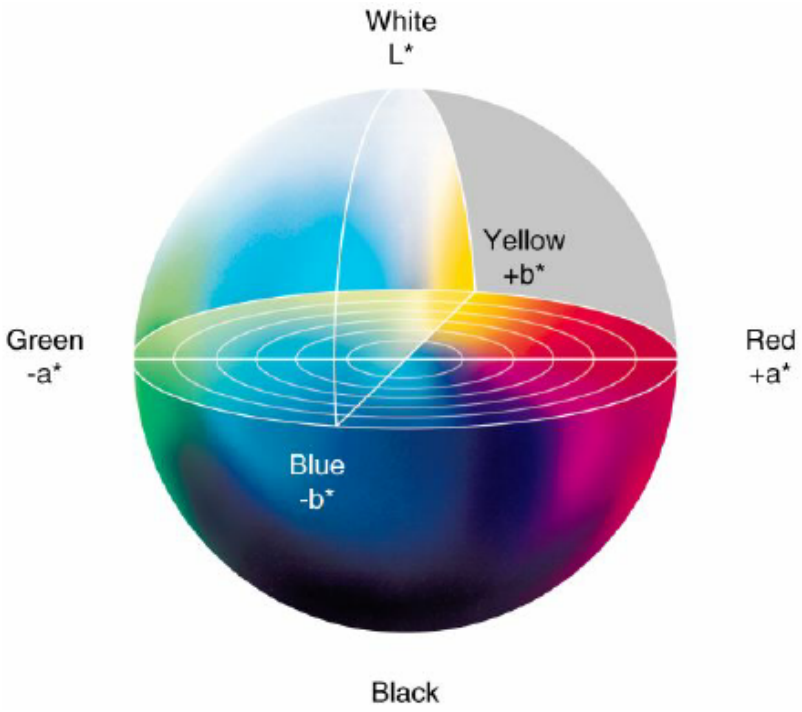
4.4 L*a*b*

L*a*b* colorspace (sometimes referred to as CIELAB or CIELab) is a three-channel model developed to be a more accurate representation of the human color perception. Components of the model are:

1. **L*** channel, sometimes referred to as "L-star", defines the lightness of the pixels, usually in the range of values between 0 (black) and 100 (white), with maximal chroma values in the middle.
2. **a*** channel defines the content of base colors between red (+a*) and green (-a*).
3. **b*** channel defines the content of base colors between yellow (+b*) and blue (-b*).

> OpenCV library defines L*a*b* channels with values in range 0..255.

Like RGB/BGR and HSV, the L*a*b* model has a graphical representation - usually in the form of a sphere.



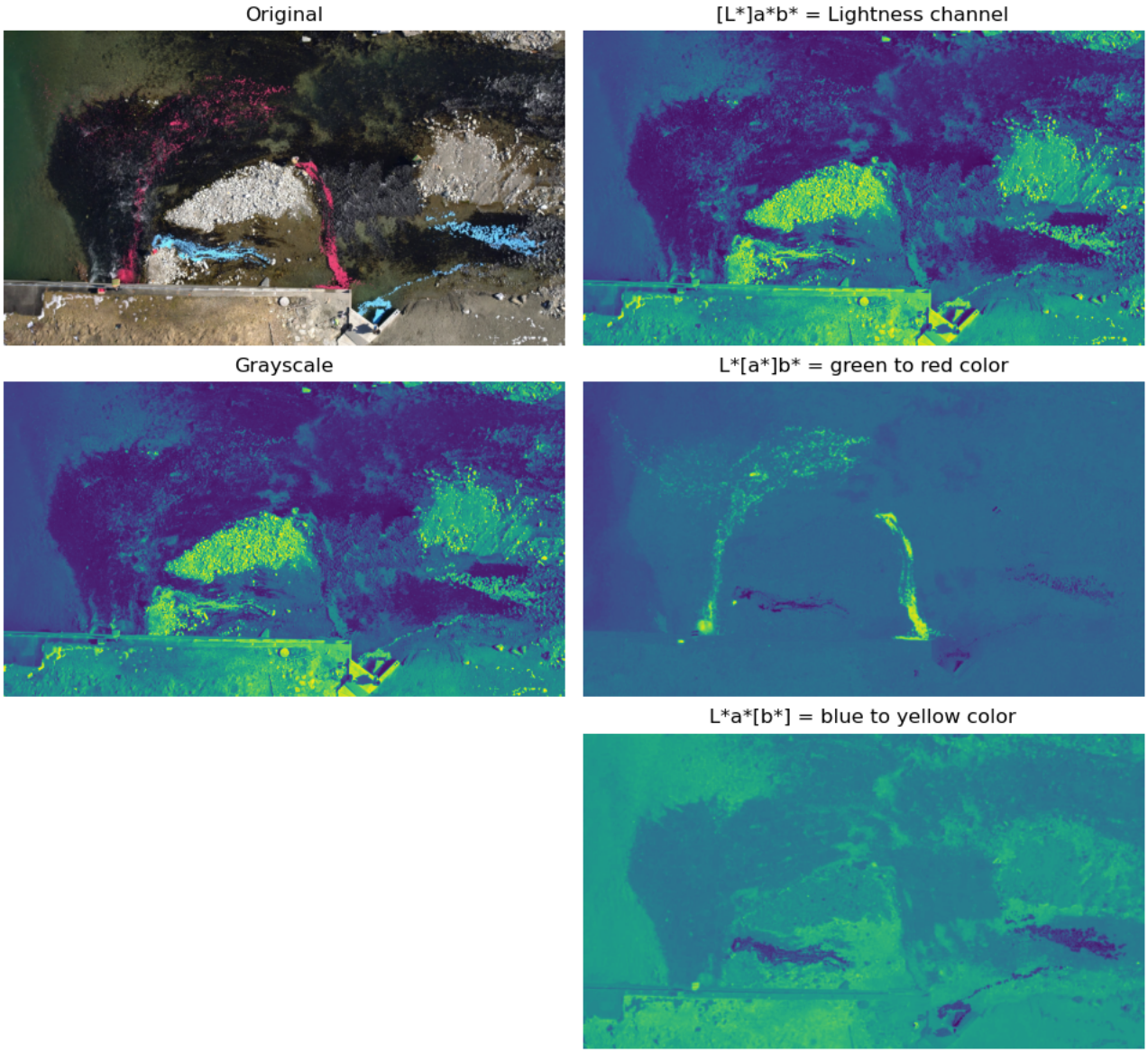
*L*a*b* colorspace model representation, (Agudo et al., 2014)*

Unlike RGB/BGR, the L*a*b* model covers a broader color gamut, i.e., broader range of colors can be represented with it. We can visualize the individual channels of the L*a*b* colorspace to get a better understanding of how it works (using **Image 4** as an example):

In [8]:
```
img_path = './1080p/4.jpg'
img_bgr = cv2.imread(img_path)
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
img_gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)
img_lab = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2LAB)

# Split image into channels
l, a, b = cv2.split(img_lab)

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

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

ax[1][0].imshow(img_gray)
ax[1][0].set_title('Grayscale')

ax[2][0].set_visible(False)

ax[0][1].imshow(l)
ax[0][1].set_title('[L*]a*b* = Lightness channel')

ax[1][1].imshow(a)
ax[1][1].set_title('L*[a*]b* = green to red color')

ax[2][1].imshow(b)
ax[2][1].set_title('L*a*[b*] = blue to yellow color')

[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



From the figure above, it is clear that the [L*]a*b* channel is perceptually very similar to the grayscale representation and HS[V]. However, it usually contains a bit more contrast between light and dark image areas than said models. Zooming in on region of cyan-colored tracer particles around coordinates (1620, 700) reveals a somewhat darker water surface and a somewhat lighter particles in the [L*]a*b* channel than in the grayscale image. Similarly, L*[a*]b* and L*a*[b*] channels hold very few visual disturbances from the image background (water surface) and can be used to manually track both magenta- and cyan-colored particles. The downside of using the L*[a*]b* and L*a*[b*] channels is that they are often quite blurry, and can require additional manipulations to be a viable model.

Reference:

> Agudo, J., Pardo, P., Sánchez, H., Pérez, Á., & Suero, M. (2014). A Low-Cost Real Color Picker Based on Arduino. In Sensors (Vol. 14, Issue 7, pp. 11943–11956). MDPI AG. https://doi.org/10.3390/s140711943

4.5 Conclusions on colorspace models

Access to different colorspace models offers different insights into image data. Instead of defaulting to the well-known grayscale representation, one should also explore different models and their individual channels, and focus the image velocimetry analyses on specific visual information such as tracer particles, or to just maximize the contrast between the image background (water surface) and the foreground (features that are to be tracked). Utilization of HSV and L*a*b* models, in particular, can sometimes be a better alternative than grayscale or RGB/BGR models.

Whatever the strategy, the colorspace manipulations should precede the application of any image filtering techniques, some of which are to be described in the following section. To help choose the best colorspace to work with, we can select a suitable sample image with visible tracer particles, and use the code bellow to go output and examine all the described models and their individual channels:

In [9]:
```
# Select a sample image
img_sample_path = './1080p/1.jpg'

# Get image extension
ext = img_sample_path.split('.')[-1]

# Get image directory
img_dir = dirname(img_sample_path)

# Output directory
out_dir = img_dir + '/colorspaces'

# Conversions
img_sample_bgr = cv2.imread(img_sample_path)
img_sample_hsv = cv2.cvtColor(img_sample_bgr, cv2.COLOR_BGR2HSV)
img_sample_lab = cv2.cvtColor(img_sample_bgr, cv2.COLOR_BGR2LAB)

# Split into channels
rgb_b, rgb_g, rgb_r = cv2.split(img_sample_bgr)
hsv_h, hsv_s, hsv_v = cv2.split(img_sample_hsv)
lab_l, lab_a, lab_b = cv2.split(img_sample_lab)
gray = cv2.cvtColor(img_sample_bgr, cv2.COLOR_BGR2GRAY)

# Create a directory in image folder
if not exists(out_dir):
    mkdir(out_dir)

# List all the channels for iterating
channels = [
    rgb_b, rgb_g, rgb_r,
    hsv_h, hsv_s, hsv_v,
    lab_l, lab_a, lab_b,
    gray
]

for c in channels:
    # Get variable/channel name
    c_name = [k for k, v in locals().items() if v is c][0]
```

```
    # Scale to 0..255
    c = (c / c.max() * 255).astype('uint8')

    # Apply a colormap for easier inspection
    c = cv2.applyColorMap(c, cv2.COLORMAP_VIRIDIS)

    # Write channel to the output folder
    cv2.imwrite('{}/{}.{}'.format(out_dir, c_name, ext), c)
```

Continue to next chapter: Image filtering >>

or

<< Back to MAIN notebook

<< 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

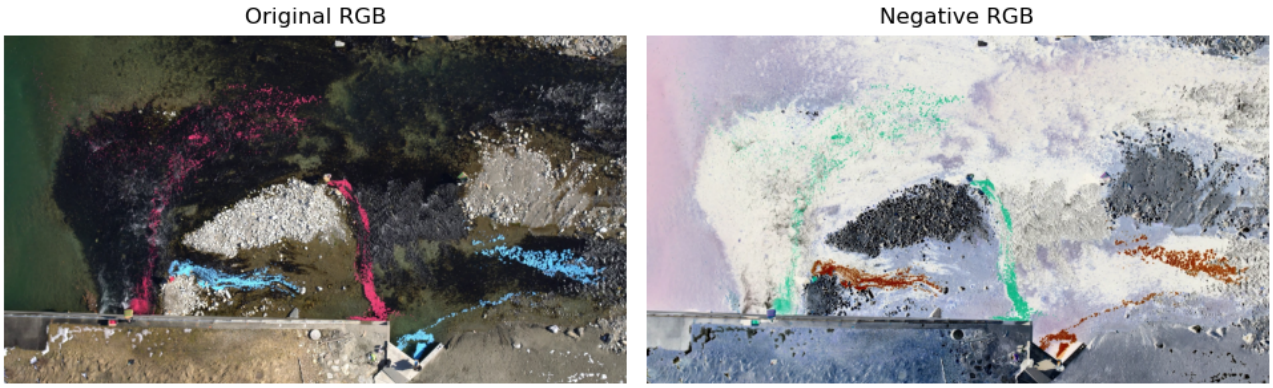
Negative of an image is obtained by "reversing" the intensity of its colors, which is often implemented by subtracting its pixel values from the maximal pixel intensity - usually 255 (for an 8bit image). The negative of an image can be obtained both for a single-channel and multi-channel image.

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



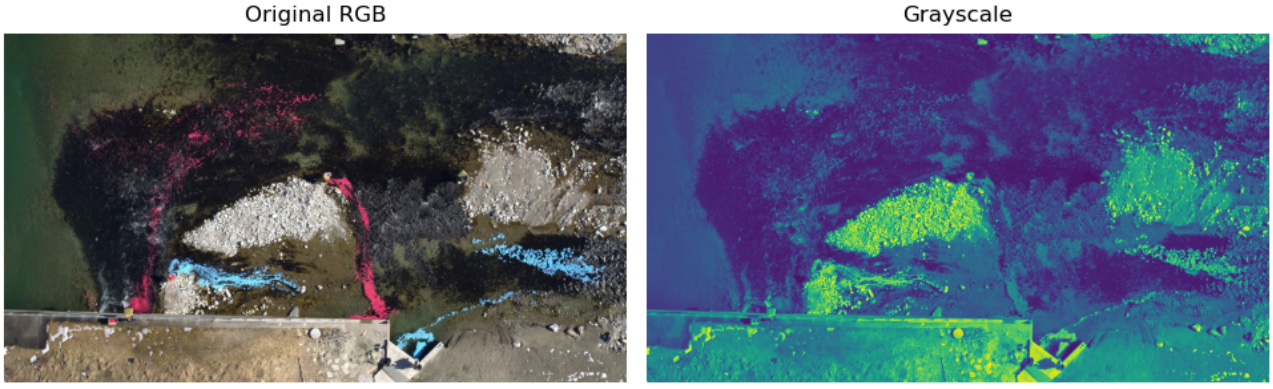
5.2 Conversion to grayscale

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.

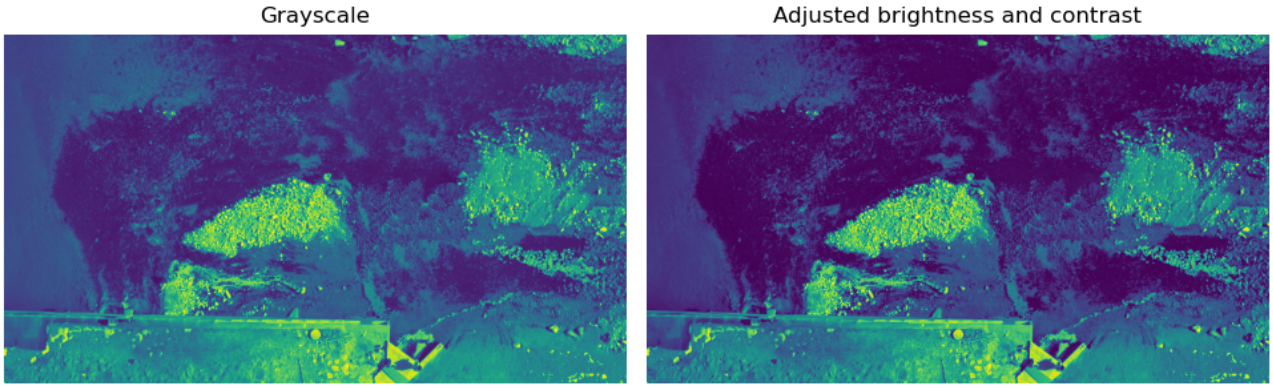
> Instead of performing the transformation directly, it is advised to use the inbuilt OpenCV finction `cv2.convertScaleAbs()` to avoid overflow problems.

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



5.4 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):

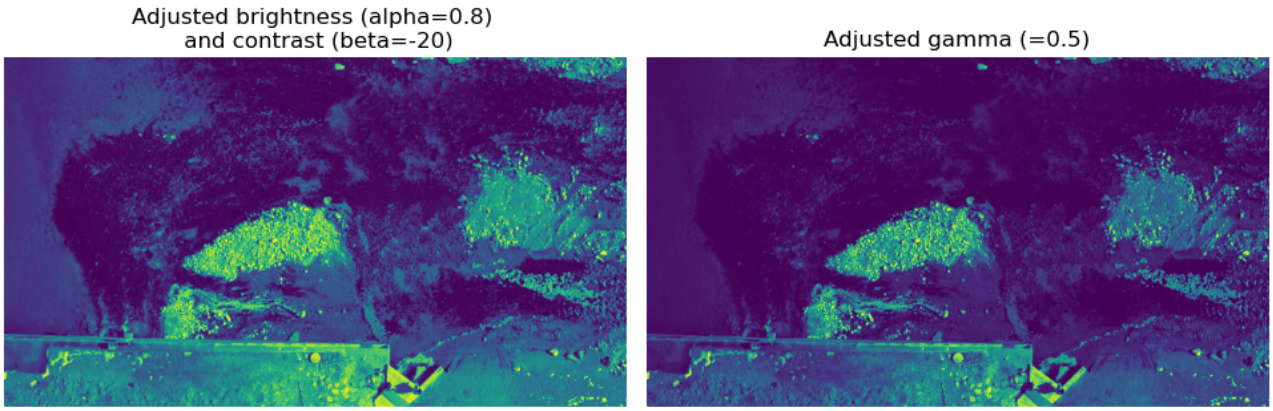
$$Result = (Original/Y_{max})^{1/\gamma} \times Y_{max},$$

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()
```

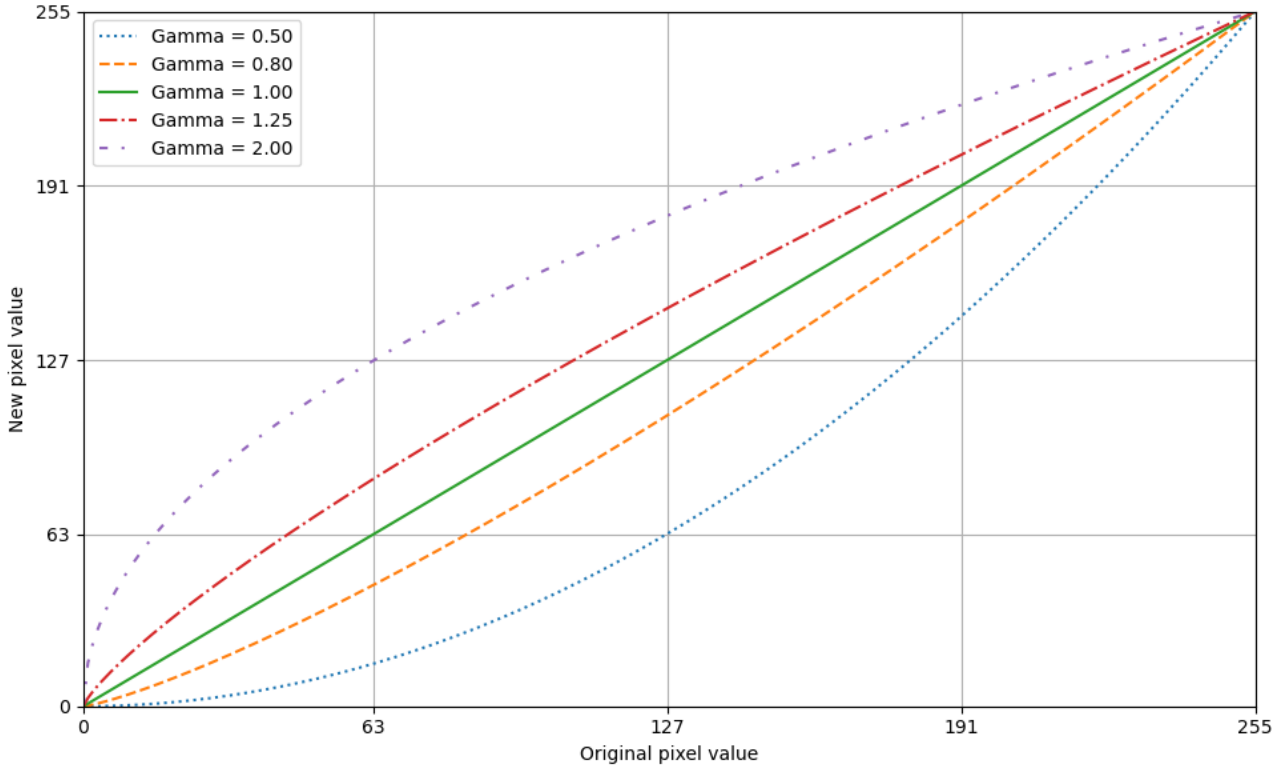
Figure



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

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

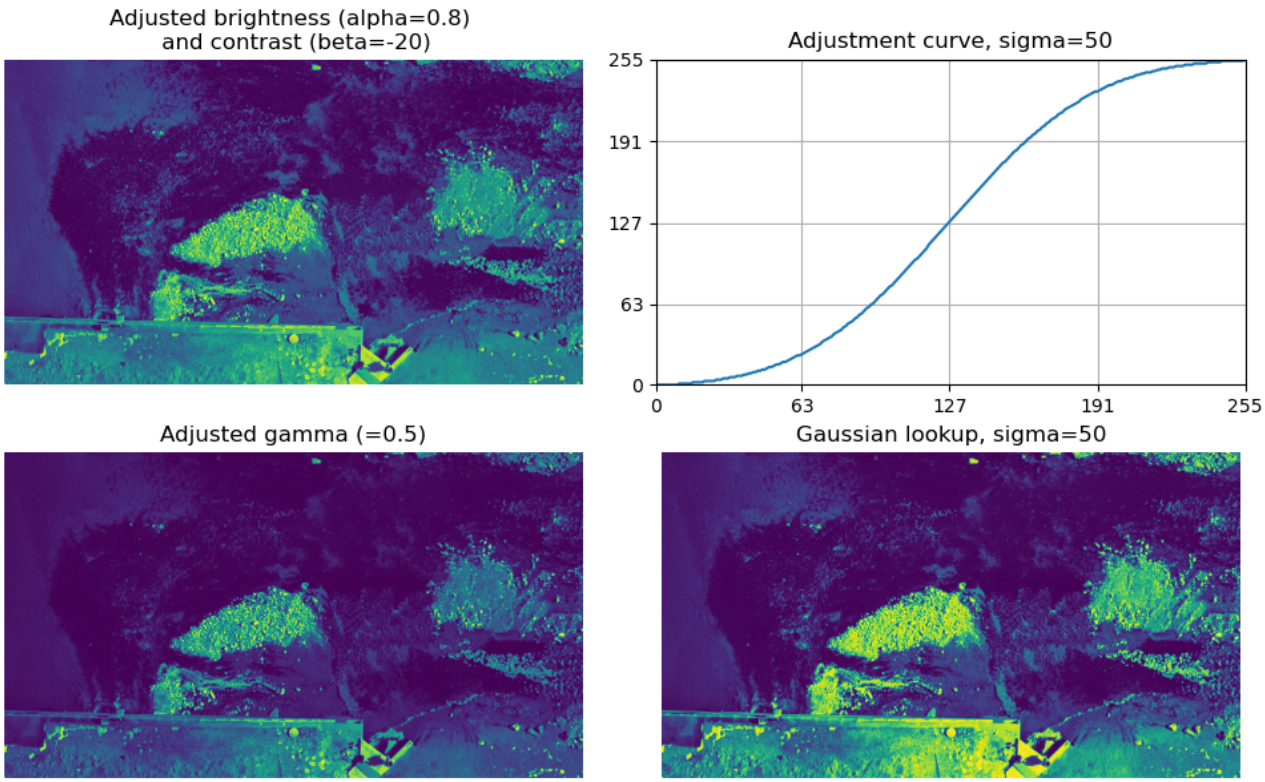
5.5 Gaussian lookup

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.

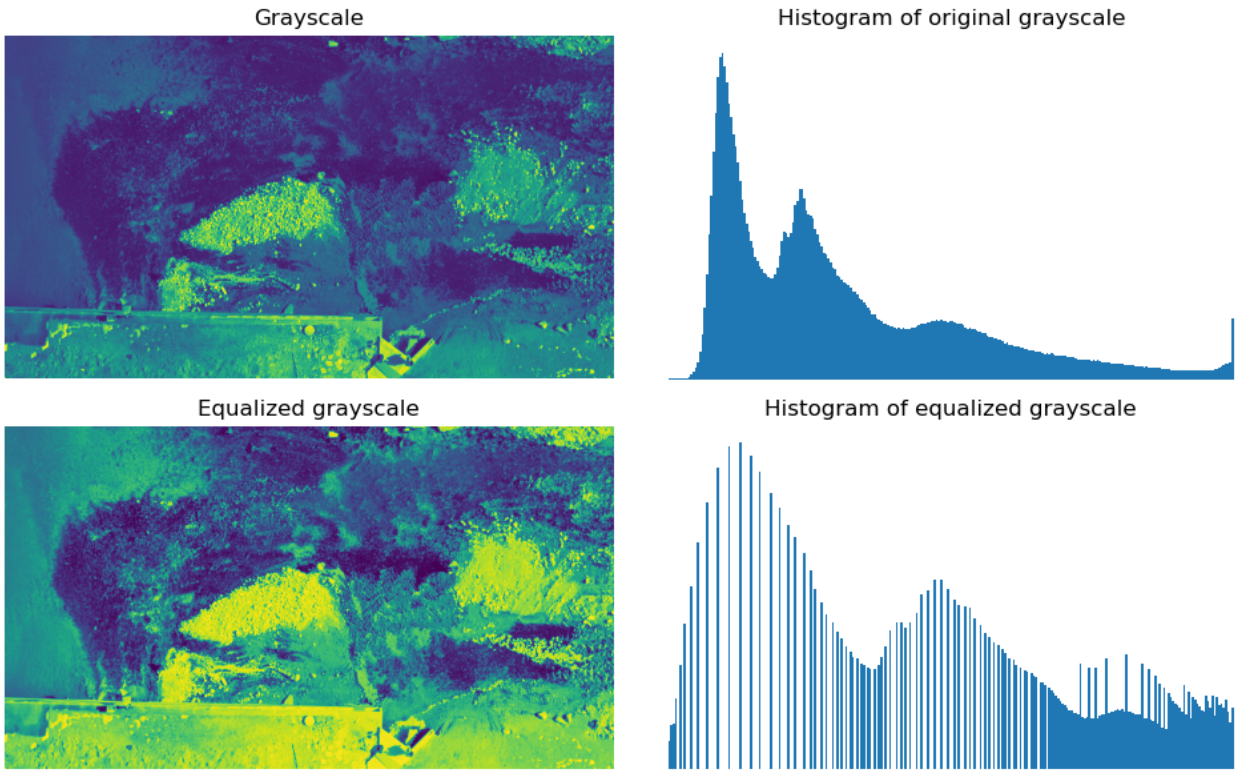
5.6 Histogram equalization

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.

5.7 Contrast-limited adaptive histogram equalization (CLAHE)

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

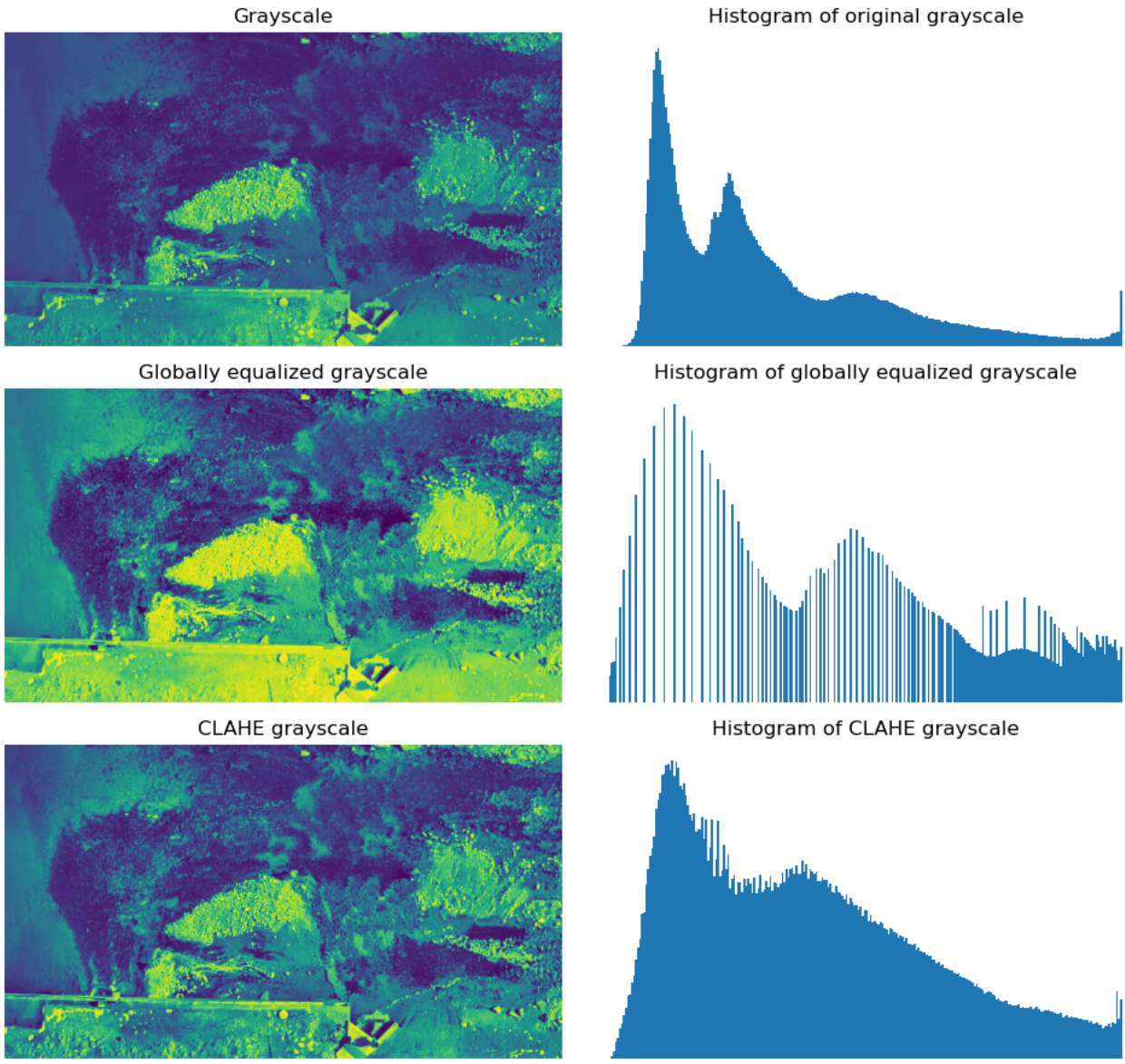
1. Instead of equalizing the histogram of an entire image, we can traverse the image using a smaller window (tile) and equalize the histogram in each window separately, which would increase the local contrast and allow for easier detection of features inside.
2. Limit the image contrast range by removing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

5.8 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.

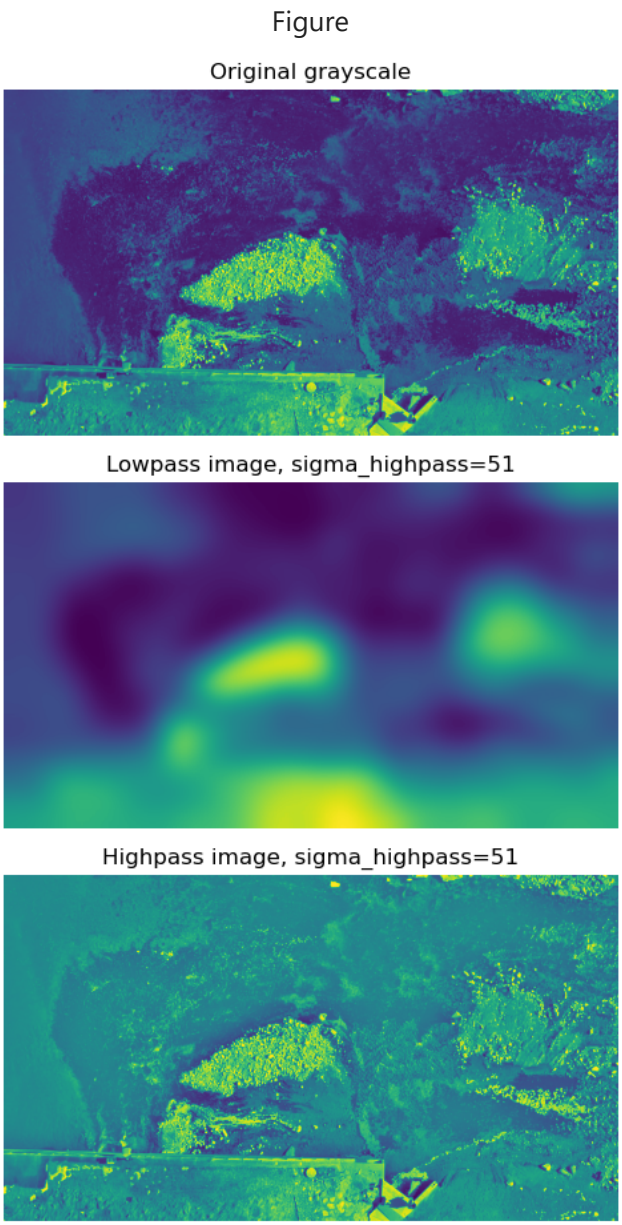
> We have all likely encountered highpass and lowpass filters in our everyday lives - sound reproduction devices use hardware based filters called **audio crossover circuits** which split the audio signal into low frequencies (which are sent to the bass speakers), mid frequencies (which are usually sent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



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.

5.9 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:

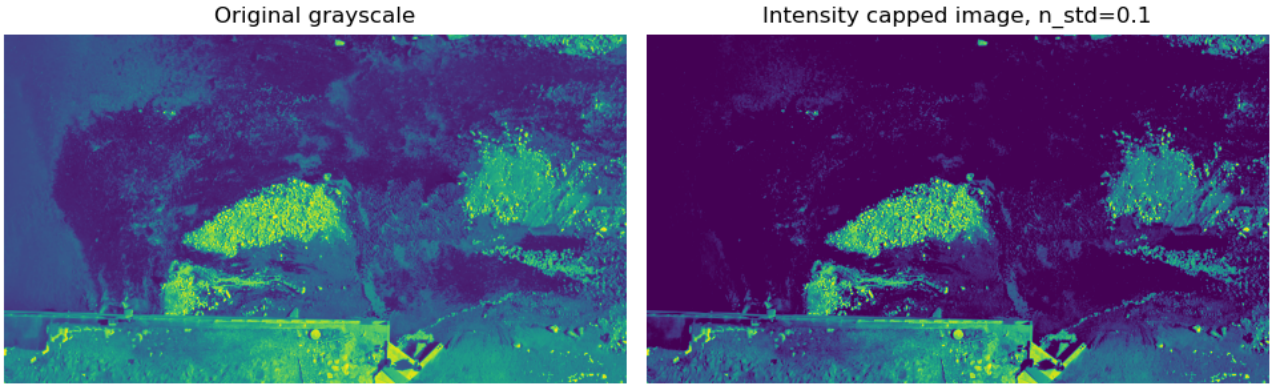
$$Y = \begin{cases} C = N_{cap} \times Y_{median} & \text{if } Y > C \\ Y & \text{else,} \end{cases}$$

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.

5.10 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

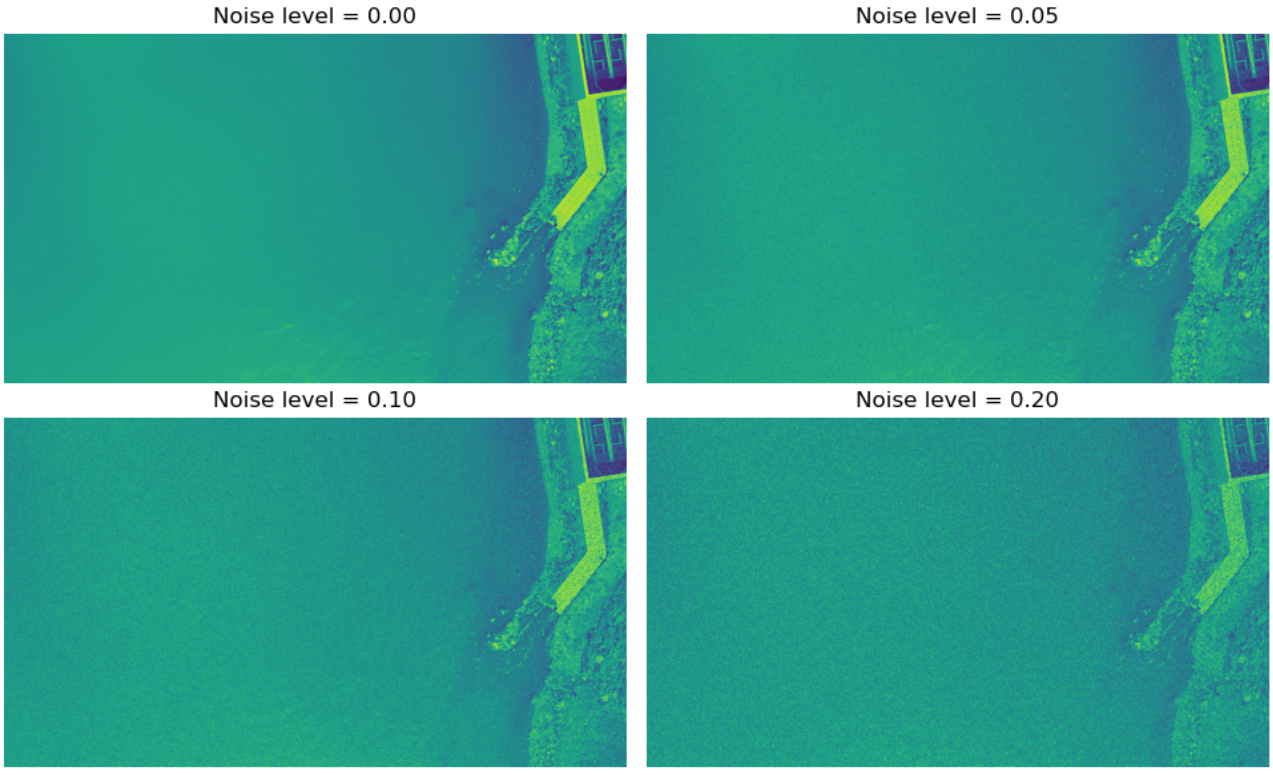


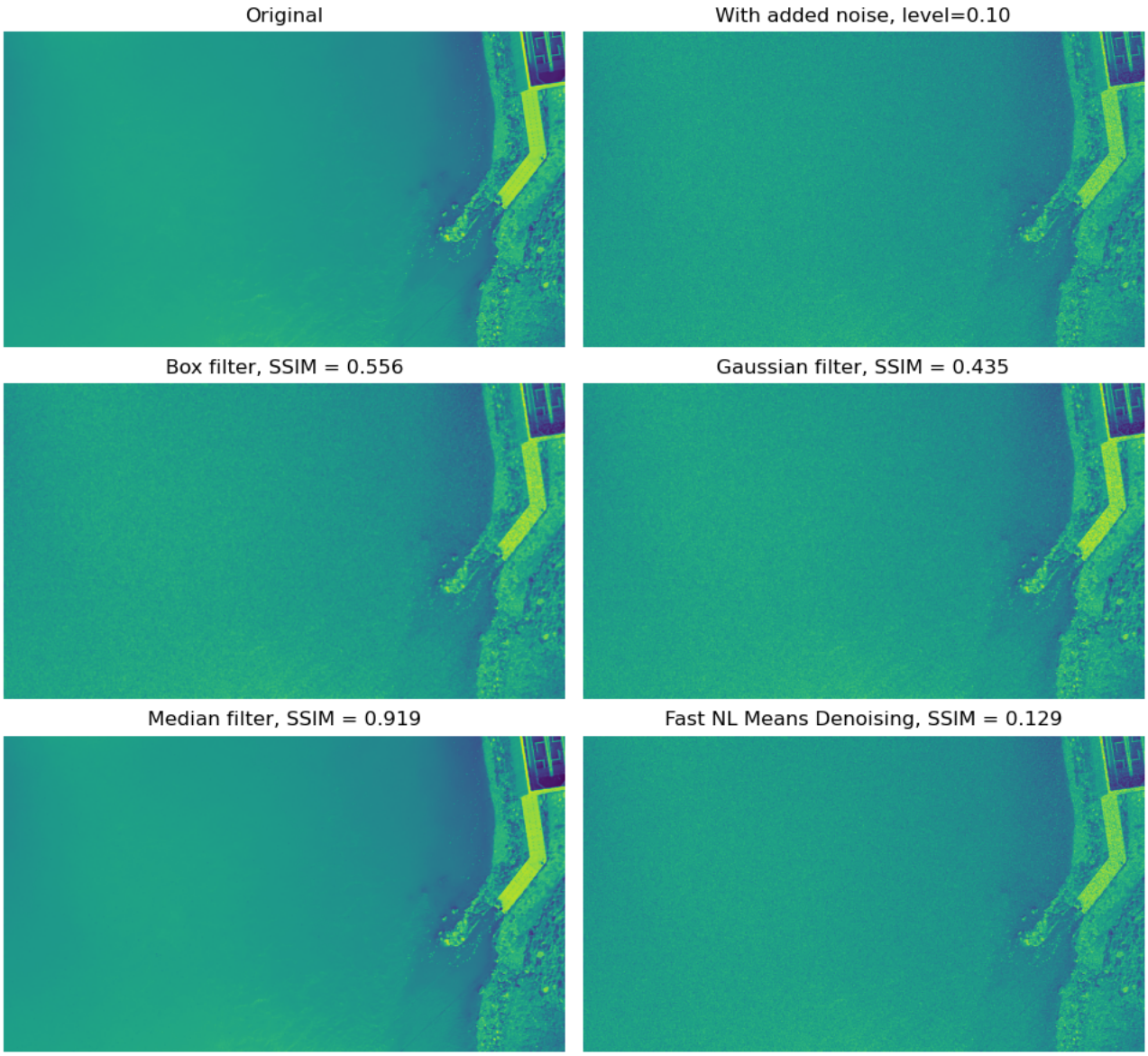
Figure above is meant to simulate what high camera ISO value (overly sensitive camera sensor) does to captured images. Now we can explore several noise removal methods:

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.

5.11 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

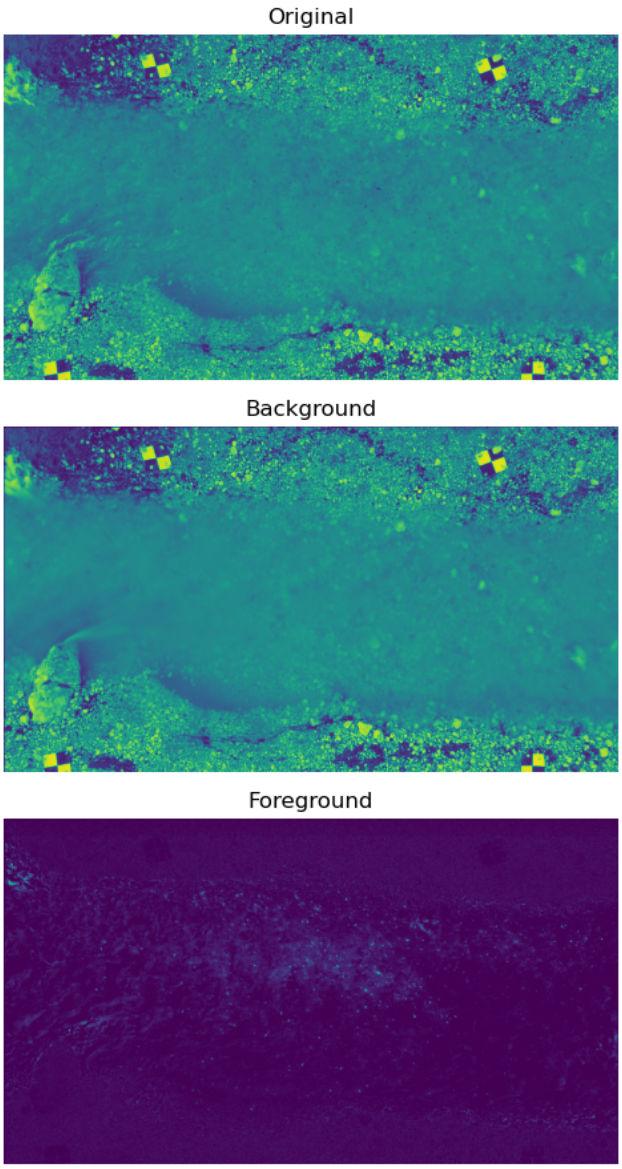
the obtained background. **In order for this to be possible, objects that are static in the real world have to be static in the images, i.e., the image stabilization has to be completed before this procedure.**

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.

5.12 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

<< Back to MAIN notebook

6 Filter stacking

Now that the previous notebooks have covered the basics of the image formats, colorspaces, and filters, we can finally go ahead and perform image enhancement on five sample images presented in the MAIN notebook. To do this, we will inevitably combine different colorspaces, image channels, and filters to create a specific workflow for each example, or an **image enhancement (filtering) stack**.

This notebook is divided into 6 sections: one section concerning a metric for measuring the accentuation of tracer particles, and five sections with examples with proposed enhancement workflows for each of the example images. Each section will aim to explain why a certain procedure is proposed, but the provided strategy will hardly be definitive and only one suitable - feel free to explore (add/remove/modify) different filters and/or adjust filter parameters until you feel satisfied with the final results.

Contents

In [1]:
```
# Necessary libraries
import numpy as np
import matplotlib.pyplot as plt
import cv2
import nbimporter

from image_filtering import lowpass_highpass, intensity_capping, normalize_image, gaussian_lo
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, but
# and [%matplotlib notebook] for Jupyter Notebook
%matplotlib widget
```

6.1 Signal-to-noise ratio

To quantify the accentuation of tracer particles relative to the image background, we can adopt a simple signal-to-noise ratio (SNR) metric, which is the ratio of mean image pixel value (defined mostly by the

background) and their standard deviation (defined mostly by the tracer particles):

In [2]:
```
def snr(a):
    # Scale between 0 and 255
    a = ((a - np.min(a)) / (np.max(a) - np.min(a)) * 255).astype(int)

    return np.mean(a)/np.std(a)
```

Image enhancement strategy should generally aim to minimize the SNR score - to increase the standard deviation of pixel values by making the tracer particles "stand out more" in the image. To properly apply this metric, we should never look at the whole image as it contains many non-relevant regions (riverbanks, islands, etc), but should rather choose certain image regions which only contains particles on water surface (background).

> Formal definition of SNR defines the mean value as the "signal" and deviation as "noise".
> In our case, the "noise" is the target of enhancement.

6.2 Image 1

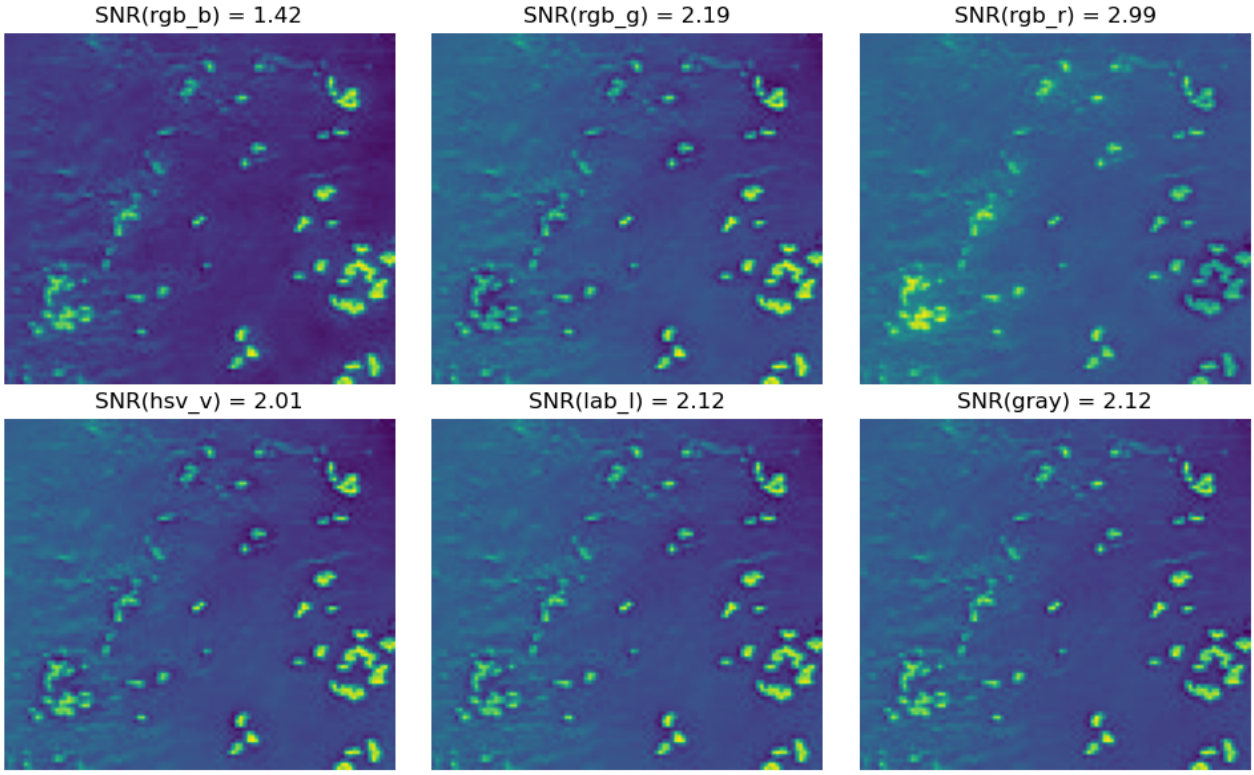
We should first explore the colorspaces using the last code block in the Image colorspaces notebook. By examining the resulting image channels, it is clear from visual inspection that the following channels are viable for image velocimetry: all three RGB channels, HS[V] channel, [L*]a*b* channel, and grayscale model.

We can now use the SNR metric to compare these channels further by selecting a representative region in the image and calculating the corresponding SNR. The selected region, defined by the coordinate range x=1370..1470 and y=685..775, contains both pink and cyan particles, as well as some surface wave features.

In [3]:
```
img_path = './1080p/1.jpg'

# Image subregion selection
(x1, x2), (y1, y2) = (1370, 1470), (685, 775)

# Conversions
img_bgr = cv2.imread(img_path)
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
img_hsv = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2HSV)
img_lab = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2LAB)

# Split into channels
rgb_b, rgb_g, rgb_r = cv2.split(img_bgr)
hsv_h, hsv_s, hsv_v = cv2.split(img_hsv)
lab_l, lab_a, lab_b = cv2.split(img_lab)
gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)

# Viable image channels
channels = [
    rgb_b, rgb_g, rgb_r,
    hsv_v, lab_l, gray
]
```

```
# Prepare a plot
ncols = 3
nrows = np.ceil(len(channels) / ncols)
fig, ax = plt.subplots(nrows=int(nrows), ncols=ncols, figsize=(9.8, 6))

for i, c in enumerate(channels):
    c_name = [k for k, v in locals().items() if v is c][0]
    subimg = channels[i][y1: y2, x1:x2]
    ax[i//ncols, i % ncols].imshow(subimg)
    ax[i//ncols, i % ncols].set_title('SNR({}) = {:.2f}'.format(c_name, snr(subimg)))

[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



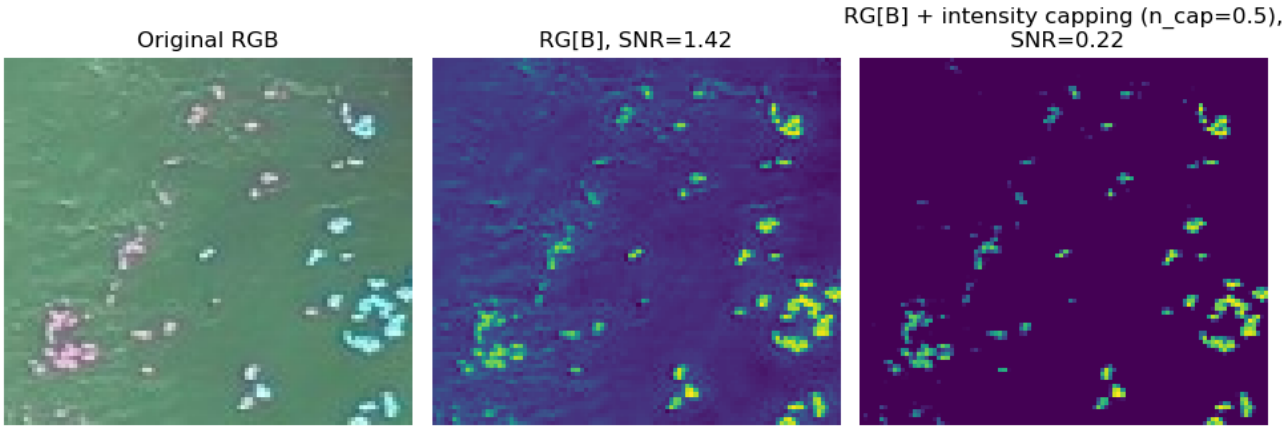
The results of the colorspace analysis indicate that the blue channel from RGB model has the highest SNR value in the selected region, and this channel will be used in the remainder of the analysis.

Since both the pink and cyan tracer particles are already well accentuated, we mostly want to remove the surface wave features, we can achieve this by pixel applying intensity capping:

In [4]:
```
fig, ax = plt.subplots(nrows=1, ncols=3, figsize=(9.8, 3.6))

ax[0].imshow(img_rgb[y1: y2, x1:x2])
ax[0].set_title('Original RGB')

ax[1].imshow(rgb_b[y1: y2, x1:x2])
ax[1].set_title('RG[B], SNR={:.2f}'
                .format(snr(rgb_b[y1: y2, x1:x2])))

# Accentuate light particles on darker background from blue channel
```

```
n_cap = 0.5
img1_cap = intensity_capping(rgb_b, n_std=n_cap, mode='LoD')

ax[2].imshow(img1_cap[y1: y2, x1:x2])
ax[2].set_title('RG[B] + intensity capping (n_cap={:.1f}),\n SNR={:.2f}'
                .format(n_cap, snr(img1_cap[y1: y2, x1:x2])))

[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

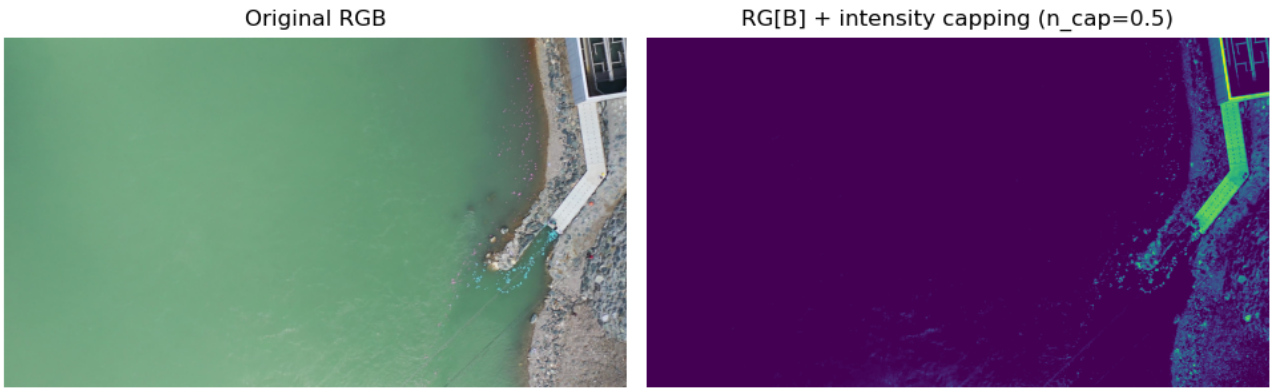


The resulting SNR is far lower than that of the raw blue channel, and surface wave features are almost completely removed. We can increase the capping parameter `n_cap` to further filter such features, but one should also be careful to not also lose information about tracer particles as well - finding a right balance is often a matter of experience through trial and error.

Finally, we should also present and compare the RGB blue channel and the capped image to verify that the desired features are accentuated across the entire image:

In [5]:
```
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(img1_cap)
ax[1].set_title('RG[B] + intensity capping (n_cap={:.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



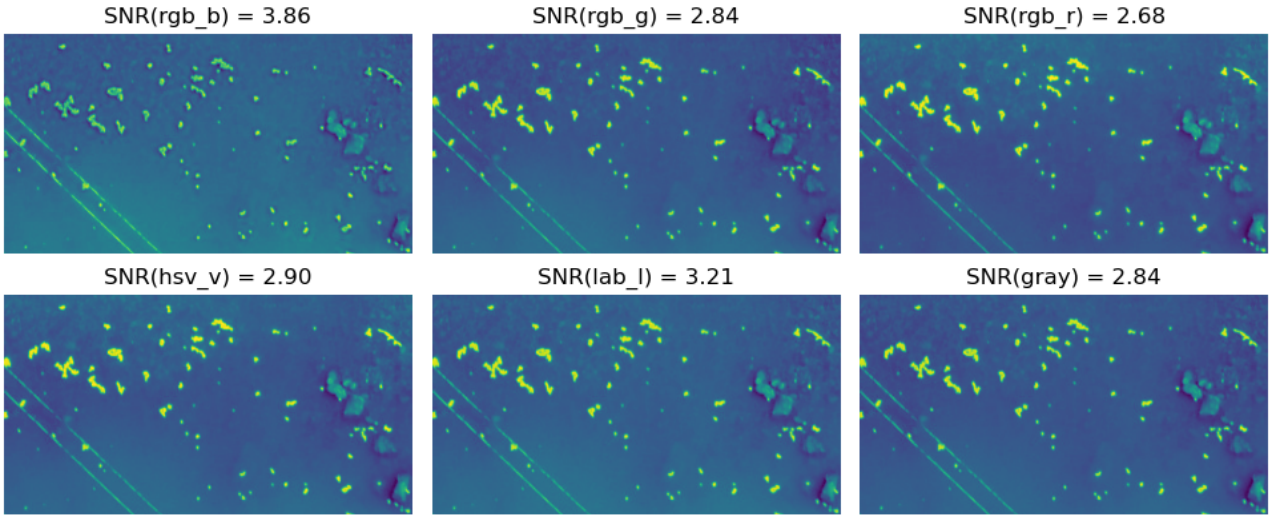
6.3 Image 2

Second example is similar to **Image 1**, but contains some regions with visual disturbances such as visible riverbed, powerlines, chroma changes near riverbanks, etc. As a representative region, we can select the top-left area on the water surface close to the riverbank. Same as with **Image 1**, viable colorspace models and channels are: all three RGB channels, HS[V] channel, [L*]a*b* channel, and grayscale model. For this example, RGB red channel has the highest SNR value in the selected region, but will fail to properly present cyan-colored particles. Because of this, we can select grayscale model for further enhancement.

In [6]:
```
img_path = './1080p/2.jpg'

# Image subregion selection
(x1, x2), (y1, y2) = (0, 335), (350, 530)

# Conversions
img_bgr = cv2.imread(img_path)
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
img_hsv = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2HSV)
img_lab = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2LAB)

# Split into channels
rgb_b, rgb_g, rgb_r = cv2.split(img_bgr)
hsv_h, hsv_s, hsv_v = cv2.split(img_hsv)
lab_l, lab_a, lab_b = cv2.split(img_lab)
gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)

# Viable image channels
channels = [
    rgb_b, rgb_g, rgb_r,
    hsv_v, lab_l, gray
]

ncols = 3
nrows = np.ceil(len(channels) / ncols)
fig, ax = plt.subplots(nrows=int(nrows), ncols=ncols, figsize=(9.8, 4.2))

for i, c in enumerate(channels):
    c_name = [k for k, v in locals().items() if v is c][0]
    subimg = channels[i][y1: y2, x1:x2]
```

```
    ax[i//ncols, i % ncols].imshow(subimg)
    ax[i//ncols, i % ncols].set_title('SNR({}) = {:.2f}'
                                      .format(c_name, snr(subimg)))

[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

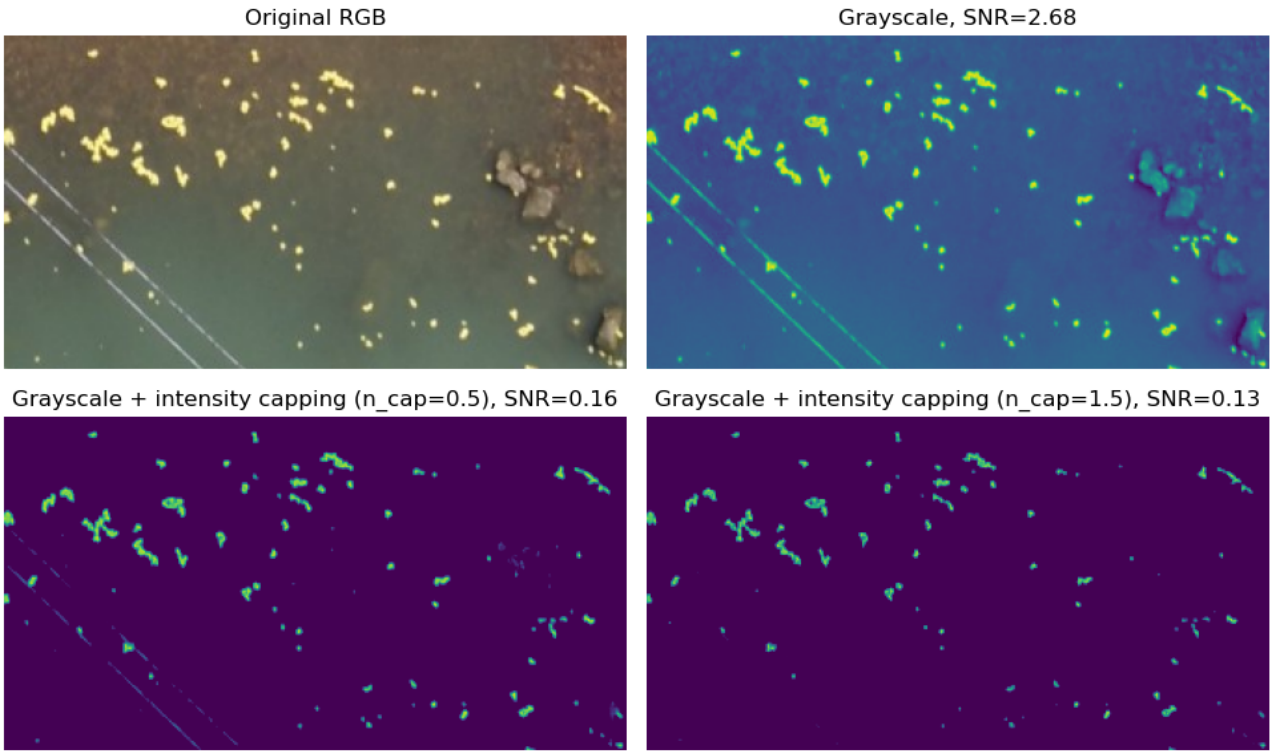


Same as with **Image 1**, we can try to apply pixel value intensity capping to accentuate the tracers on the water surface. As a side effect, intensity capping can also be successfully used to remove some of the riverbed features (right side in images below), as well as to remove powerlines from the bottom-left corner when the intensity capping parameter is high enough (N > 1.0):

In [7]:
```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(9.8, 6.0))

ax[0][0].imshow(img_rgb[y1: y2, x1:x2])
ax[0][0].set_title('Original RGB')

ax[0][1].imshow(gray[y1: y2, x1:x2])
ax[0][1].set_title('Grayscale, SNR={:.2f}'.format(snr(rgb_r[y1: y2, x1:x2])))

# Intensity capping
n_cap = 0.5
img2_cap1 = intensity_capping(gray, n_std=n_cap, mode='LoD')

ax[1][0].imshow(img2_cap1[y1: y2, x1:x2], vmin=0, vmax=255)
ax[1][0].set_title('Grayscale + intensity capping (n_cap={:.1f}), SNR={:.2f}'
                   .format(n_cap, snr(img2_cap1[y1: y2, x1:x2])))

# Intensity capping
n_cap = 1.5
img2_cap2 = intensity_capping(gray, n_std=n_cap, mode='LoD')

ax[1][1].imshow(img2_cap2[y1: y2, x1:x2], vmin=0, vmax=255)
ax[1][1].set_title('Grayscale + intensity capping (n_cap={:.1f}), SNR={:.2f}'
                   .format(n_cap, snr(img2_cap2[y1: y2, x1:x2])))
```

```
[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()
```

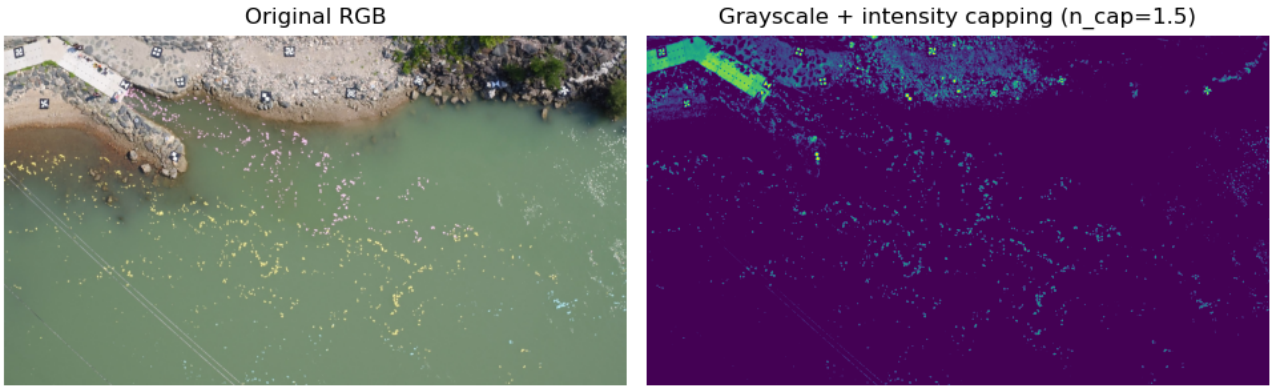
Figure



Finally, we can inspect the entire image to make sure that the tracer particles are presented correctly:

In [8]:
```
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(img2_cap2)
ax[1].set_title('Grayscale + intensity capping (n_cap={:.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



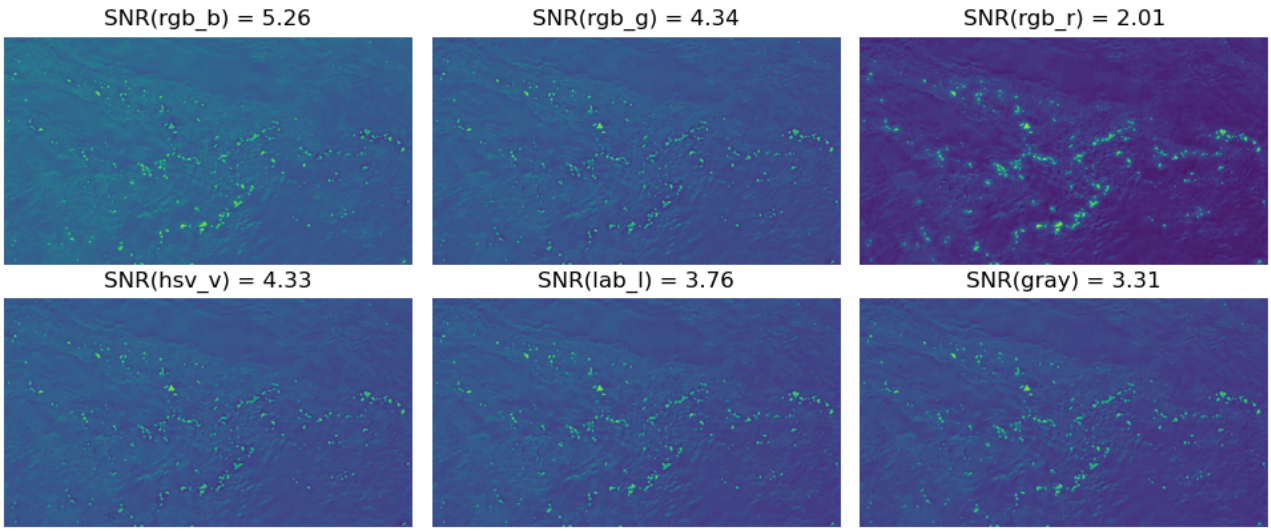
6.4 Image 3

Image 3 represents the same area as in **Image 2**, but the background contains significantly more surface wave features and light reflections. Perhaps the most intuitive reaction to the appearance of additional visual disturbances would be to increase ("strengthen") the intensity capping parameter, but it would be the wrong one. Addition of background visual disturbances actually increases the standard deviation of pixel intensities, so the correct response is more likely to reduce the capping parameter.

In [9]:
```
img_path = './1080p/3.jpg'

# Image subregion selection
(x1, x2), (y1, y2) = (1130, 1560), (670, 910)

# Conversions
img_bgr = cv2.imread(img_path)
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
img_hsv = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2HSV)
img_lab = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2LAB)

# Split into channels
rgb_b, rgb_g, rgb_r = cv2.split(img_bgr)
hsv_h, hsv_s, hsv_v = cv2.split(img_hsv)
lab_l, lab_a, lab_b = cv2.split(img_lab)
gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)

# Viable image channels
channels = [
    rgb_b, rgb_g, rgb_r,
    hsv_v, lab_l, gray
]

ncols = 3
nrows = np.ceil(len(channels) / ncols)
fig, ax = plt.subplots(nrows=int(nrows), ncols=ncols, figsize=(9.8, 4.2))

for i, c in enumerate(channels):
    c_name = [k for k, v in locals().items() if v is c][0]
    subimg = channels[i][y1: y2, x1:x2]
    ax[i//ncols, i % ncols].imshow(subimg)
    ax[i//ncols, i % ncols].set_title('SNR({}) = {:.2f}'
```

```
                                    .format(c_name, snr(subimg)))

[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

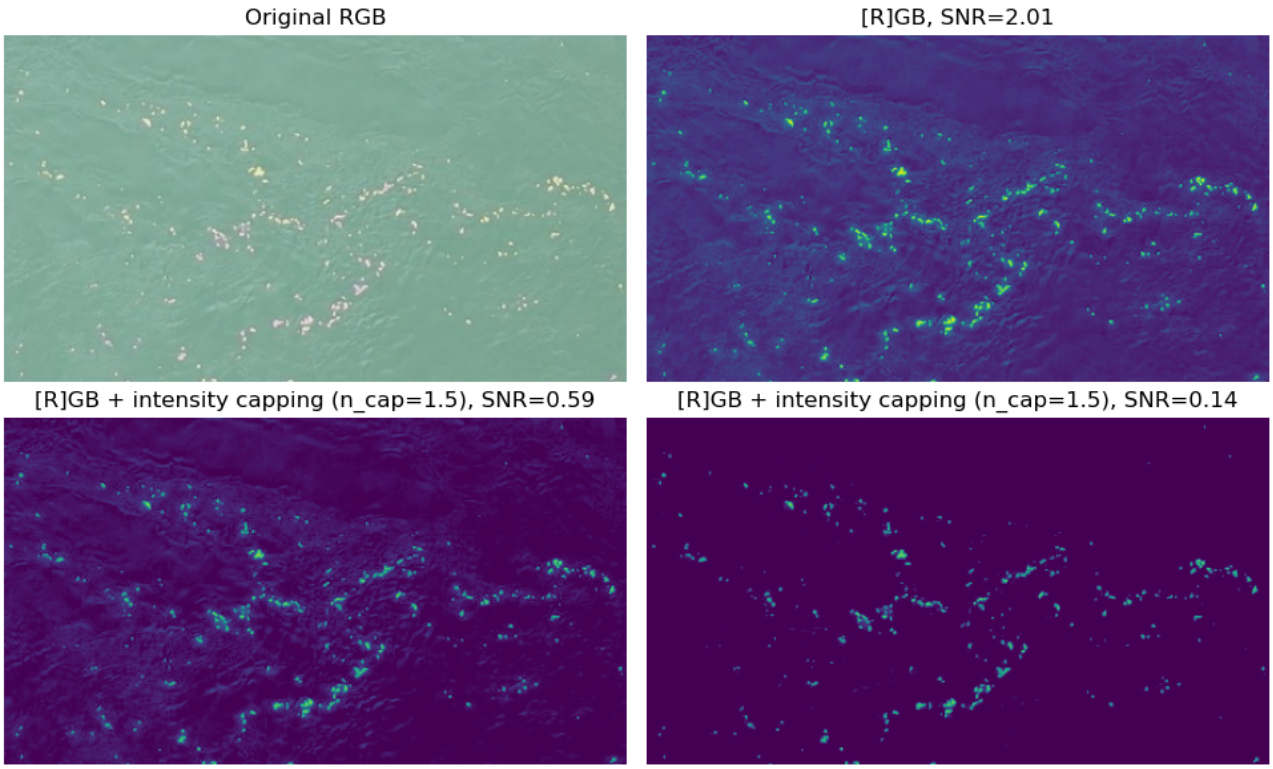


The [R]GB channel has considerably more contrast than the remaining options and will be used further on. This will, however, limit the applicability of the enhanced images for velocimetry using only yellow- and red-colored tracer particles. We can capture cyan-colored tracer particles using the same procedure in RG[B] channel. To capture all of the tracer particles within a single image, we can superimpose the resulting [R]GB and RG[B] components:

In [10]:
```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(9.8, 6.0))

ax[0][0].imshow(img_rgb[y1: y2, x1:x2])
ax[0][0].set_title('Original RGB')

ax[0][1].imshow(rgb_r[y1: y2, x1:x2])
ax[0][1].set_title('[R]GB, SNR={:.2f}'.format(snr(rgb_r[y1: y2, x1:x2])))

# Intensity capping in [R]GB
n_cap_r = 0.0
img3_cap1 = intensity_capping(rgb_r, n_std=n_cap_r, mode='LoD')

ax[1][0].imshow(img3_cap1[y1: y2, x1:x2], vmin=0, vmax=255)
ax[1][0].set_title('[R]GB + intensity capping (n_cap={:.1f}), SNR={:.2f}'
                   .format(n_cap, snr(img3_cap1[y1: y2, x1:x2])))

# Intensity capping in [R]GB
n_cap_r = 0.5
img3_cap2 = intensity_capping(rgb_r, n_std=n_cap_r, mode='LoD')

ax[1][1].imshow(img3_cap2[y1: y2, x1:x2], vmin=0, vmax=255)
ax[1][1].set_title('[R]GB + intensity capping (n_cap={:.1f}), SNR={:.2f}'
                   .format(n_cap, snr(img3_cap2[y1: y2, x1:x2])))

# Intensity capping RG[B]
```

```
n_cap_b = 1.0
img3_cap3 = intensity_capping(rgb_b, n_std=n_cap_b, mode='LoD')

[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()
```

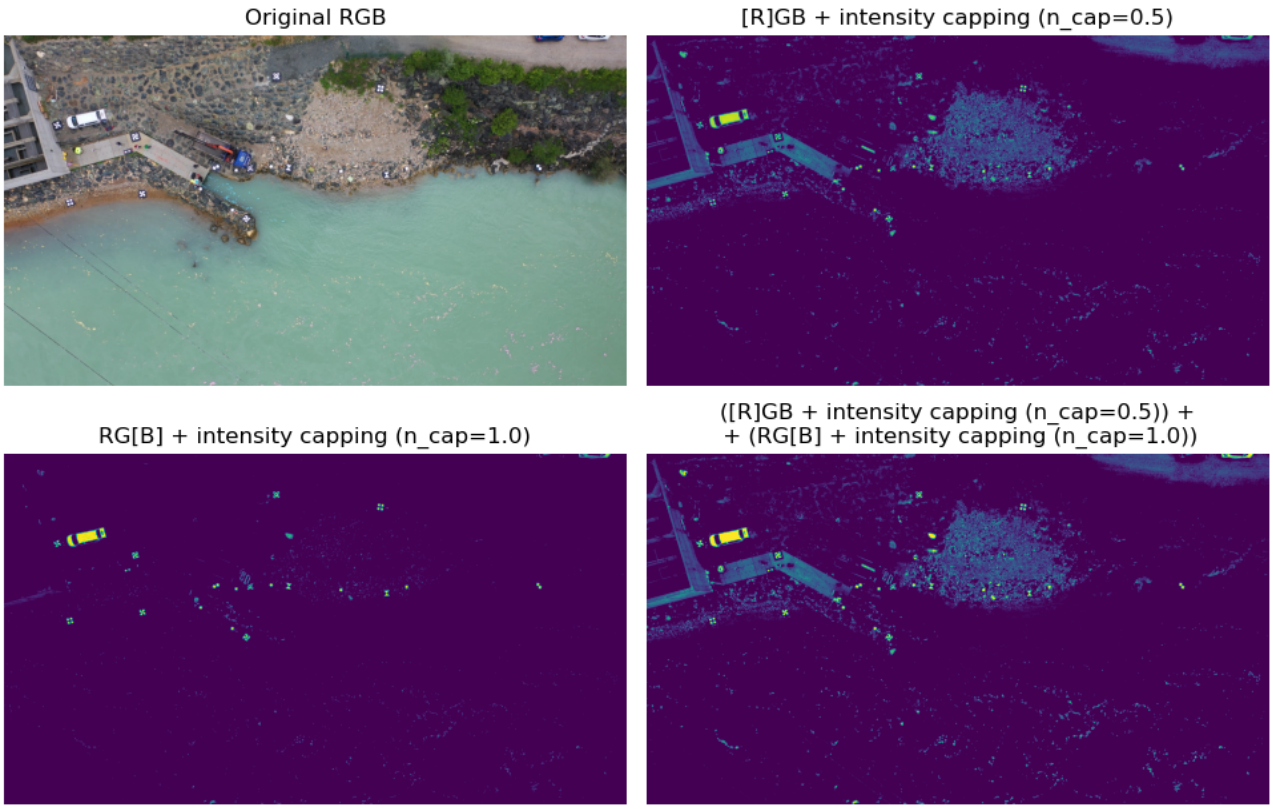
Figure



Inspect the whole image, and pay close attention to the region around [730, 490]:

In [11]:
```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(9.8, 6.4))

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

ax[0][1].imshow(img3_cap2)
ax[0][1].set_title('[R]GB + intensity capping (n_cap={:.1f})'.format(n_cap_r))

ax[1][0].imshow(img3_cap3)
ax[1][0].set_title('RG[B] + intensity capping (n_cap={:.1f})'.format(n_cap_b))

ax[1][1].imshow(cv2.add(img3_cap2, img3_cap3))
ax[1][1].set_title('([R]GB + intensity capping (n_cap={:.1f})) +'\
                   '\n + (RG[B] + intensity capping (n_cap={:.1f}))'
                   .format(n_cap_r, n_cap_b))

[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 idea of superimposing data from different image channels is quite powerful when tracers of different color are used for seeding. An alternative approach could be to perform image velocimetry using several sets of images (for each of the colors) and superimpose the velocity data, but such approach will consume more time and storage space.

6.5 Image 4

The following example is likely the most complex one in this report. **Image 4** contains magenta- and cyan-colored particles in various seeding densities across the water surface, visible riverbed of various shades of green color, surface waves and light reflections, etc. Unlike previous examples, this image cannot be tackled by applying a single filter. Therefore, we can try the following:

1. **Select the appropriate image channel** based on visual inspection and SNR metric,
2. Remove as many background features such as riverbed using a **highpass filter**,
3. Apply **intensity capping** to remove the remaining background features.

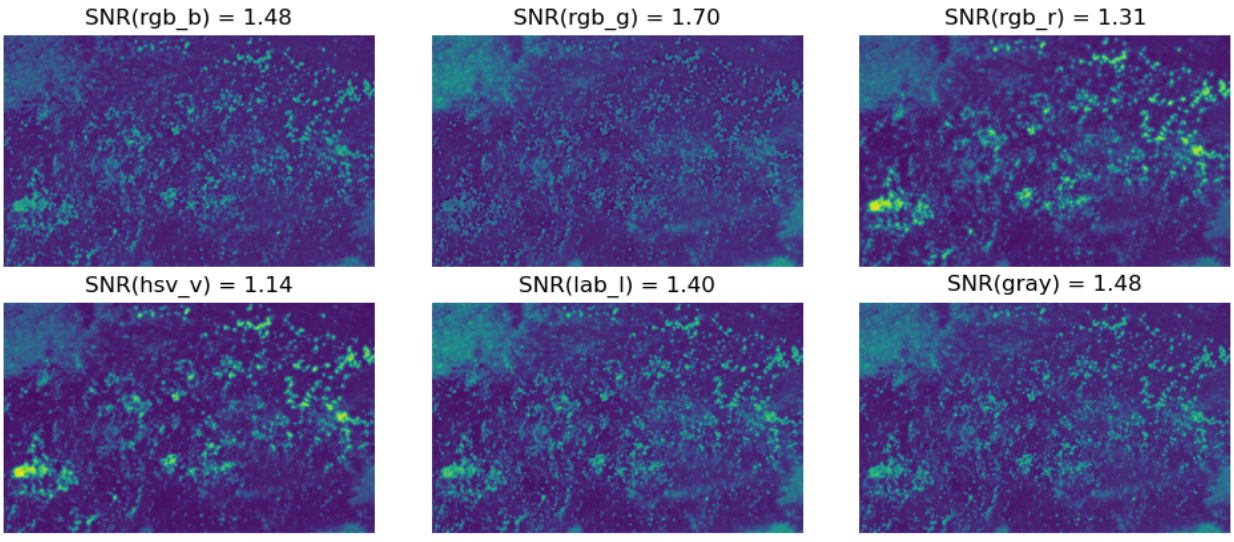
This procedure is adequate, but not the only possible, as we'll show later on.

First we should analyze the viable colorspace channels:

In [12]:
```
img_path = './1080p/4.jpg'

# Image subregion selection
(x1, x2), (y1, y2) = (560, 930), (150, 380)

# Conversions
```

```
img_bgr = cv2.imread(img_path)
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
img_hsv = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2HSV)
img_lab = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2LAB)

# Split into channels
rgb_b, rgb_g, rgb_r = cv2.split(img_bgr)
hsv_h, hsv_s, hsv_v = cv2.split(img_hsv)
lab_l, lab_a, lab_b = cv2.split(img_lab)
gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)

# Viable image channels
channels = [
    rgb_b, rgb_g, rgb_r,
    hsv_v, lab_l, gray
]

ncols = 3
nrows = np.ceil(len(channels) / ncols)
fig, ax = plt.subplots(nrows=int(nrows), ncols=ncols, figsize=(9.8, 4.2))

for i, c in enumerate(channels):
    c_name = [k for k, v in locals().items() if v is c][0]
    subimg = channels[i][y1: y2, x1:x2]
    ax[i//ncols, i % ncols].imshow(subimg)
    ax[i//ncols, i % ncols].set_title('SNR({}) = {:.2f}'
                                      .format(c_name, snr(subimg)))

[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

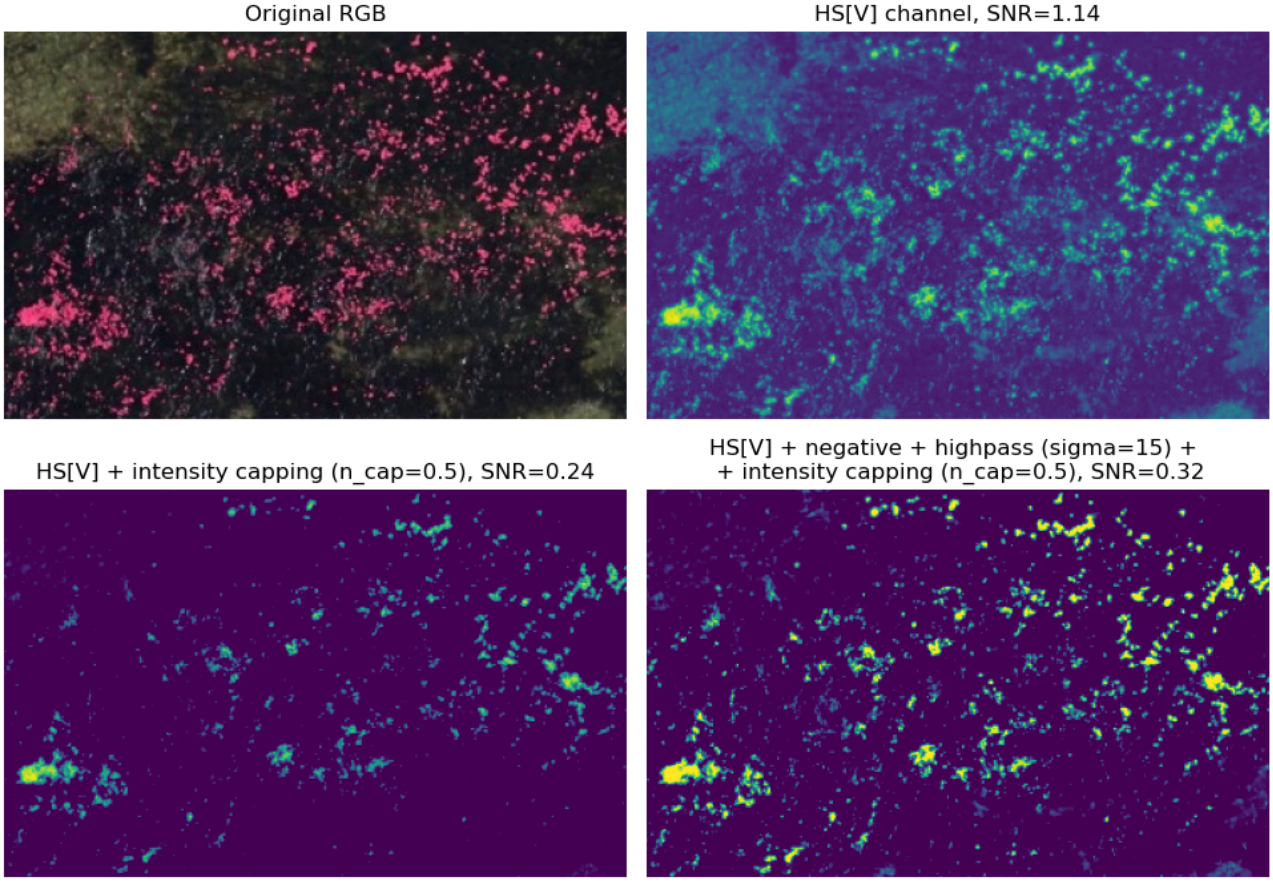


Now we can try **highpass filter** and **intensity capping** to remove as much background features as possible without losing too much information about tracer particles:

In [13]:
```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(9.8, 7.0))

ax[0][0].imshow(img_rgb)
```

```
ax[0][0].set_title('Original RGB')

ax[0][1].imshow(hsv_v)
ax[0][1].set_title('HS[V] channel, SNR={:.2f}'.format(snr(hsv_v[y1: y2, x1:x2])))

# Intensity capping
n_cap = 0.5
img4_v1 = intensity_capping(hsv_v, n_std=n_cap, mode='LoD')

ax[1][0].imshow(img4_v1, vmin=0, vmax=255)
ax[1][0].set_title('HS[V] + intensity capping (n_cap={:.1f}), SNR={:.2f}'
                   .format(n_cap, snr(img4_v1[y1: y2, x1:x2])))

# Intensity capping
sigma = 15
n_cap = 0.5
img4_v2 = lowpass_highpass(hsv_v, sigma=sigma)[1]
img4_v2 = intensity_capping(img4_v2, n_std=n_cap, mode='LoD')

ax[1][1].imshow(img4_v2, vmin=0, vmax=255)
ax[1][1].set_title('HS[V] + negative + highpass (sigma={}) + \n + intensity capping (n_cap={:
                   .format(sigma, n_cap, snr(img4_v2[y1: y2, x1:x2])))

[a.axis('off') for a in ax.reshape(-1)]
[a.set_xlim([x1, x2]) for a in ax.reshape(-1)]
[a.set_ylim([y2, y1]) 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



As evident from the figure above, when riverbed is quite visible with pixel intensities similar to those of the tracer particles, intensity capping on its own is not adequate for particle accentuation - there is a significant and unacceptable tradeoff between riverbed filtering and particle accentuation. A workaround is to first remove low frequency content from the image, which riverbed features generally belong to. Afterwards, intensity capping will likely provide better results. However, while it would appear that the highpass filter slightly increases the SNR, and thus worsens the particle accentuation, visual inspection of images (especially comparison with the original RGB) indicates otherwise. This also leads to the conclusion about SNR metric:

> **Warning:** Signal-to-noise metric is not a perfect indicator of the tracer particle accentuation and should only be used as a loose guide during the enhancement process. **VISUAL INSPECTION should always dictate of the image enhancement workflow**.

There is one more, albeit exotic, approach available for images such as these. In the notebook on **Image colorspaces** we've discussed the L*a*b* model and how it manipulates colors in ranges between red and green (L*[a*]b*), and blue and yellow (L*a*[b*]). However, L*[a*]b* and L*a*[b*] suffer from one significant limitation - they will always appear relatively blurry compared to the [L*]a*b* and the original image. There is an interesting but obscure manipulation available that can help properly accentuate particles of red, green, blue, and yellow color - we can rearrange the channels of the L*a*b* model as follows:

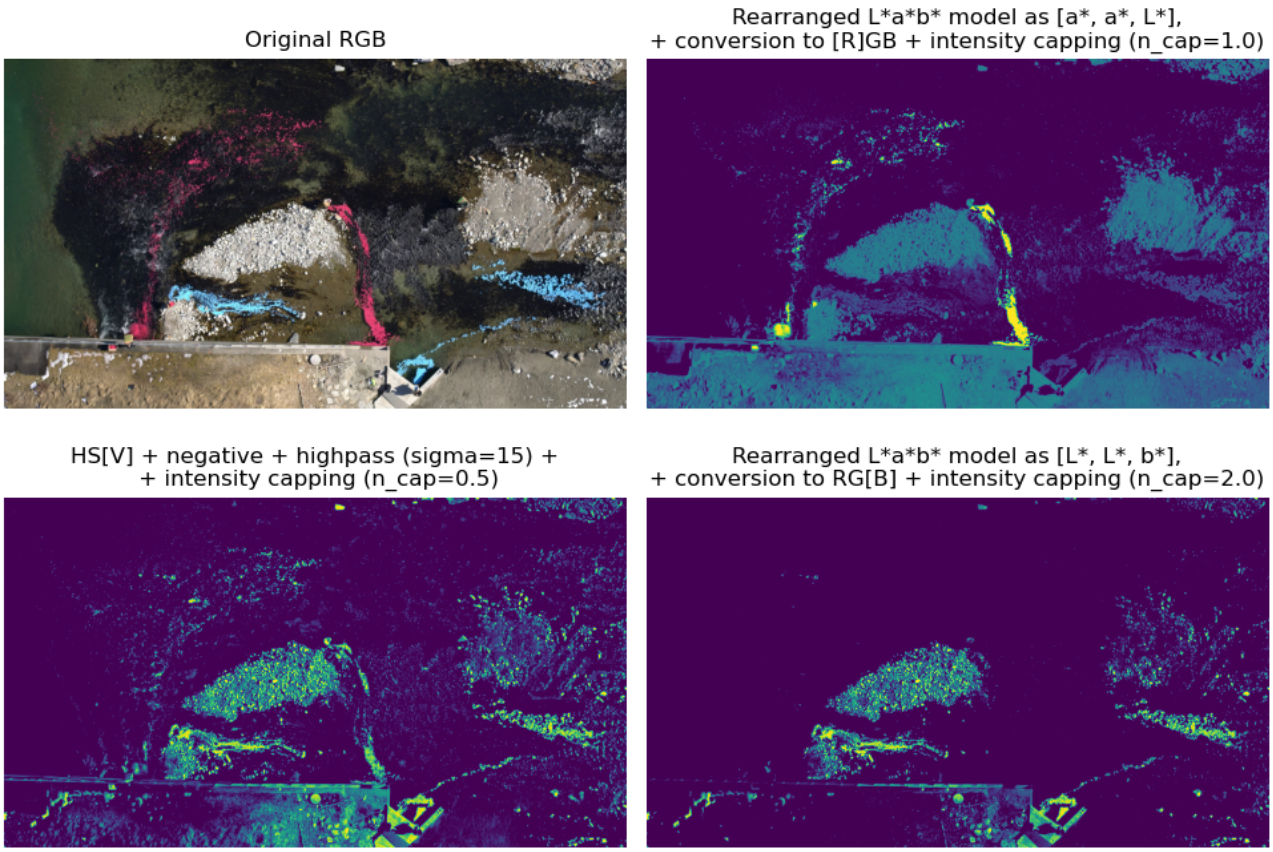
1. For the accentuation of red particles = arrangement (a*, a*, L*),
2. For the accentuation of green particles = arrangement (~a*, ~a*, L*), where ~ operator denotes image negative,
3. For the accentuation of cyan particles = arrangement (L*, L*, b*),
4. ...

The idea is usually (but not always) to put the [L*]a*b* in the place of a channel that is not representing the targeted color, and the channel with the targeted color in place of the [L*]a*b*. The only thing left then is to convert from L*a*b* to RGB/BGR and find the adequate subchannel which best represents the targeted features (trial and error). You can see it in action wit the code below, where we will also apply only the intensity capping (make sure to read the code comments for additional info):

In [14]:
```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(9.8, 7.0))

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

ax[1][0].imshow(img4_v2, vmin=0, vmax=255)
ax[1][0].set_title('HS[V] + negative + highpass (sigma={}) +' \
                   '\n + intensity capping (n_cap={:.1f})'
                   .format(sigma, n_cap))

# Targeting MAGENTA tracer particles ========================================

# Change the order and structure of L*a*b* model
lab_rearranged = cv2.merge([lab_a, lab_a, lab_l])

# Convert to RGB and take the first (red) channel
```

```
red_rearranged = cv2.cvtColor(lab_rearranged, cv2.COLOR_LAB2RGB)[:, :, 0]

# Apply some intensity capping
n_cap = 1.0
red_rearranged_cap = intensity_capping(red_rearranged, n_std=n_cap, mode='LoD')

ax[0][1].imshow(red_rearranged_cap, vmin=0, vmax=255)
ax[0][1].set_title('Rearranged L*a*b* model as [a*, a*, L*],'\
                   '\n+ conversion to [R]GB + intensity capping (n_cap={:.1f})'
                   .format(n_cap))

# ------------------------------------------------------------------------

# Targeting CYAN tracer particles ============================================

lab_rearranged = cv2.merge([lab_l, lab_l, lab_b])

# Convert to RGB and take the third (blue) channel
blue_rearranged = cv2.cvtColor(lab_rearranged, cv2.COLOR_LAB2RGB)[:, :, 2]

n_cap = 2.0
blue_rearranged_cap = intensity_capping(blue_rearranged, n_std=n_cap, mode='LoD')

ax[1][1].imshow(blue_rearranged_cap, vmin=0, vmax=255)
ax[1][1].set_title('Rearranged L*a*b* model as [L*, L*, b*],'\
                   '\n+ conversion to RG[B] + intensity capping (n_cap={:.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



As evident from the figure above, compared to simply taking the HS[V] channel, we can target particular tracer particle colors using L*a*b* model and obtain results that are far "cleaner" - fewer "parasitic" features and visual disturbances, fewer waves and light reflections, etc. This method allows for more controlled targeting of color key and chroma, but requires a bit more experimentation, especially when rearanging the channels of the L*a*b* model.

Tip: The resulting images in the previous figure also contain visible static features such as islands and some light reflection of surface waves. While these are usually undesirable, sometimes they cannot be removed by an automated procedure. However, if the video is stabilized such features will appear motionless and can easily be filtered by velocity magnitude after the image velocimetry step. Alternatively, images filtered in this way can also be subjected to background removal.

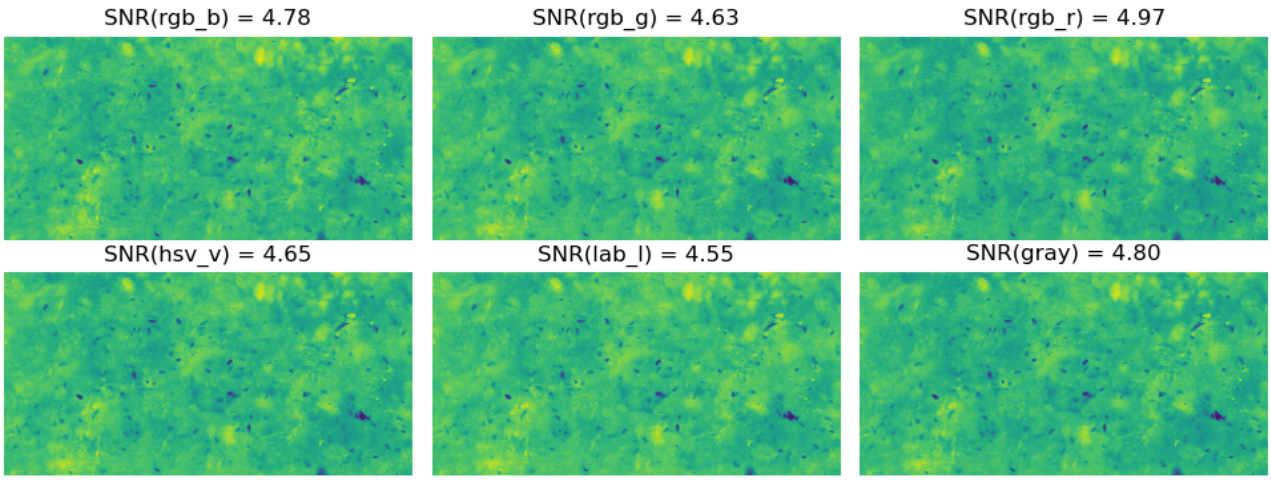
6.6 Image 5

The final example in this report is also somewhat unusual as the tracer particles are darker than the water surface, and the flow is supercritical. Same as with all the previous examples, the three RGB channels, HS[V] and [L*]a*b* are viable options for enhancement:

In [15]:
```
img_path = './1080p/5.jpg'

# Image subregion selection
(x1, x2), (y1, y2) = (820, 1220), (340, 540)
```

```
# Conversions
img_bgr = cv2.imread(img_path)
img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
img_hsv = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2HSV)
img_lab = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2LAB)

# Split into channels
rgb_b, rgb_g, rgb_r = cv2.split(img_bgr)
hsv_h, hsv_s, hsv_v = cv2.split(img_hsv)
lab_l, lab_a, lab_b = cv2.split(img_lab)
gray = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2GRAY)

# Viable image channels
channels = [
    rgb_b, rgb_g, rgb_r,
    hsv_v, lab_l, gray
]

ncols = 3
nrows = np.ceil(len(channels) / ncols)
fig, ax = plt.subplots(nrows=int(nrows), ncols=ncols, figsize=(9.8, 3.8))

for i, c in enumerate(channels):
    c_name = [k for k, v in locals().items() if v is c][0]
    subimg = channels[i][y1: y2, x1:x2]
    ax[i//ncols, i % ncols].imshow(subimg)
    ax[i//ncols, i % ncols].set_title('SNR({}) = {:.2f}'
                                      .format(c_name, snr(~subimg)))

[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



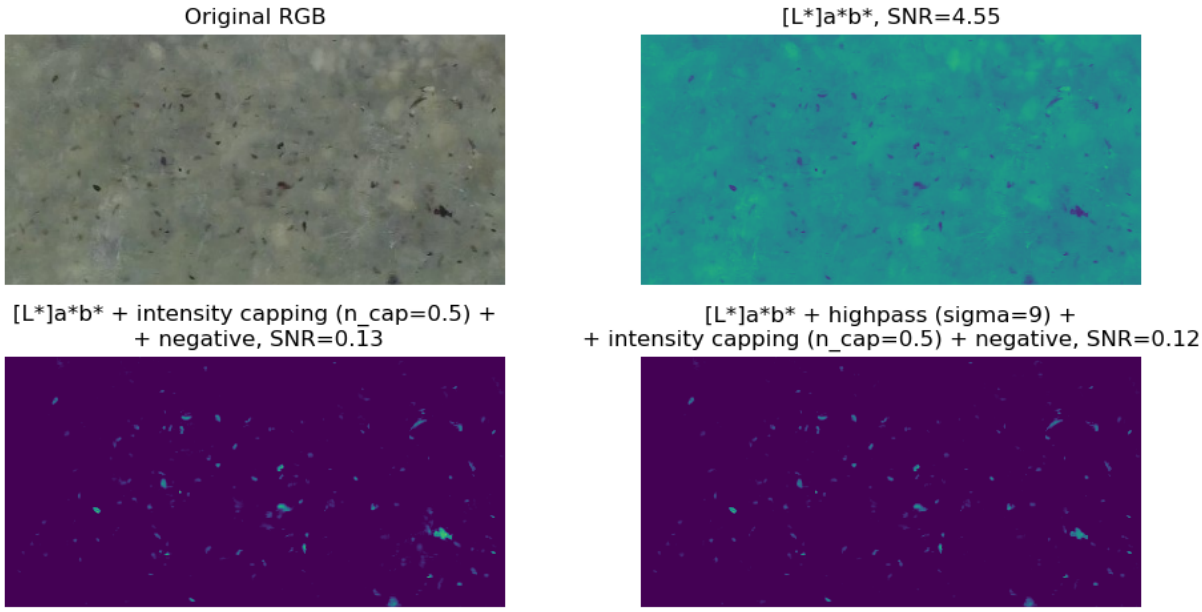
There are very few differences between the image channels in the figure above, although the [L*]a*b*, R[G]B, and HS[V] have a somewhat higher SNR than the remaining options. The primary task here should be the removal of visible riverbed features, which is usually done with image capping.

Keep in mind: When images contain dark tracer particles on lighter background, one should apply

negative filter (~ operator) before calculating SNR metric to obtain proper results where lower SNR are better.

In [16]:
```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(9.8, 4.8))

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

ax[0][1].imshow(lab_l)
ax[0][1].set_title('[L*]a*b*, SNR={:.2f}'
                   .format(snr(~lab_l[y1: y2, x1:x2])))

# Intensity capping, set dark-on-light (DoL) mode
n_cap = 0.5
img5_v1 = intensity_capping(lab_l, n_std=n_cap, mode='DoL')

ax[1][0].imshow(~img5_v1, vmin=0, vmax=255)
ax[1][0].set_title('[L*]a*b* + intensity capping (n_cap={:.1f}) +' \
                   '\n + negative, SNR={:.2f}'
                   .format(n_cap, snr(~img5_v1[y1: y2, x1:x2])))

# Intensity capping
sigma = 9
n_cap = 0.5
img5_v2 = lowpass_highpass(lab_l, sigma=sigma)[1]
img5_v2 = intensity_capping(img5_v2, n_std=n_cap, mode='DoL')

ax[1][1].imshow(~img5_v2, vmin=0, vmax=255)
ax[1][1].set_title('[L*]a*b* + highpass (sigma={}) +' \
                   '\n+ intensity capping (n_cap={:.1f}) + negative, SNR={:.2f}'
                   .format(sigma, n_cap, snr(~img5_v2[y1: y2, x1:x2])))

[a.axis('off') for a in ax.reshape(-1)]
[a.set_xlim([x1, x2]) for a in ax.reshape(-1)]
[a.set_ylim([y2, y1]) 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



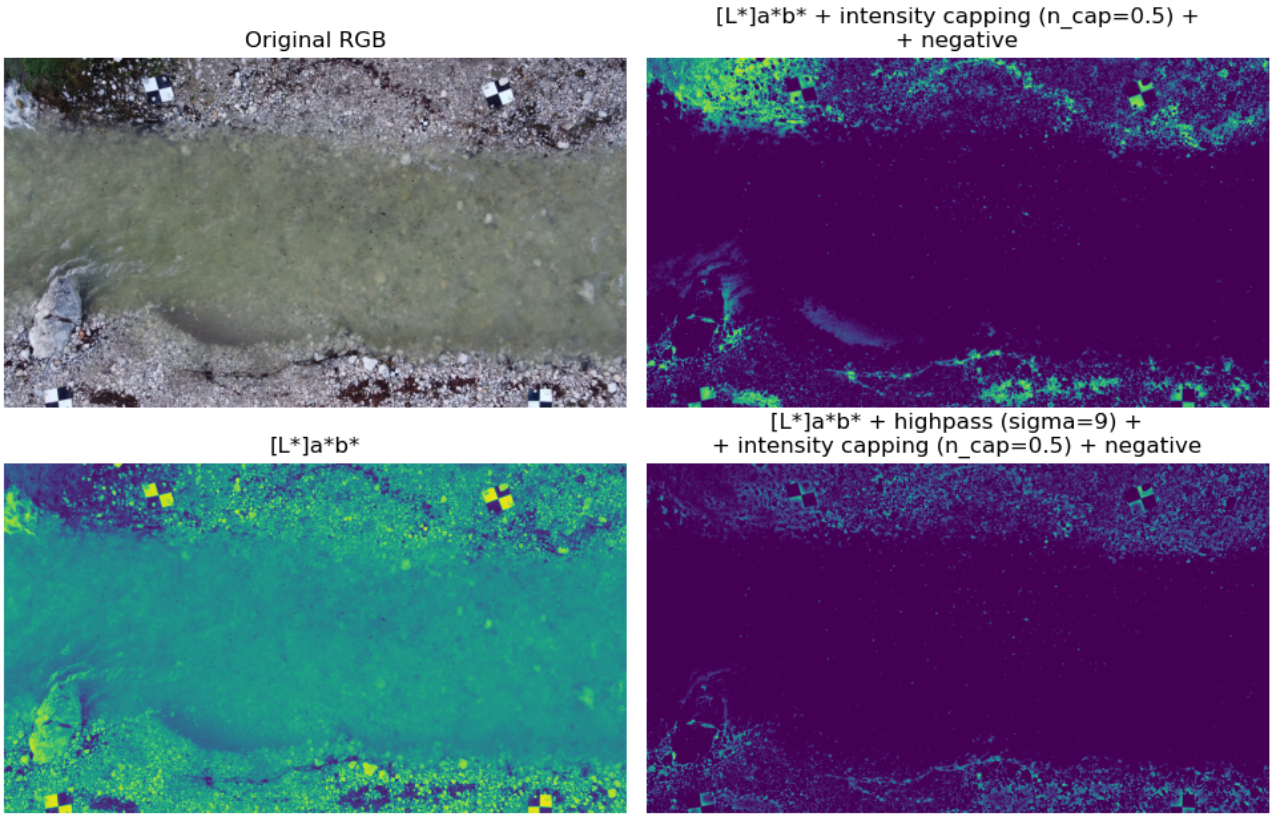
At the first glance, the differences between the two approaches are quite small - highpass filter manages to remove some additional riverbed noise (consider area around [1145, 495]), but the remainder of the subimage is practically identical. However, larger differences are more obvious when the entire image is considered. Highpass filter does not help accentuate the tracer particles, but manages to remove surface wave features and riverbed features (consider, for example, area around [600, 840]). The conjunction of highpass and intensity capping should therefore ALWAYS start with highpass filter followed by intensity capping, as the latter indiscriminately removes a significant amount of information.

> **Tip:** Image enhancement workflow which involves several filter is usually performed in order of the least invasive filter towards the most invasive. For example, highpass filter generally precedes intensity capping because it removes less information than capping. Thresholding algorithms such as intensity capping are often performed last.

In [17]:
```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(9.8, 6.4))

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

ax[1][0].imshow(lab_l)
ax[1][0].set_title('[L*]a*b*')

ax[0][1].imshow(~img5_v1)
ax[0][1].set_title('[L*]a*b* + intensity capping (n_cap={:.1f}) +' \
                   '\n+ negative'.format(n_cap))

ax[1][1].imshow(~img5_v2)
ax[1][1].set_title('[L*]a*b* + highpass (sigma={}) +' \
                   '\n+ intensity capping (n_cap={:.1f}) + negative'
                   .format(sigma, 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



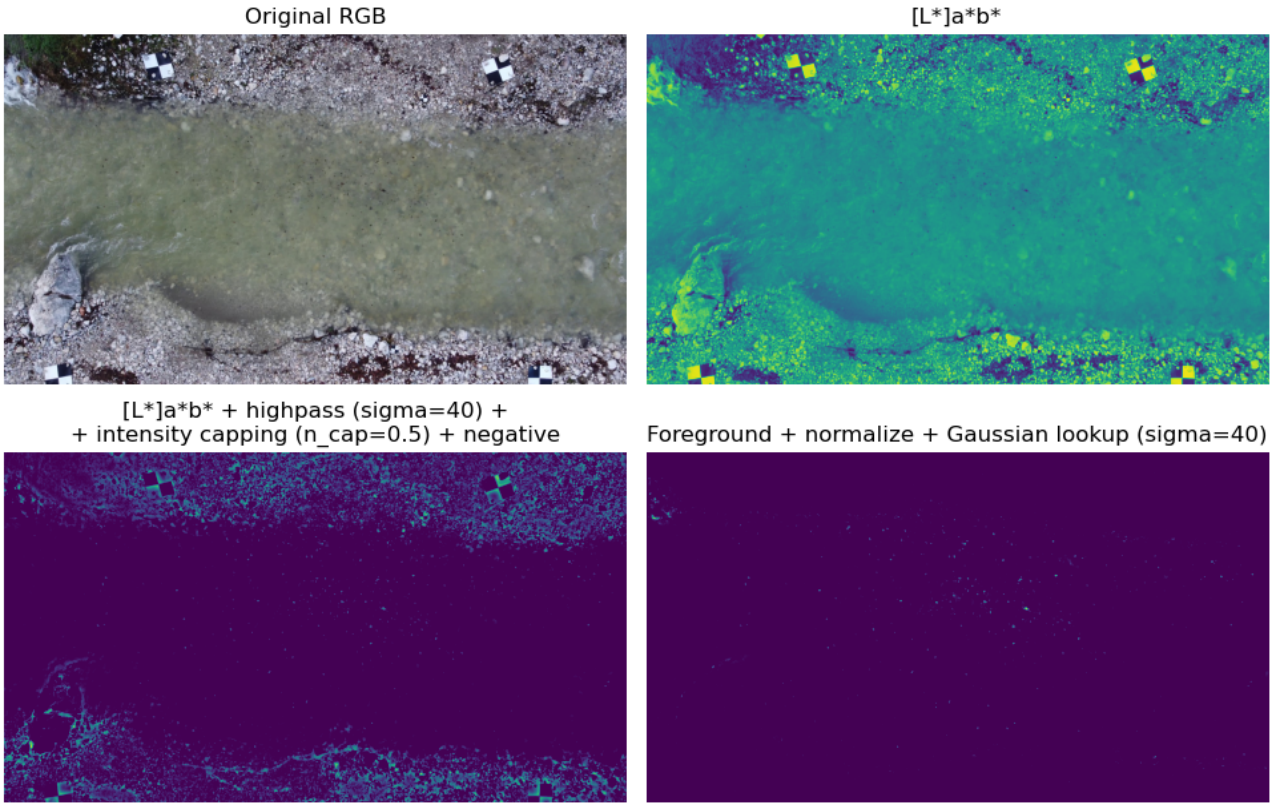
In conclusion of the final example, the combination of highpass filter to remove riverbed and wave features and intensity capping to flatten the background produces a well-defined foreground with tracer particles across the entire water surface.

An alternative approach to this example could be to use **background removal**, followed by **image normalization** and **Gaussian lookup**:

In [18]:
```
# Load background from file and get the [L*]a*b* channel
background_bgr = cv2.imread('./background.jpg')
background_l = cv2.cvtColor(background_bgr, cv2.COLOR_BGR2LAB)[:, :, 0]

# Calculate the foreground
foreground = cv2.subtract(background_l, lab_l)

# Normalize foreground
foreground = normalize_image(foreground)

# Gaussian lookup
sigma = 40
_, _, foreground = gaussian_lookup(foreground, sigma=sigma)

fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(9.8, 6.4))

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

ax[0][1].imshow(lab_l)
ax[0][1].set_title('[L*]a*b*')
```

```
ax[1][0].imshow(~img5_v2)
ax[1][0].set_title('[L*]a*b* + highpass (sigma={}) +' \
                   '\n+ intensity capping (n_cap={:.1f}) + negative'
                   .format(sigma, n_cap))

ax[1][1].imshow(foreground)
ax[1][1].set_title('Foreground + normalize + Gaussian lookup (sigma={})'.format(sigma))

[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



Background removal procedure appears to perform quite well, with the tracer particles being far more accentuated when compared to the highpass + intensity capping approach.

6.7 Conclusions on filter stacking

The presented examples underline the conclusion from notebook on Image filtering that not all of the described procedures are equally important and adequate for achieving optimal enhancement for image velocimetry. The most pervasive problems which require image enhancement are:

1. Low contrast between the tracer particles and the water surface,
2. Surface waves (when such are not used as features for tracking),
3. Light reflections (usually against surface waves), and
4. Visible riverbed.

The presented enhancement procedures have all begun with colorspace model exploration, and their analysis in terms of signal-to-noise (SNR) metric to quantify the local accentuation of tracer particles against the water surface. This approach is highly recommended before the actual application of filtering methods, especially in highly complex ground/water surface conditions. This is most evident in Section 6.5 where specific colorspace manipulations have been applied to obtain images with highly accentuated tracer particles. This underlines another important recommendation:

> Use of colored tracer particles - especially those in base (red, green, blue, yellow) colors - is highly recommended in complex ground/water surface conditions, as they enable various colorspace manipulations, which can significantly facilitate the image enhancement workflow.

Of all the presented filtering methods, pixel value **intensity capping** and **highpass filter** are demonstrated to hold the most practical potential. For cases 1..3 described above, **intensity capping** is usually capable enough to accentuate the tracer particles against the background, especially when applied to an adequate colorspace model. Image normalization should generally follow the intensity capping procedure. The capping method is usually monoparametric, with the capping parameter performing well when in range between 0.0 and 2.0 (higher values are more "aggressive", removing more background).

Highpass filter, which targets and removes low-frequency content, is an adequate method for the removal of larger background features such as riverbed structures and larger patches of light reflections or surface waves. However, intensity capping is still a recommended follow-up step. Highpass filter method is also monoparametric, with the parameter `sigma` defining the size of the targeted low-frequency content. While this parameter's value is usually in the double-digits, it is best explored through trial and error for each different case.

The two methods mentioned above also underline an important filter stacking principle - experience indicates that it is more feasible to apply less aggressive methods first, which target and remove less information, and usee the more aggressive methods in the end. This approach of gradual increase of the filtering intensity seems to retain the most ammount of relevant information.

The conclusions about the remaining filtering methods can be summarized as follows:

1. Adjustments of brightness, contrast, and gamma are usefull only if the images are over- or underexposed,
2. Both global histogram equalization and CLAHE are generaly unfavorable as they almost inevitably lead to overexposed water surface,
3. Denoising algorithms should be applied only when there is visible salt-and-pepper noise in the image,
4. Background removal is useful, especially when followed by other filtering methods such as image normalization, Gaussian lookup, etc.

Continue to next chapter: Image enhancement using SSIMS >>

or

<< Back to MAIN notebook

7 Image enhancement using SSIMS

This final section will demonstrate the use of the free and open-source tool **SSIMS: Preprocessing tool for UAV image velocimetry** for image enhancement. The tool is available through its GitHub repository, and it is recommended to download its latest release (v0.3.2.0 as of writing). A detailed explanation on how to download, install, and setup the tool is available on the GitHub homepage.

7.1 Video unpacking

The main form of the tool (Figure 1 left) contains several workflows for different steps of the UAV image preprocessing. The focus of this section will be on the unpacking the video and image enhancement, which uses the top-left section of the main form (dashed line in Figure 1 left).

Video unpacking can be performed by clicking on the top-left button in the main form labeled **Unpack video into frames**, which opens a new form (Figure 1 right). Here, the user can select a video file, choose camera parameters or calibrate the camera, define image extraction format and quality, cut a section of the video, etc. More detailed explanation for using this form can be found in the Readme section on the tool's GitHub homepage.

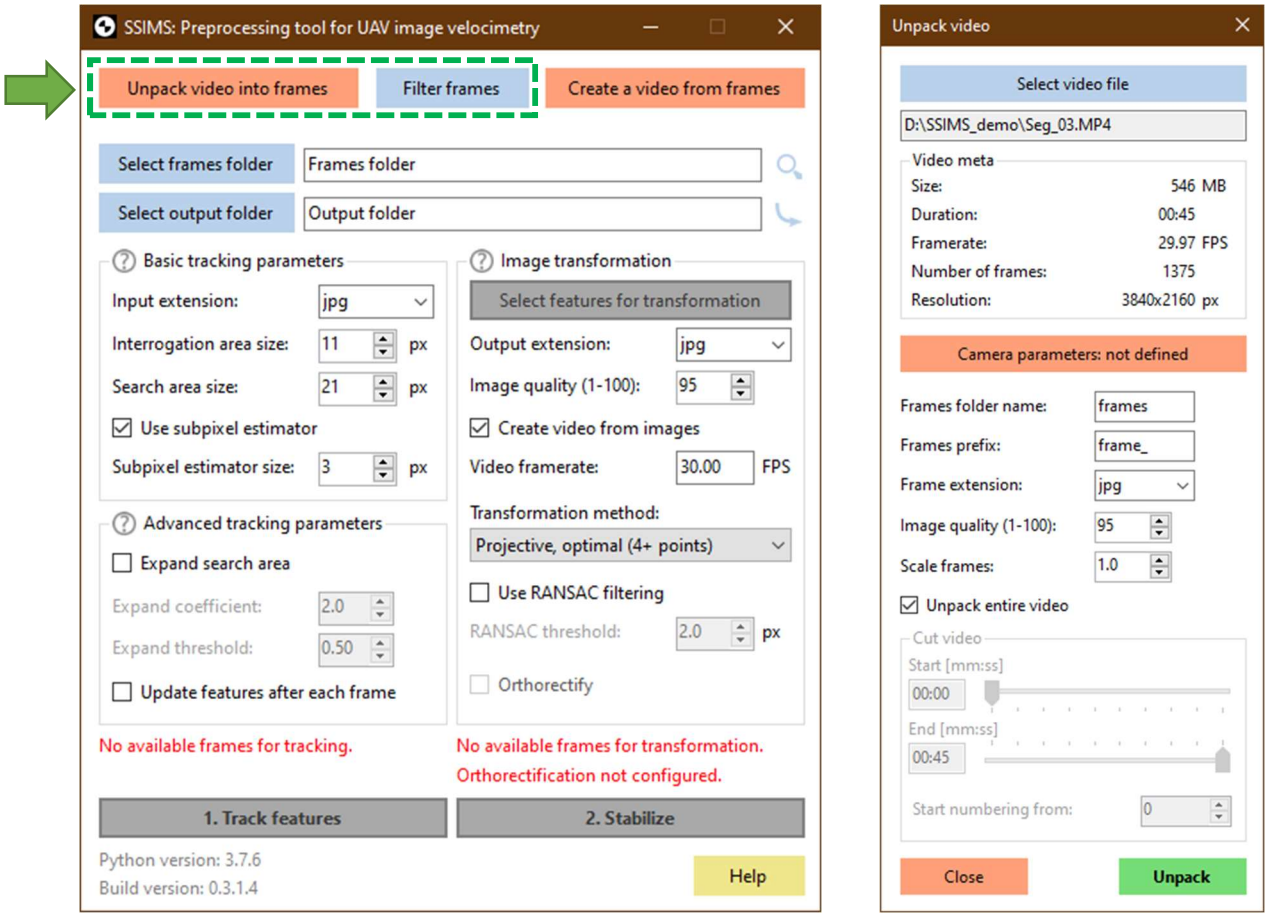


***Figure 1**. Left: SSIMS main form. Right: Video unpacking form.*

7.2 Filter frames form

Once the video has been unpacked into individual frames, we can start the image enhancement by clicking on the **Filter images** button in the top-center of the main form (Figure 1 left). This will open the "Filter frames" form (Figure 2) in which we can define the entire image enhancement procedure.

The "Filter frames" form allows us to select the folder which contains unpacked video frames, and we need to specify their file extension as well. The tool will automatically count and display the number of available frames with the given extension in the selected folder.

Tip: SSIMS will eventually create a file in the frames' folder describing the selected filters and their parameters. If the frames' folder already contains said file, we will be greeted with a prompt asking is we wish to automatically load them to the stack. This can be especially useful when revisiting old projects.

The lower section of the form is divided into two sections:

1. **Left section** contains all the available filters and image enhancement procedures, while the
2. **Right section** is the filtering stack which will eventually contain all the filters which we will select (Figure 2 right).

Selecting filters is done by simply clicking on their corresponding orange buttons on the left side of the form, which will add them to the filtering stack on the right side. Available filters are ordered loosely by their general applicability for image enhancement for velocimetry purposes using tracer particles – filters closer to the top are usually more useful than the remainder.

Keep in mind: Selected filters will become unavailable in the left side as long as they are in the filtering stack (Figure 2 right), which means that the **same filter cannot be added twice** to the filtering stack.

***Figure 2**. Left: initial view of the image enhancement form. Right: After selecting filters.*

Filters in the stack will be executed against every frame **in the order which they hold in the stack.** We can reorder the selected filters in the stack (Figure 2 right) by dragging and dropping them at desired positions.

"Filter frames" form also contains options for performing various colorspace conversions (grayscale/RGB/HSV/L*a*b*). **By default, images are loaded in the RGB format**. SSIMS **will keep track** of the colorspace models throughout the workflow – for example, if we use (1) Convert to HSV, then (2) Convert to L*a*b*, and finally (3) Convert to RGB, the result will be identical to the original image. This is very important when using the "Single image channel" filter, which will single-out a specific image channel from a three-channel image – **channel order will always correspond to the last resulting colorspace** in the stack, or to the RGB model if no conversion was performed.

7.3 Filter parameters

Once we add the filter to the stack, we can **edit its parameters** by clicking on the corresponding green button in the stack. This action will open a window which will contain the description of the filter, and trackbar(s) + numeric box(es) which we can use to define the filter parameters (Figure 3 left). If the filter does not have any settable parameters, an appropriate message will be shown, and trackbars/numeric boxes will be omitted (Figure 3 right). Once the parameter values have been set, we can accept them by clicking on the **Apply** button.

The effects of individual filters can be examined by clicking on the **Preview this filter** button in the "Filter parameters" form. This will initiate a new window demonstrating the effects of just the selected filter. We can remove the selected filter from the stack by clicking on the **Remove** button in the Filter parameters form.

Tip: If the resulting image is grayscale, i.e., all three image channels are identical, the preview window will apply a colormap to provide more viewing contrast (Figure 4). However, once saved to disk, the images will be grayscale.

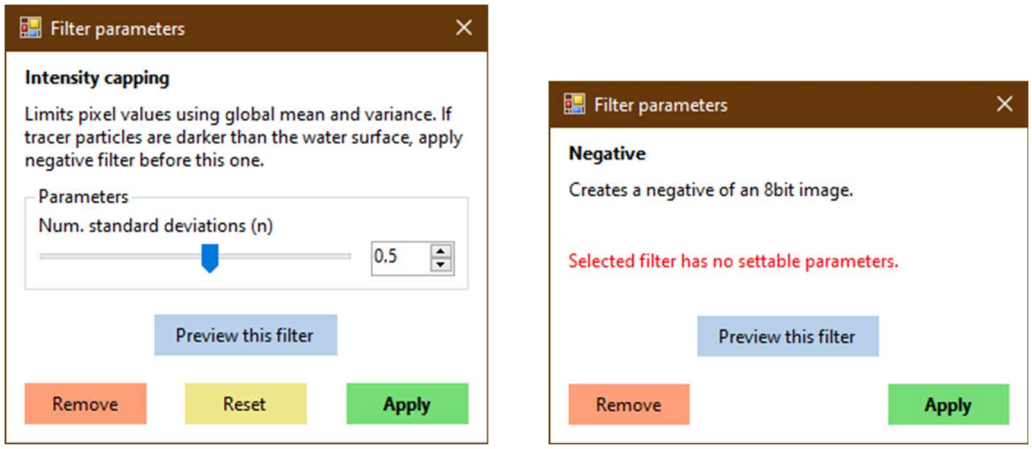


Figure 3. Examples of Filter parameters form.

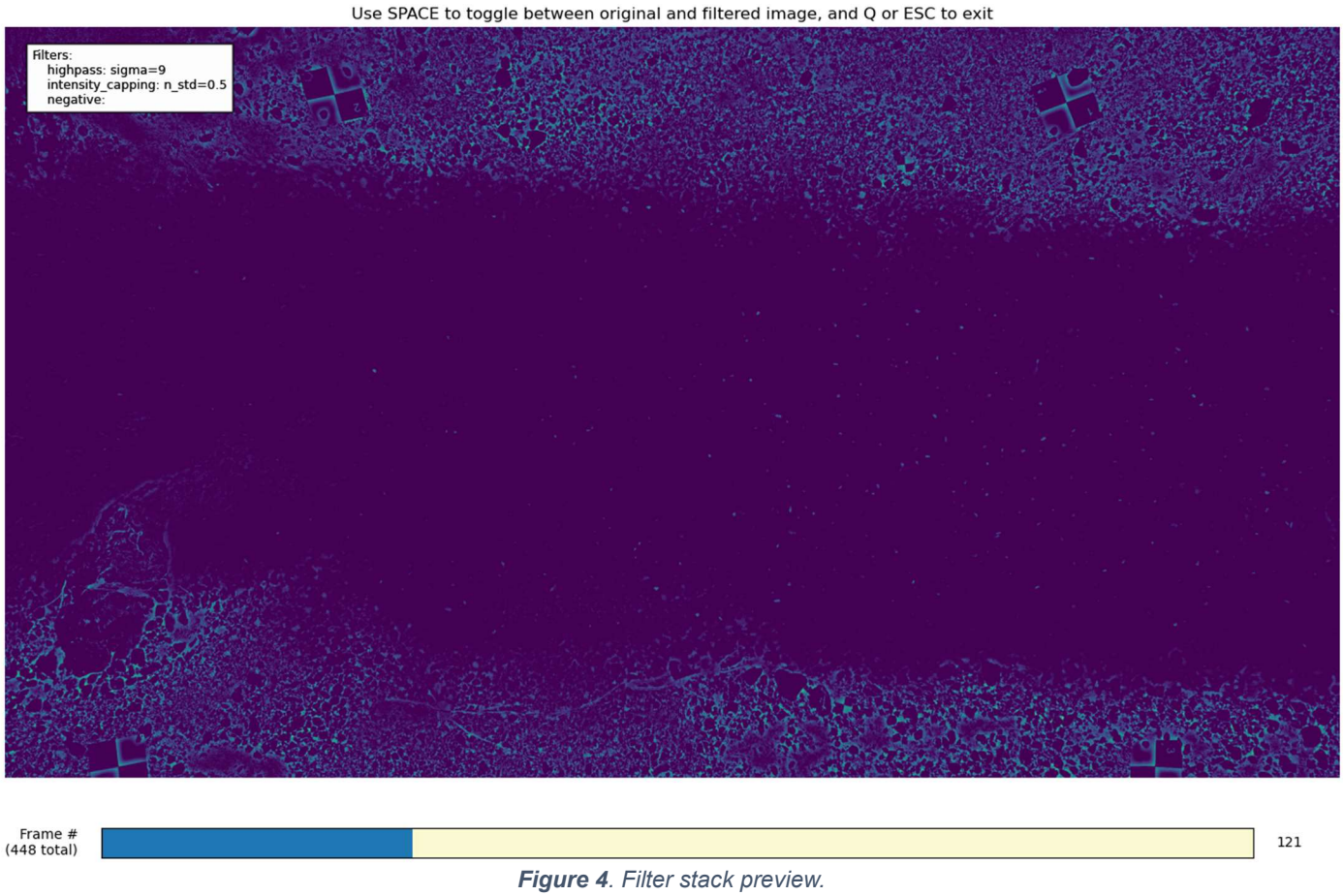


Figure 4. Filter stack preview.

7.4 Final results

When desired filters have been selected and their parameters are set, we can **preview their combined effect** by clicking the **Preview filter stack** button in the bottom center of the "Filter frames" form (Figure 2). This will be shown in a new window, as shown in Figure 4. List of applied filters and their parameters will be shown in the top left corner of the window. To fully examine the final results, a slider shown at the bottom of the window can be used to switch between the individual frames from the selected folder. Cross-examination between the original image and the filtered one can be performed by toggling the **SPACE** key on the keyboard.

Once we are happy with the results of our filter stack, we can start filtering all images from the selected folder by clicking on the **Apply filters** button in the bottom right of the "Filter frames" form. This will create a new folder next to the frames' folder with added suffix `_filtered`, where the filtered frames will be saved.

Appendix: Creating and modifying filters

Even though SSIMS comes with a number of available filters, user can implement and add own filters with relative ease. The procedure consists of two steps and requires basic knowledge of the Python programming language and familiarity with the OpenCV library.

Tip: The author of the tool welcomes **comments and suggestions** about new functionalities, **ideas** on what to include in the future releases, as well as constructive **criticisms** about the SSIMS tool, and encourages sending them directly at rljubicic@grf.bg.ac.rs or ljubicicrobert@gmail.com, or through the official GitHub repository.

Step 1: Creating filter definition

We need to edit the `filters.xml` file, located in the `%SSIMS_path%/scripts` folder. This file contains the definitions for individual filters, with the structure as shown in Figure 5. Definition requires:

`<Filter>…</Filter>`	= Enclosing tag of the single filter definition,
`<Name>…</Name>`	= Short name of the filter, to display in "Filter frames" form,
`<Func>…</Func>`	= Name of the Python function,
`<Description>…</Description>`	= Short description of what the filter does,
`<Parameters>…</Parameters>`	= Collection of filter parameters.

```
<Filter>
    <Name>CLAHE</Name>
    <Func>clahe</Func>
    <Description>Adaptive version of the histogram equalization with histogram clipping.</Description>

    <Parameters>
        <Parameter>
            <PName>Clip limit</PName>
            <PType>float</PType>
            <PUpper>10.0</PUpper>
            <PStart>2.0</PStart>
            <PLower>0.1</PLower>
            <PStep>0.1</PStep>
        </Parameter>
        <Parameter>
            <PName>Tile size</PName>
            <PType>int</PType>
            <PUpper>64</PUpper>
            <PStart>8</PStart>
            <PLower>4</PLower>
            <PStep>4</PStep>
        </Parameter>
    </Parameters>
</Filter>
```

***Figure 5**. Example of a filter definition from `filters.xml` file.*

`<Func>…</Func>` **must be equal** to the name of the Python function which handles the filtering. This will be explained further in Step 2. `<Parameters>…</Parameters>` collection contains definitions of all the individual parameters of the filter. Each parameter definition must contain the following:

`<Parameter>…</Parameter>`	= Enclosing tag of the single parameter definition,
`<PName>…</PName>`	= Name of the parameter,
`<PType>…</PType>`	= Can be either `int` or `float`,
`<PUpper>…</PUpper>`	= Highest allowed value of the parameter,
`<PStart>…</PStart>`	= Initial (default) parameter value.
`<PLower>…</PLower>`	= Lowest allowed value of the parameter.
`<PStep>…</PStep>`	= Increment/decrement step of the parameter value.

Multiparameter filters should contain consecutive `<Parameter>…</Parameter>` definitions. Filters with no settable parameters should just contain empty `<Parameters></Parameters>` collection.

SSIMS tool will now automatically recognize the filter definition and add it to the list in the “Filter frames” form.

Step 2: Creating a Python filter function

Second and final step of the procedure is to add a function which handles the actual filtering to the `filter_frames.py` file, also located in the `%SSIMS_path%/scripts` folder. **The name of the function must correspond** to the name defined in the corresponding `<Func>…</Func>` in the `filters.xml` file (Figure 6).

```
def clahe(img, clip=2.0, tile=8):
    print('[FILTER] CLAHE: clip={:.1f}, tile={:.0f}'.format(clip, tile))

    clahe = cv2.createCLAHE(clipLimit=clip, tileGridSize=(int(tile), int(tile)))
    img_gray = convert_img(img, colorspace, 'grayscale')
    img_clahe = clahe.apply(img_gray)

    return convert_img(img_clahe, 'grayscale', colorspace)
```

***Figure 6**. Example of a Python function which handles a filter.*

Function header should be defined as:

```
def function_name(img, *params):
```

where `function_name` corresponds to the `<Func>…</Func>` from the `filters.xml`, `img` is the initial (original) image and must be the first parameter, with the remaining parameters are to be listed after. The names of the remaining parameters do not have to correspond to their names from the `filters.xml`, but their total number and order should.

Keep in mind: Parameter values are **passed to the function as floats** by default. If you require their integer values, this conversion must be explicit – see end of line 4 in Figure 6.

The return value of the function is usually defined as:

```
return convert_img(resulting_img, colorspace_filter, colorspace)
```

where **`convert_img`** is the in-built function which handles all colorspace model conversions, `resulting_img` is the filtered image, `colorspace_filter` is the colorspace used when handling image inside the filter function. Third parameter `colorspace` should not be changed, as the `colorspace` global variable keeps track of the current working colorspace model.

For example in Figure 6, image was converted to grayscale model (line 5) from the working colorspace defined by variable `colorspace`, because OpenCV’s CLAHE function only works with single-channel images. However, the image was later returned to the working colorspace model (line 8) using an inverse transformation.

Keep in mind: Each filter receives a three-channel image and **must return a three-channel image**. This allows various filters to be performed sequentially, without the need to keep track of the previous image manipulations. If the output is a single-channel (grayscale) image, the image should be copied to channels two and three:

```
return cv2.merge([img_single, img_single, img_single])
```

The Python script will now be able to properly connect the filter definition to the handling function.