<< 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

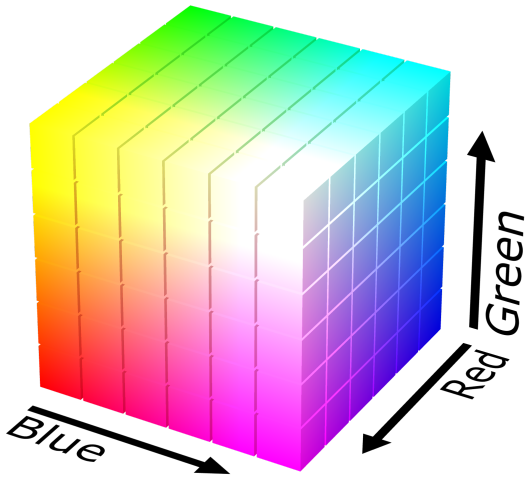
In [1]:
```
# Necessary libraries
import itertools
import numpy as np
import matplotlib.pyplot as plt
import cv2

from os import mkdir
from os.path import exists, dirname
from axes_tiein import on_lims_change

# 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:

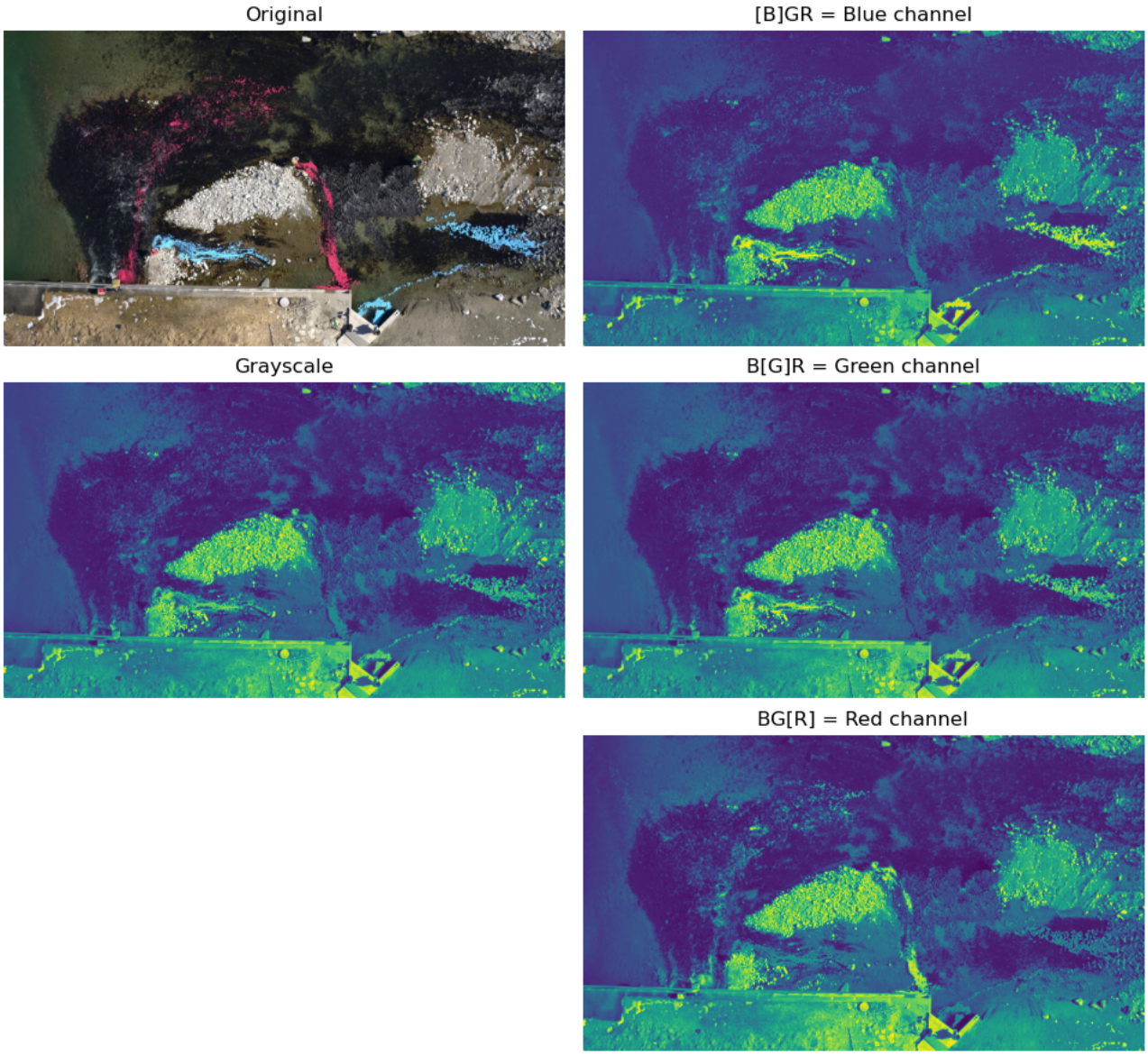


RGB/BGR colorspace model representation, source: Wikimedia

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()
```



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`.

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

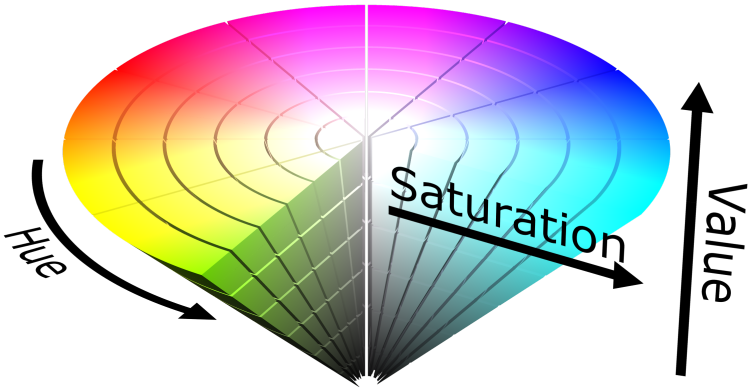
Unlike RGB/BGR model where individual colors intensities are represented in separate image channels, HSV takes a somewhat different approach of separating image data into three channels:

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.



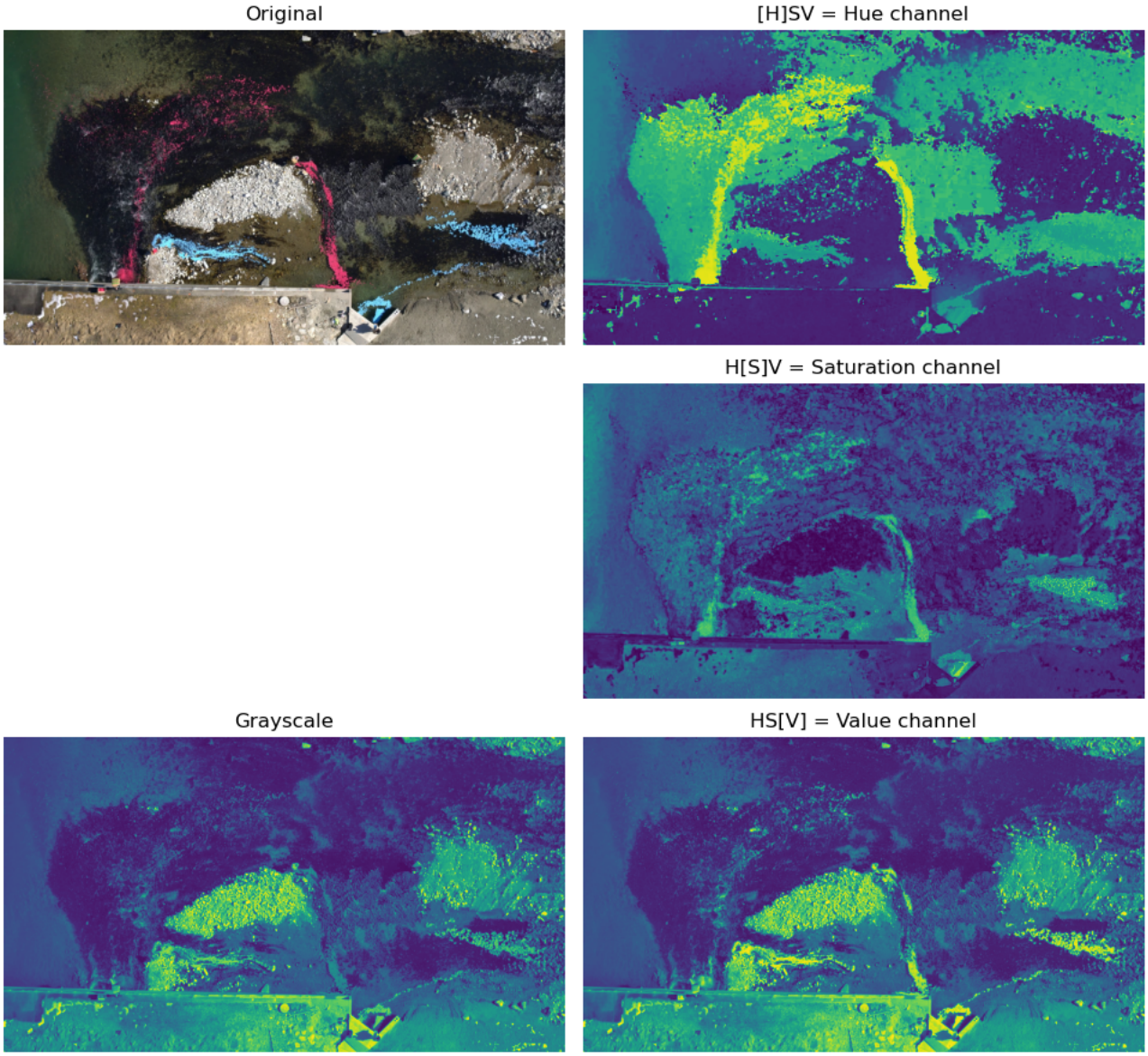
HSV colorspace model representation, source: Wikimedia

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



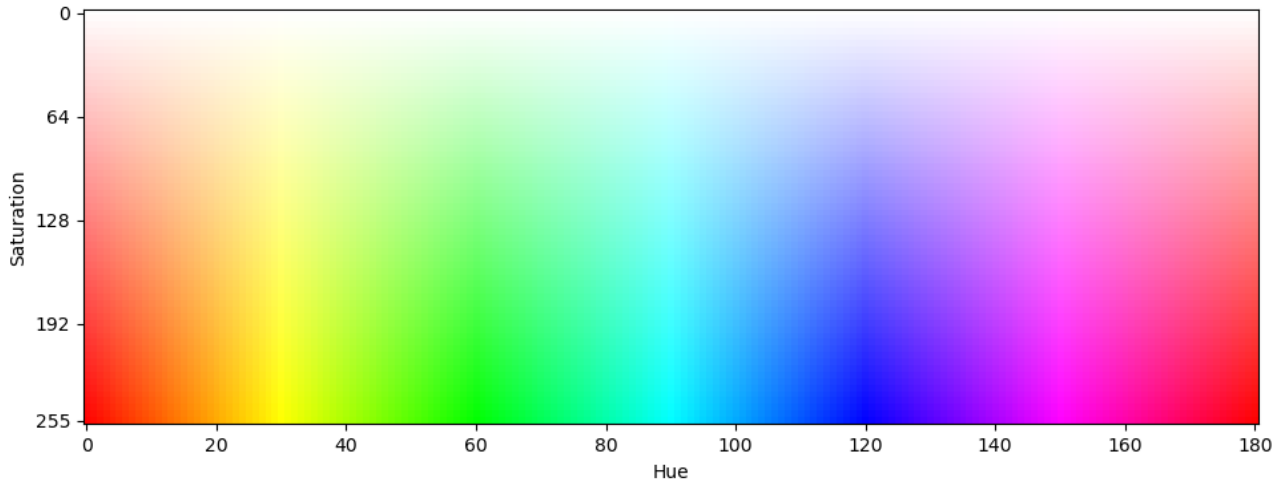
When placed next to each other, grayscale model and HS[V] channel appear alike - this is not by accident as grayscale is primarily a representation of human perception of color brightness. However, in the case of **Image 4**, tracers in the HS[V] channel appear to be significantly more pronounced than 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

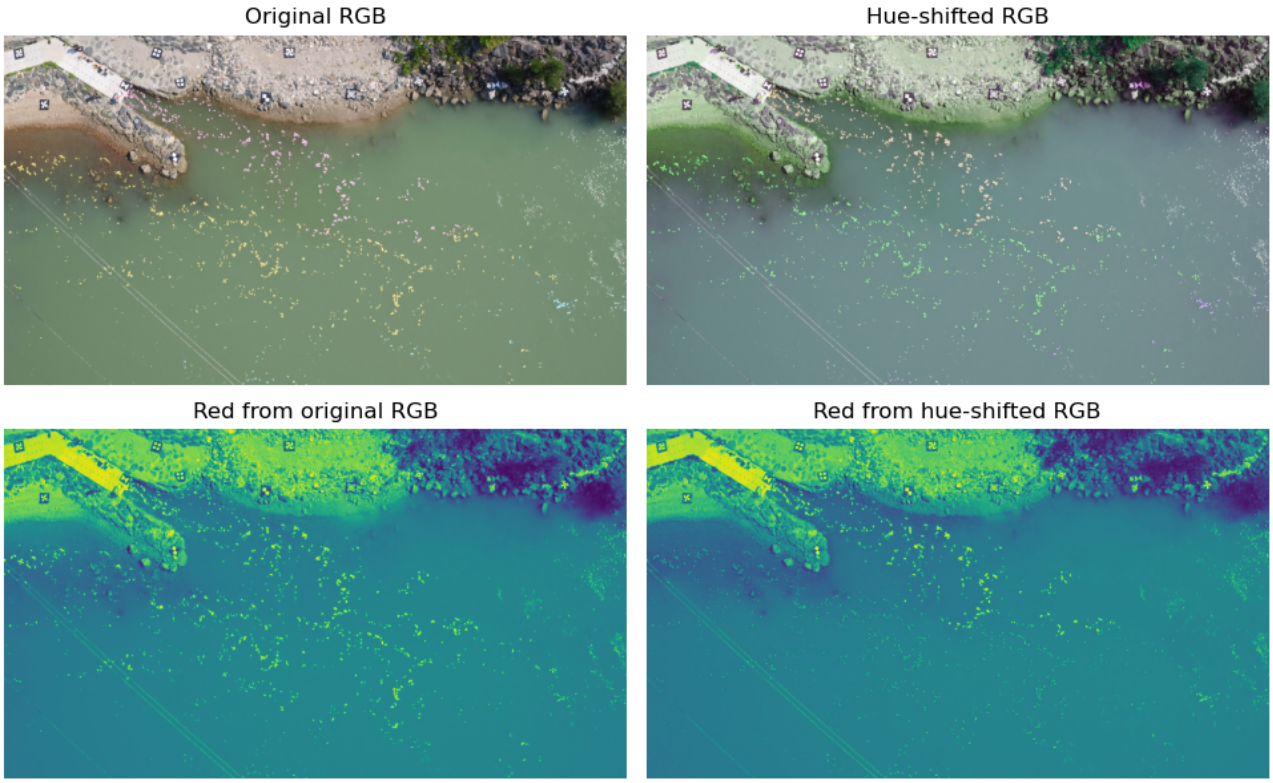


HSV model allows us to do something that RGB/BGR does not - to manipulate color hue in order to better grab certain information from RGB channels. For instance, pink-colored tracer particles from **Image 2** have a hue value of about 145 (check by hovering with mouse over these areas in figures above). 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).

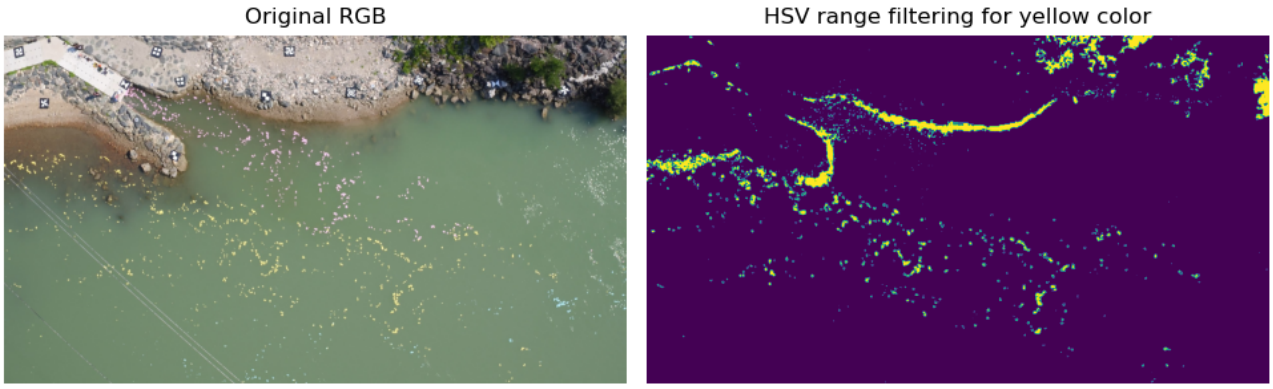
There are other methods of manipulating the HSV colorspace (for example by creating lookup tables) but such strategies can be considerably more complex and worthwhile only in specific cases, thus will not be described in this report.

> **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.

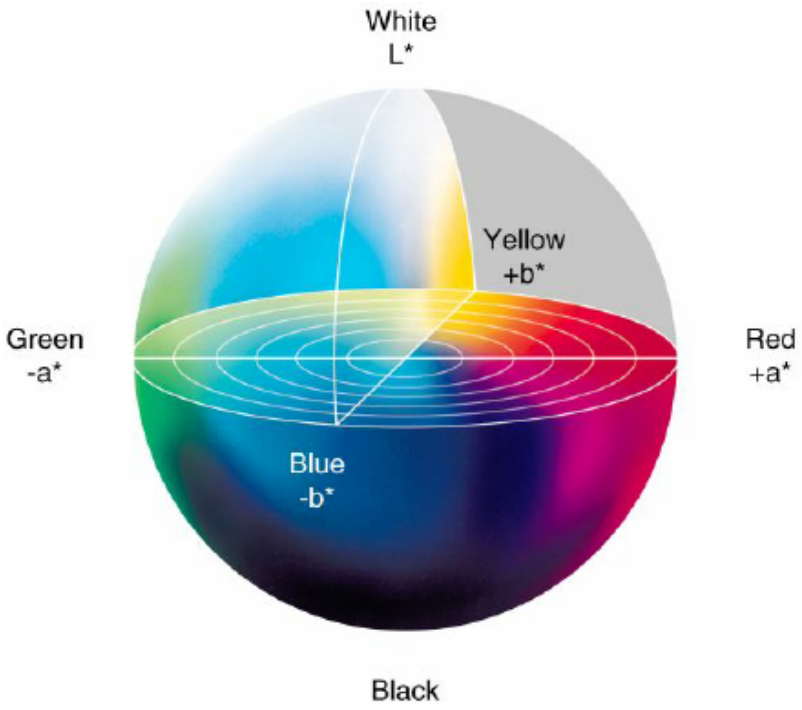
4.4 L*a*b*

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.



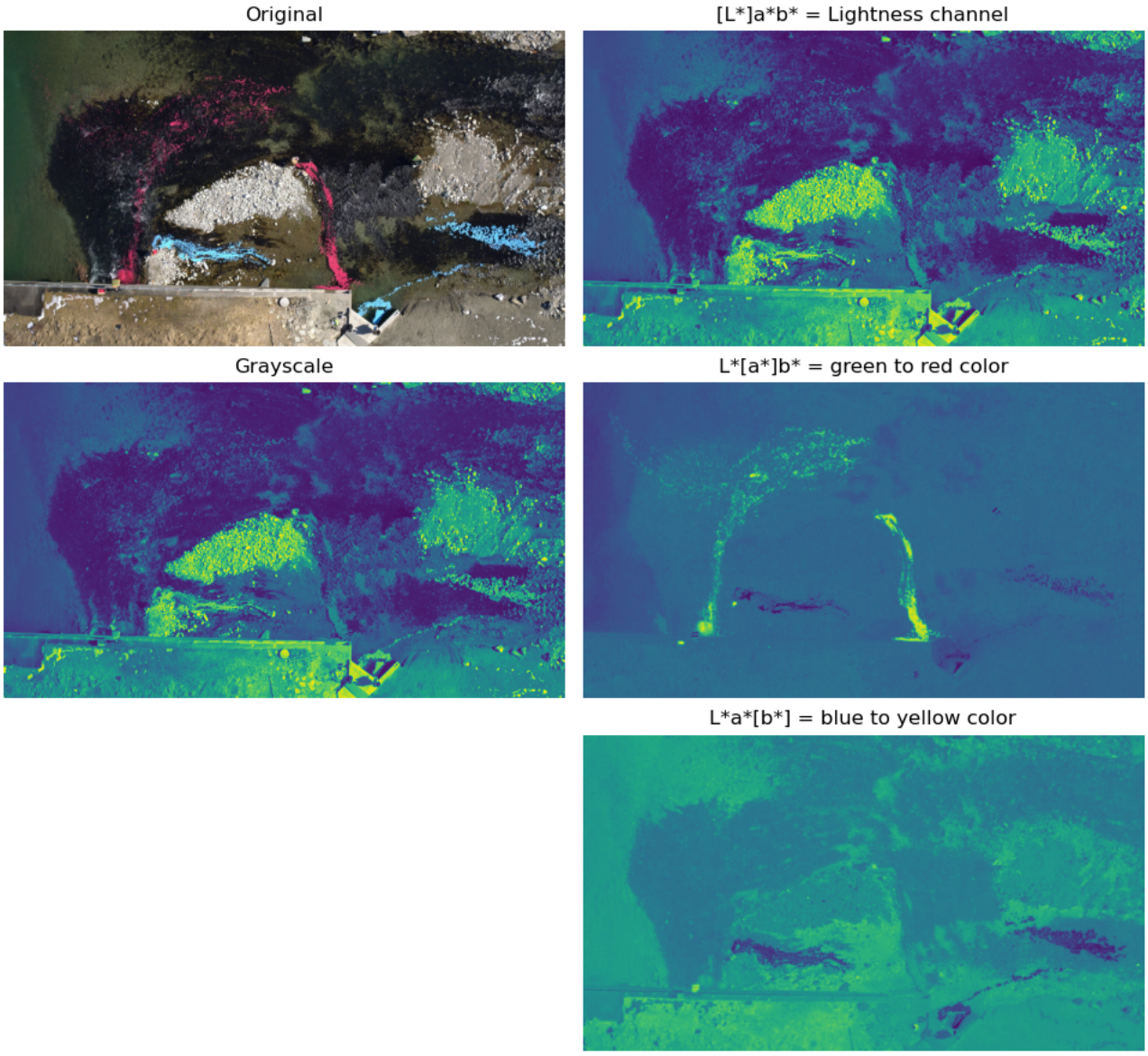
*L*a*b* colorspace model representation, (Agudo et al., 2014)*

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