Exam

INF250
Fall 2020
Candidate 103

Table of contents

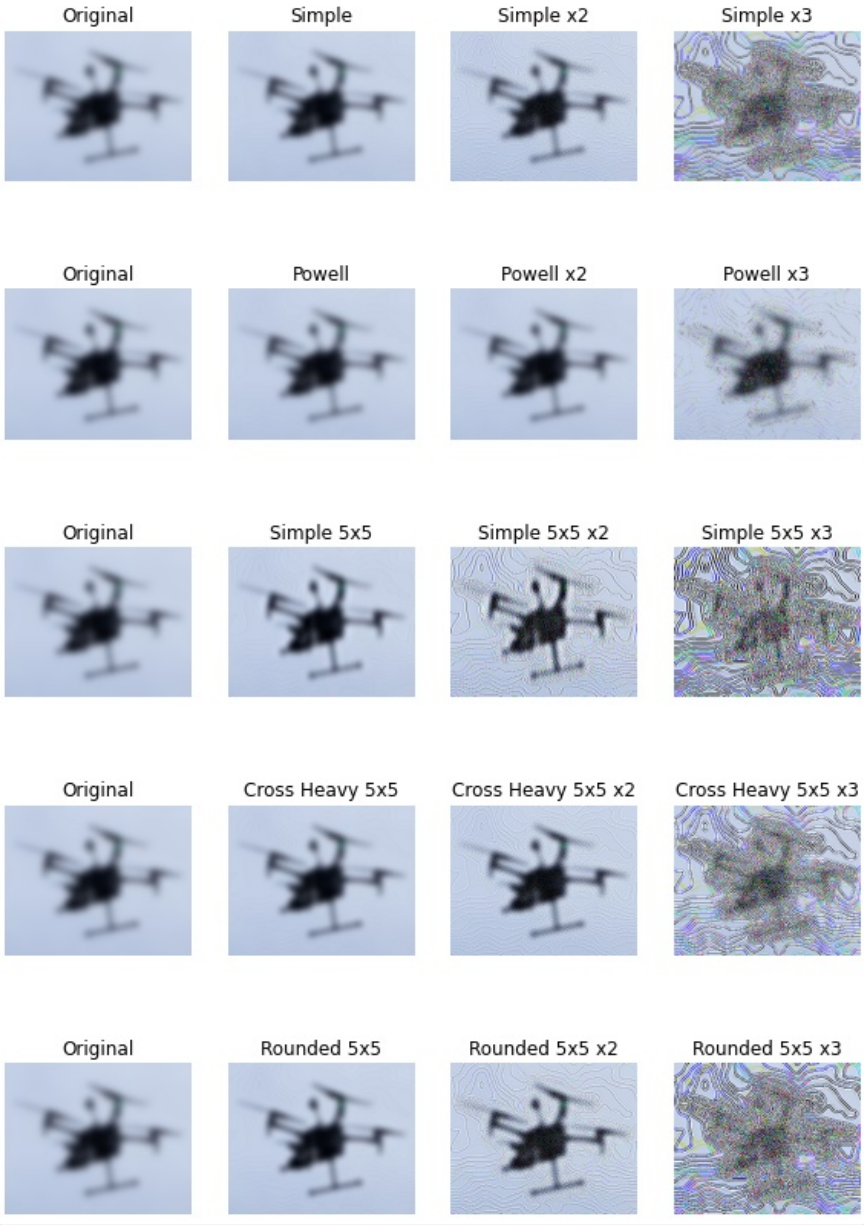
Exercise 1

Sharpening is any method of increasing contrast in an image to enhance and bring forth features. It does not increase resolution (but there are deep neural nets that can infer new pixels to an image, if that's needed).

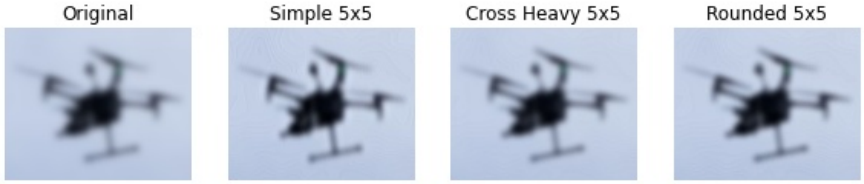
The given image is of high resolution, and filtering won't show any noticable difference, unless we let the image fill the entire screen. So for the report we'll crop it and only focus on the drone to make the effects stand out better to the human eye.

Then we'll try out some filtering with a variety of kernels and passthroughs, before picking out the best performers with regards to human legibility.

Exploring sharpening filters



Best performers for human legibility



Code

Code with more detailed comments below:

```
# %%

"""
INF250 2020 Fall
Exam Exercise 1
"""

import numpy as np
import cv2
from matplotlib import pyplot as plt
import matplotlib.image as mpimg
import math
from typing import Any, Callable, List, Optional, Sequence
from typing_extensions import TypedDict


def crop(
    image: np.ndarray,
    x=[0.0, 1.0],
    y=[0.0, 1.0],
) -> np.ndarray:
    """
    Crop image with percentages
    """

    def get_dimensions(image):
        return len(image[0]), len(image)

    cropped = image[
        int(y[0] * len(image)) : int(y[1] * len(image)),
    ]
    cropped = [
        row[int(x[0] * len(image[0])) : int(x[1] * len(image[0]))]
        for row in cropped
    ]

    return np.array(cropped)


def sharpen_simple(image: np.ndarray) -> np.ndarray:
    """
    Simple sharpening with a typical kernel mentioned in numerous blog posts
    and StackOverflow questions
    """
    return cv2.filter2D(

        image,
        -1,
        np.array(
            [
```

```
                [-1, -1, -1],
                [-1, 9, -1],
                [-1, -1, -1],
            ]
        ),
    )


def sharpen_powell(image: np.ndarray) -> np.ndarray:
    """
    Sharpening with kernel given by Victor Powell in Image kernels visually
    explained:
    https://setosa.io/ev/image-kernels/
    """
    return cv2.filter2D(
        image,
        -1,
        np.array(
            [
                [0, -1, 0],
                [-1, 5, -1],
                [0, -1, 0],
            ],
        ),
    )


def sharpen_simple_3x5(image: np.ndarray) -> np.ndarray:
    """
    Simple sharpening with a typical kernel mentioned in numerous blog posts
    and StackOverflow questions

    Scaled to 3x5
    """
    return cv2.filter2D(
        image,
        -1,
        np.array(
            [
                [-1, -1, -1, -1, -1],
                [-1, -1, 15, -1, -1],
                [-1, -1, -1, -1, -1],
            ]
        ),
    )


def sharpen_cross_heavy_5x5(image: np.ndarray) -> np.ndarray:
    """
    Exploratory sharpening with a 5x5 kernel (horizontal and vertical).
    """
    return cv2.filter2D(
        image,
        -1,
        np.array(
```

```
            [
                [0, 0, -1, 0, 0],
                [0, 0, -1, 0, 0],
                [-1, -1, 9, -1, -1],
                [0, 0, -1, 0, 0],
                [0, 0, -1, 0, 0],
            ],
        ),
    )


def sharpen_rounded_5x5(image: np.ndarray) -> np.ndarray:
    """
    Exploratory sharpening with a 5x5 kernel (radius around center)
    """
    return cv2.filter2D(
        image,
        -1,
        np.array(
            [
                [0, 0, -1, 0, 0],
                [0, -1, -1, -1, 0],
                [-1, -1, 13, -1, -1],
                [0, -1, -1, -1, 0],
                [0, 0, -1, 0, 0],
            ],
        ),
    )


def process(
    image,
    filters: Sequence[Callable[[np.ndarray, Any], np.ndarray]],
    passthroughs=1,
) -> np.ndarray:
    filtered = image

    for _ in range(passthroughs):
        for f in filters:
            filtered = f(filtered)

    return filtered


def show(
    images: List[
        TypedDict(
            "ImageDict",
            {
                "label": str,
                "image": np.ndarray,
            },
        )
    ],
    title=None,
```

```
    cols=2,
    scale=2.5,
    savepath="",
) -> None:
    """
    Shows dynamic rows of images given a list of image dictionaries (see typ
    and an optional amount of columns

    Will save figure if path is given
    """
    rows = int(math.ceil(len(images) / cols))

    fig, ax = plt.subplots(
        nrows=rows,
        ncols=cols,
        figsize=(
            scale * cols,
            scale * rows + scale if title else scale * rows,
        ),
    )

    if title:
        fig.suptitle(
            f"\n{title}",  # Newline to change absolute position
            fontsize=12 * scale - 4,
        )

    if rows > 1:
        for row in ax:
            for axis in row:
                axis.axis("off")
    else:
        for axis in ax:
            axis.axis("off")

    for i, image in enumerate(images):
        if rows > 1:
            axis = ax[int(i / cols)][i % cols]
        else:
            axis = ax[i % cols]

        axis.set_title(image["label"])
        axis.imshow(
            image["image"],
            cmap="gray"
            if type(image["image"][0][0]) in [np.uint8, float]
            else None,
        )

    if savepath:
        plt.savefig(savepath, bbox_inches="tight")


if __name__ == "__main__":
    # Import original image
```

```
original = mpimg.imread("./resources/drone_blurr.tif")

# Crop image to make differences more legible
cropped = crop(original, [0.43, 0.5125], [0.305, 0.355])

# List of image dictionaries to organize and easily loop through methods
images = [
    {
        "label": "Original",
        "image": cropped,
    },
    {
        "label": "Simple",
        "image": process(cropped, [sharpen_simple]),
    },
    {
        "label": "Simple x2",
        "image": process(cropped, [sharpen_simple], 2),
    },
    {
        "label": "Simple x3",
        "image": process(cropped, [sharpen_simple], 3),
    },
    {
        "label": "Original",
        "image": cropped,
    },
    {
        "label": "Powell",
        "image": process(cropped, [sharpen_powell]),
    },
    {
        "label": "Powell x2",
        "image": process(cropped, [sharpen_powell], 2),
    },
    {
        "label": "Powell x3",
        "image": process(cropped, [sharpen_powell], 3),
    },
    {
        "label": "Original",
        "image": cropped,
    },
    {
        "label": "Simple 5x5",
        "image": process(cropped, [sharpen_simple_3x5]),
    },
    {
        "label": "Simple 5x5 x2",
        "image": process(cropped, [sharpen_simple_3x5], 2),
    },
    {
        "label": "Simple 5x5 x3",
        "image": process(cropped, [sharpen_simple_3x5], 3),
    },
```

```
        {
            "label": "Original",
            "image": cropped,
        },
        {
            "label": "Cross Heavy 5x5",
            "image": process(cropped, [sharpen_cross_heavy_5x5]),
        },
        {
            "label": "Cross Heavy 5x5 x2",
            "image": process(cropped, [sharpen_cross_heavy_5x5], 2),
        },
        {
            "label": "Cross Heavy 5x5 x3",
            "image": process(cropped, [sharpen_cross_heavy_5x5], 3),
        },
        {
            "label": "Original",
            "image": cropped,
        },
        {
            "label": "Rounded 5x5",
            "image": process(cropped, [sharpen_rounded_5x5]),
        },
        {
            "label": "Rounded 5x5 x2",
            "image": process(cropped, [sharpen_rounded_5x5], 2),
        },
        {
            "label": "Rounded 5x5 x3",
            "image": process(cropped, [sharpen_rounded_5x5], 3),
        },
    ]

    show(
        images,
        "Exploring sharpening filters",
        cols=4,
        savepath="./images/exercise-1-sharpening.jpg",
    )

    # List of the best performing sharpening
    best_performers = [
        {
            "label": "Original",
            "image": cropped,
        },
        {
            "label": "Simple 5x5",
            "image": process(cropped, [sharpen_simple_3x5]),
        },
        {
            "label": "Cross Heavy 5x5",
            "image": process(cropped, [sharpen_cross_heavy_5x5]),
        },
```

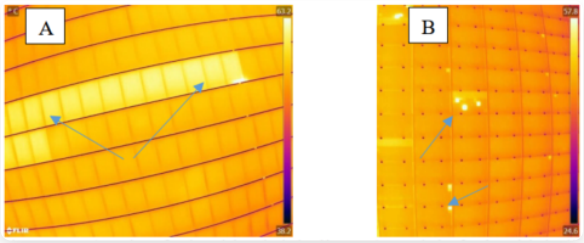
```
        {
            "label": "Rounded 5x5",
            "image": process(cropped, [sharpen_rounded_5x5]),
        },
    ]

    show(
        best_performers,
        "Best performers for human legibility",
        cols=4,
        savepath="./images/exercise-1-best.jpg",
    )
```

Resources

- Image kernels visually explained by Victor Powell

Exercise 2

There are two types of faults that needs to be distinguished, referred to as string (A) and module (B).



Features of the two types of faults:

- A
  - brighter than surrounding area
  - large area
  - squares
- B
  - brighter than surrounding area
  - tiny area
  - circular dots

As we have three color channels, we'll work with three different approaches, and see which one works the best:

1. Find color range of faults using all three channels

2. Find threshold range of faults using grayscale
3. Mix and combine layers based on the color channels features

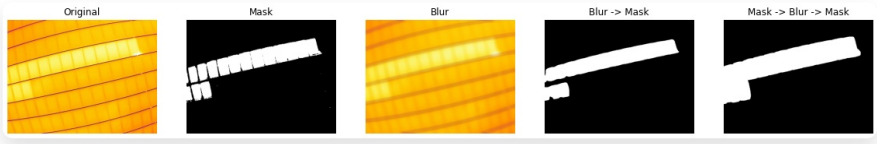
Following how we choose our starting point, the methods will be quite similar:

As both types of faults are brighter than the surrounding area, we can make them more pronounced by sharpening the image followed by some blurring and masking (thresholding) combinations to make them stand more out.

Sharpening wil introduce some noise, so denoising would be a good idea following the sharpening.

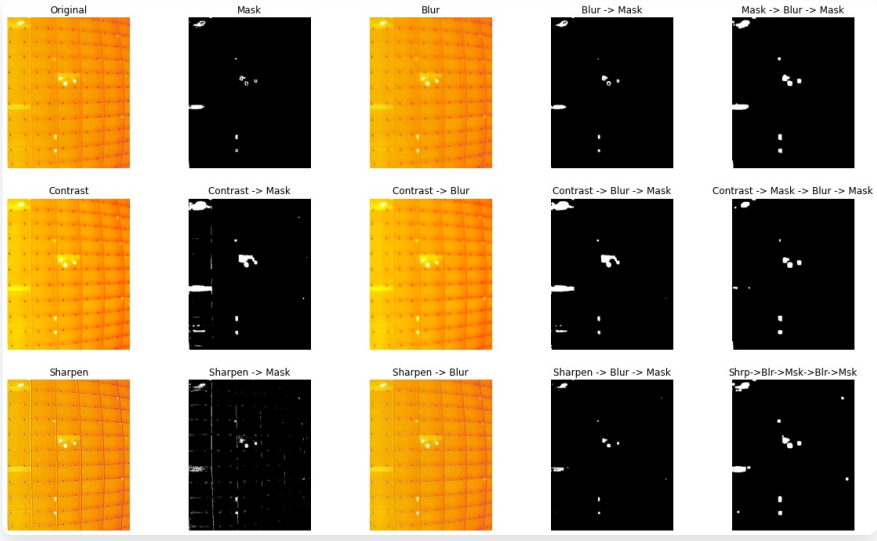
Using contours we can draw squares around the errors, where we can distinguish the two types based on area size. We could differentiate them by shape (square and circle/dot), but it is not safe to assume that dot detection would hold for most module errors. Especially if they overlap or bleed into eachother. Here, area size seems like the safer bet.

First let's see how well we can do with all color channels and using a color range to do fault detection:



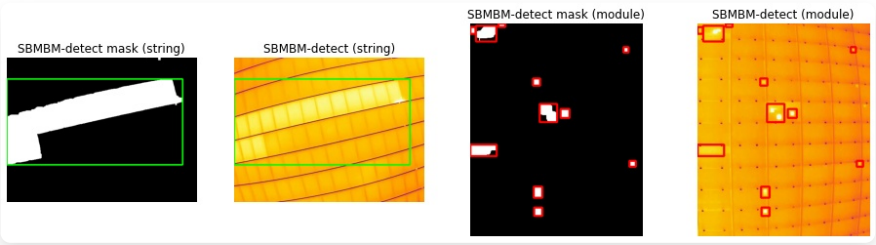
Pretty straight forward for the string fault.

Now, let's see if we can spot the modules:



A bit more experimentation was needed here, but it is likely the same method will work (with some tweaks) for strings.

We have now spotted each fault individually, but the approach needs to distinguish them, so let's combine the two approaches and do some detection:



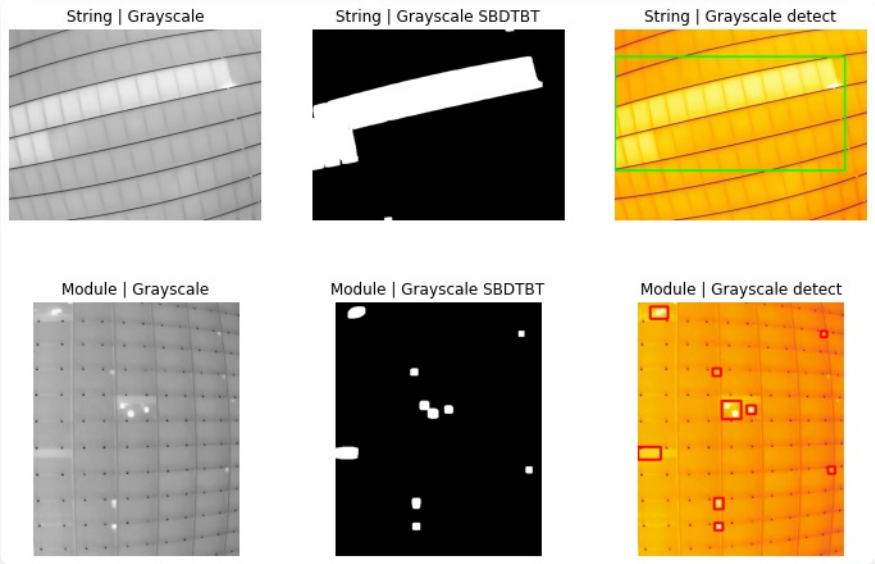
That went pretty well!

There are two tiny errors we couldn't pick up, but further tweaking might fix that. The approach might not be generally applicable across many different kinds of pictures of this nature, so for it to be used in production one would have to either...

- ...standardize how images are taken to reduce distortion, orthogonally over errors and not slanted like here
- ...color correct the image to be more uniform and homogenous
- ...or more dynamically spot areas brighter than the surrounding area, and ditch the absolute range we used here

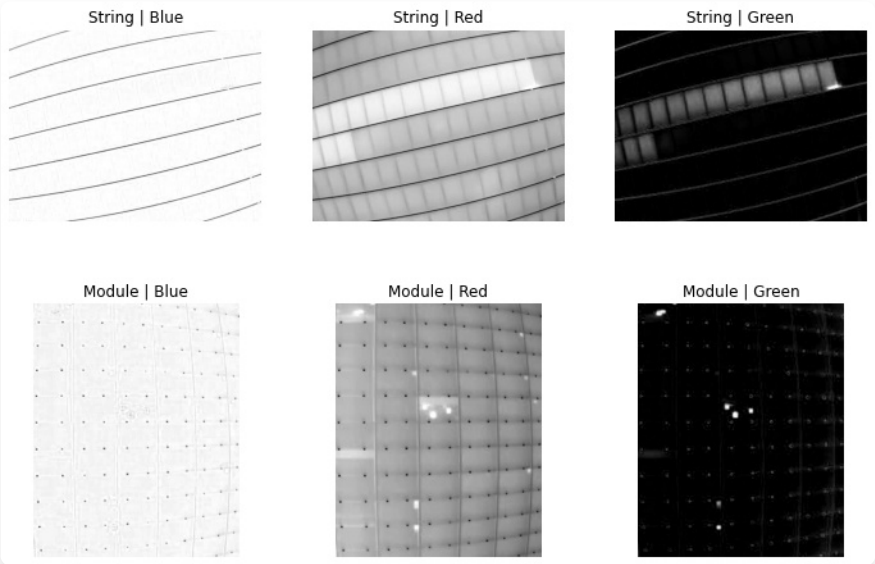
The more dynamic way of dealing with a wider variety of images is the most promising, but we won't implement that in this exercise, as it would require a lot of time and experimentation. Additionally, we don't have any more pictures to test the algorithm, so the foundation to actually carry it out is just not there.

Now, let's see how using a grayscale version of the incomming image performs:



It performs similarly to the color range, but as the masking reveals, it is not as good at detecting modules as the color range - which is understandable, as we do lose some data by mashing all channels into one.

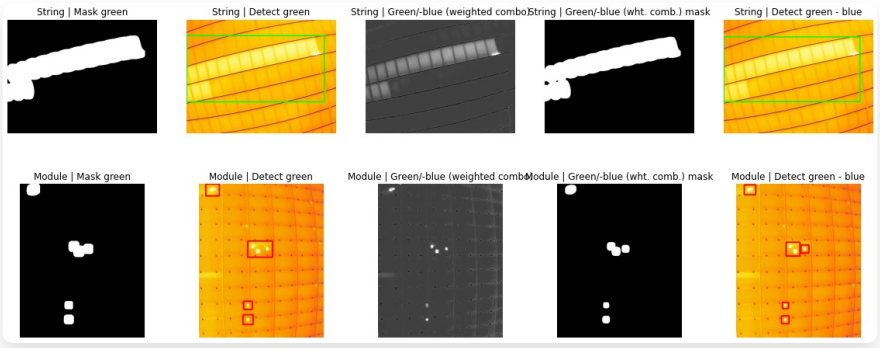
Let's take a deeper dive into the channels and split them up. There may be useful features in each channel that we can use to more easily detect and distinguish the faults.



Look at that! We can easily spot most faults by just using the green channel. There are some possible errors the channel omits, as we can see in the full RGB image or the red channel.

We'll ignore the red channel here, as using it would be basically the same as the color range and grayscale above.

The blue channel seems to hold a lot of information we want to omit, so we can use that to our advantage. If we remove the noise and only keep the pronounced dots and lines, we can then subtract it from the green channel. Alternatively, we can inverse it and average them (or a weighted combination) to keep some information, but still have more pronounced faults.



Even though this approach might be more robust and simple (at least conceptually - the filters got a bit out of hand in the code), it does not beat our color range approach.

So in conclusion, the most promising way forward is to use a dynamic color range approach as mentioned above, to reap the benefits of better detection while being more generally applicable.

Code

Code with more detailed comments below:

```
# %%

"""
INF250 2020 Fall
Exam Exercise 2
"""

from functools import partial, reduce
import matplotlib.image as mpimg
import numpy as np
import cv2
from typing import Any, List, Tuple
from numpy.core.defchararray import mod
from functional_pipeline import pipeline
from pipetools import pipe
from typing_extensions import TypedDict
from PIL import ImageEnhance

from exercise_1 import crop, show
```

```python
def mask(
    image: np.ndarray, color_range=((0, 0, 0), (255, 255, 255))
) -> np.ndarray:
    """
    Masking based on this StackOverflow answer:
    https://stackoverflow.com/a/51702767/11255767
    """
    # First we must copy the image to not mutate the original
    output = np.copy(image)

    # Mask
    mask = cv2.inRange(output, color_range[0], color_range[1])

    # Turn selection white
    output[mask == 0] = [0, 0, 0]
    output[mask != 0] = [255, 255, 255]

    return output



def blur(image: np.ndarray, radius=60) -> np.ndarray:
    x = 1 + radius * 2
    kernel = np.ones((x, x), np.float32) / (x * x)
    output = cv2.filter2D(image, -1, kernel)
    return output


def brightness_contrast(
    image: np.ndarray,
    brightness: float = 0,
    contrast: float = 1.0,
):
    """
    Inspired by this StackOverflow answer:
    https://stackoverflow.com/a/50053219/11255767
    """
    output = image
    output = cv2.addWeighted(image, contrast, image, 0, brightness)
    return output


def sharpen(image: np.ndarray, radius=2) -> np.ndarray:
    x = 1 + radius * 2
    kernel = np.ones((x, x), np.float32) * -1
    kernel[int(x / 2)][int(x / 2)] = x * x
    output = cv2.filter2D(image, -1, kernel)
    return output


def sharpen_xy(image: np.ndarray, x_radius=2, y_radius=2) -> np.ndarray:
    x = 1 + x_radius * 2
    y = 1 + y_radius * 2
    kernel = np.ones((y, x), np.float32) * -1
    kernel[y_radius][x_radius] = x * y
```

```
    output = cv2.filter2D(image, -1, kernel)
    return output


def area_boxes(
    image: np.ndarray,
    detections: List[
        TypedDict(
            "DetectionDict",
            {
                "label": str,
                "range": Tuple[int, int],
                "rectangle_color": Tuple[int, int, int],
            },
        )
    ],
    background: np.ndarray = None,
    thickness=3,
) -> np.ndarray:
    """
    Marks regions of a given size of high values in an image with rectangles
    """

    # image = cv2.imread('bus.png')
    # image = cv2.resize(image,(400,500))
    gray = image

    if type(image) != np.ndarray:
        print(f" recieved image of type {type(image)} - should be np.ndarray
        return np.ndarray([[]])

    if type(image[0][0]) not in [np.uint8, float, int]:
        gray = cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)

    gray = cv2.threshold(gray, 127, 255, 0)[1]

    use_background = type(background) == np.ndarray
    output = background.copy() if use_background else image.copy()

    contours = cv2.findContours(gray, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE
        0
    ]

    for detection in detections:
        for cnt in contours:
            if (
                detection["range"][0]
                < cv2.contourArea(cnt)
                < detection["range"][1]
            ):
                (x, y, w, h) = cv2.boundingRect(cnt)
                cv2.rectangle(
                    output,
                    (x, y),
                    (x + w, y + h),
```

```
                    detection["rectangle_color"],
                    thickness,
                )

    return output


def fault_mask_rgb(image: np.ndarray) -> np.ndarray:
    return pipeline(
        image,
        [
            partial(sharpen, radius=1),
            partial(blur, radius=2),
            partial(mask, color_range=((2, 220, 10), (255, 255, 255))),
            partial(blur, radius=7),
            partial(mask, color_range=((9, 9, 9), (255, 255, 255))),
        ],
    )


def find_faults(
    image: np.ndarray,
    split=0.05,
    show_mask=False,
    image_is_mask=False,
    background=None,
) -> np.ndarray:
    image_area = len(image) * len(image[0])
    split = split * image_area

    return pipeline(
        image,
        [
            partial(lambda x: x) if image_is_mask else fault_mask_rgb,
            partial(
                area_boxes,
                detections=[
                    {
                        "label": "String",
                        "range": (split, image_area),
                        "rectangle_color": (0, 255, 0),
                    },
                    {
                        "label": "Module",
                        "range": (image_area * 0.0005, split),
                        "rectangle_color": (255, 0, 0),
                    },
                ],
                background=background
                if type(background) in [np.ndarray, list]
                else None
                if show_mask
                else image,
            ),
        ],
```

```
    )


if __name__ == "__main__":
    # Import string image
    string = mpimg.imread("./resources/string.jpg")

    # String has a user interface that will interfere with processing,
    # so let's crop that out
    string = crop(string, [0, 0.94], [0.06, 0.95])

    blur_radius = 8
    # Using Windows PowerToys color picker to get values,
    # then tweaking to desired result
    mask_range = ((240, 215, 15), (255, 255, 255))
    mask_binary = ((10, 10, 10), (255, 255, 255))

    # List of image dictionaries to organize and easily loop through
    string_images = [
        {
            "label": "Original",
            "image": string,
        },
        {
            "label": "Mask",
            "image": pipeline(
                string,
                [
                    partial(mask, color_range=mask_range),
                ],
            ),
        },
        {
            "label": "Blur",
            "image": pipeline(
                string,
                [
                    partial(blur, radius=blur_radius),
                ],
            ),
        },
        {
            "label": "Blur -> Mask",
            "image": pipeline(
                string,
                [
                    partial(blur, radius=blur_radius),
                    partial(mask, color_range=mask_range),
                ],
            ),
        },
        {
            "label": "Mask -> Blur -> Mask",
            "image": pipeline(
                string,
```

```
            [
                partial(mask, color_range=mask_range),
                partial(blur, radius=blur_radius),
                partial(mask, color_range=mask_binary),
            ],
        ),
    },
]

# Import module image
module = mpimg.imread("./resources/module.tif")

brightness = -35
contrast = 1.3
contrast_mask = ((255, 220, 0), (255, 255, 255))

sharpen_mask = ((2, 220, 10), (255, 255, 255))
sharpen_radius = 1
blur_radius = 2

# List of image dictionaries to organize and easily loop through
module_images = [
    {
        "label": "Original",
        "image": module,
    },
    {
        "label": "Mask",
        "image": pipeline(
            module,
            [
                partial(mask, color_range=mask_range),
            ],
        ),
    },
    {
        "label": "Blur",
        "image": pipeline(
            module,
            [
                partial(blur, radius=blur_radius),
            ],
        ),
    },
    {
        "label": "Blur -> Mask",
        "image": pipeline(
            module,
            [
                partial(blur, radius=blur_radius),
                partial(mask, color_range=mask_range),
            ],
        ),
    },
    {
```

```
        {
            "label": "Mask -> Blur -> Mask",
            "image": pipeline(
                module,
                [
                    partial(mask, color_range=mask_range),
                    partial(blur, radius=blur_radius),
                    partial(mask, color_range=mask_binary),
                ],
            ),
        },
        {
            "label": "Contrast",
            "image": pipeline(
                module,
                [
                    partial(
                        brightness_contrast,
                        brightness=brightness,
                        contrast=contrast,
                    ),
                ],
            ),
        },
        {
            "label": "Contrast -> Mask",
            "image": pipeline(
                module,
                [
                    partial(
                        brightness_contrast,
                        brightness=brightness,
                        contrast=contrast,
                    ),
                    partial(mask, color_range=contrast_mask),
                ],
            ),
        },
        {
            "label": "Contrast -> Blur",
            "image": pipeline(
                module,
                [
                    partial(blur, radius=blur_radius),
                    partial(
                        brightness_contrast,
                        brightness=brightness,
                        contrast=contrast,
                    ),
                ],
            ),
        },
        {
            "label": "Contrast -> Blur -> Mask",
            "image": pipeline(
                module,
```

```
            [
                partial(
                    brightness_contrast,
                    brightness=brightness,
                    contrast=contrast,
                ),
                partial(blur, radius=blur_radius),
                partial(mask, color_range=contrast_mask),
            ],
        ),
    },
    {
        "label": "Contrast -> Mask -> Blur -> Mask",
        "image": pipeline(
            module,
            [
                partial(
                    brightness_contrast,
                    brightness=brightness,
                    contrast=contrast,
                ),
                partial(mask, color_range=mask_range),
                partial(blur, radius=blur_radius),
                partial(mask, color_range=mask_binary),
            ],
        ),
    },
    {
        "label": "Sharpen",
        "image": pipeline(
            module,
            [
                partial(sharpen, radius=sharpen_radius),
            ],
        ),
    },
    {
        "label": "Sharpen -> Mask",
        "image": pipeline(
            module,
            [
                partial(sharpen, radius=sharpen_radius),
                partial(mask, color_range=sharpen_mask),
            ],
        ),
    },
    {
        "label": "Sharpen -> Blur",
        "image": pipeline(
            module,
            [
                partial(sharpen, radius=sharpen_radius),
                partial(blur, radius=blur_radius),
            ],
        ),
```

```
        },
        {
            "label": "Sharpen -> Blur -> Mask",
            "image": pipeline(
                module,
                [
                    partial(sharpen, radius=sharpen_radius),
                    partial(blur, radius=blur_radius),
                    partial(mask, color_range=sharpen_mask),
                ],
            ),
        },
        {
            "label": "Shrp->Blr->Msk->Blr->Msk",
            "image": pipeline(
                module,
                [
                    partial(sharpen, radius=sharpen_radius),
                    partial(blur, radius=blur_radius),
                    partial(mask, color_range=sharpen_mask),
                    partial(blur, radius=3),
                    partial(mask, color_range=mask_binary),
                ],
            ),
        },
    ]

    show(
        string_images,
        scale=4,
        cols=5,
        savepath="./images/exercise-2-string.jpg",
    )

    show(
        module_images,
        scale=4,
        cols=5,
        savepath="./images/exercise-2-module.jpg",
    )

    detection_images = [
        {
            "label": "SBMBM-detect mask (string)",
            "image": find_faults(string, show_mask=True),
        },
        {
            "label": "SBMBM-detect (string)",
            "image": find_faults(string),
        },
        {
            "label": "SBMBM-detect mask (module)",
            "image": find_faults(module, show_mask=True),
        },
        {
```

```
        "label": "SBMBM-detect (module)",
        "image": find_faults(module),
    },
]

show(
    detection_images,
    scale=4,
    cols=4,
    savepath="./images/exercise-2-sbmbm-detection.jpg",
)

# Workin with two images and duplicate and similar pipelines got messy
# above, so let's tidy things a bit up by separating the original images
# and pipelines. We'll also do method exploration for both images
# simultaneously, so we don't need to find a new compromise/combination
# the end.
originals = [
    {"name": "String", "image": string},
    {"name": "Module", "image": module},
]

def error(x):
    print(min(x.flatten()), max(x.flatten()))
    cv2.threshold(
        x, min(x.flatten()), max(x.flatten()), cv2.THRESH_BINARY
    )[1]

pipepiles_grayscale = [
    {
        "name": "Grayscale",
        "pipe": (
            pipe
            | (
                lambda x: (
                    pipe | (lambda y: cv2.cvtColor(y, cv2.COLOR_RGB2GRAY
                )(x)
            )
        ),
    },
    {
        "name": "Grayscale SBDTBT",
        "pipe": (
            pipe
            | (
                lambda x: (
                    pipe
                    | (lambda y: cv2.cvtColor(y, cv2.COLOR_RGB2GRAY))
                    | (lambda y: sharpen_xy(y, 1, 2))
                    | (lambda y: blur(y, 3))
                    | (lambda y: cv2.fastNlMeansDenoising(y, None, 10))
                    | (
                        lambda y: cv2.threshold(
                            np.array(y), 210, 255, cv2.THRESH_BINARY
                        )[1]
```

```
                    )
                    | (lambda y: blur(y, 6))
                    | (
                        lambda y: cv2.threshold(
                            np.array(y), 5, 255, cv2.THRESH_BINARY
                        )[1]
                    )
                )(x)
            )
        ),
    },
    {
        "name": "Grayscale detect",
        "pipe": (
            pipe
            | (
                lambda x: (
                    pipe
                    | (lambda y: cv2.cvtColor(y, cv2.COLOR_RGB2GRAY))
                    | (lambda y: sharpen_xy(y, 1, 2))
                    | (lambda y: blur(y, 3))
                    | (lambda y: cv2.fastNlMeansDenoising(y, None, 10))
                    | (
                        lambda y: cv2.threshold(
                            np.array(y), 210, 255, cv2.THRESH_BINARY
                        )[1]
                    )
                    | (lambda y: blur(y, 6))
                    | (
                        lambda y: cv2.threshold(
                            np.array(y), 5, 255, cv2.THRESH_BINARY
                        )[1]
                    )
                    | (
                        lambda y: find_faults(
                            cv2.cvtColor(y, cv2.COLOR_GRAY2RGB),
                            background=x,
                            image_is_mask=True,
                        )
                    )
                )(x)
            )
        ),
    },
]

images = []

for o in originals:
    for p in pipepiles_grayscale:
        images.append(
            {
                "label": f"{o['name']} | {p['name']}",
                "image": p["pipe"](o["image"]),
            }
```

```
        )

show(
    images,
    scale=4,
    cols=len(pipepiles_grayscale),
    savepath="./images/exercise-2-grayscale.jpg",
)

pipelines_split = [
    {"name": "Blue", "pipe": (pipe | (lambda x: cv2.split(x)[0]))},
    {"name": "Red", "pipe": (pipe | (lambda x: cv2.split(x)[1]))},
    {"name": "Green", "pipe": (pipe | (lambda x: cv2.split(x)[2]))},
]

images = []

for o in originals:
    for p in pipelines_split:
        images.append(
            {
                "label": f"{o['name']} | {p['name']}",
                "image": p["pipe"](o["image"]),
            }
        )

show(
    images,
    scale=4,
    cols=len(pipelines_split),
    savepath="./images/exercise-2-color-split.jpg",
)

pipelines_split_detect = [
    {
        "name": "Mask green",
        "pipe": (
            pipe
            | (
                lambda z: (
                    pipe
                    | (lambda x: cv2.split(x)[2])
                    | (lambda x: cv2.fastNlMeansDenoising(x, None, 40))
                    | (lambda x: blur(x, 6))
                    | (
                        lambda x: cv2.threshold(
                            x, 35, 255, cv2.THRESH_BINARY
                        )[1]
                    )
                    | (lambda x: blur(x, 10))
                    | (
                        lambda x: cv2.threshold(
                            x, 5, 255, cv2.THRESH_BINARY
                        )[1]
                    )
```

```
                    )(z)
                )
            ),
        },
        {
            "name": "Detect green",
            "pipe": (
                pipe
                | (
                    lambda z: (
                        pipe
                        | (lambda x: cv2.split(x)[2])
                        | (lambda x: cv2.fastNlMeansDenoising(x, None, 40))
                        | (lambda x: blur(x, 6))
                        | (
                            lambda x: cv2.threshold(
                                x, 35, 255, cv2.THRESH_BINARY
                            )[1]
                        )
                        | (lambda x: blur(x, 10))
                        | (
                            lambda x: cv2.threshold(
                                x, 5, 255, cv2.THRESH_BINARY
                            )[1]
                        )
                        | (
                            lambda x: find_faults(
                                cv2.cvtColor(x, cv2.COLOR_GRAY2RGB),
                                background=z,
                                image_is_mask=True,
                            )
                        )
                    )(z)
                )
            ),
        },
        {
            "name": "Green/-blue (weighted combo)",
            "pipe": (
                pipe
                | (
                    lambda z: (
                        pipe
                        | (
                            lambda x: np.array(
                                np.round(
                                    cv2.split(x)[2] * 0.75
                                    + (
                                        pipe
                                        | (
                                            lambda y: cv2.fastNlMeansDenoisi
                                                y, None, 10
                                            )
                                        )
                                        | (
```

```
                                    lambda y: cv2.threshold(
                                        y,
                                        240,
                                        255,
                                        cv2.THRESH_BINARY,
                                    )[1]
                                )
                            )(cv2.split(x)[0])
                            * 0.25
                        ),
                        dtype=np.uint8,
                    )
                )
            )(z)
        )
    ),
},
{
    "name": "Green/-blue (wht. comb.) mask",
    "pipe": (
        pipe
        | (
            lambda z: (
                pipe
                | (
                    lambda x: np.array(
                        np.round(
                            cv2.split(x)[2] * 0.75
                            + (
                                pipe
                                | (
                                    lambda y: cv2.fastNlMeansDenoisi
                                        y, None, 10
                                    )
                                )
                                | (
                                    lambda y: cv2.threshold(
                                        y,
                                        240,
                                        255,
                                        cv2.THRESH_BINARY,
                                    )[1]
                                )
                            )(cv2.split(x)[0])
                            * 0.25
                        ),
                        dtype=np.uint8,
                    )
                )
                | (lambda x: cv2.fastNlMeansDenoising(x, None, 40))
                | (lambda x: blur(x, 6))
                | (
                    lambda x: cv2.threshold(
                        x,
                        85,
```

```
                            255,
                            cv2.THRESH_BINARY,
                        )[1]
                    )
                    | (lambda x: blur(x, 5))
                    | (
                        lambda x: cv2.threshold(
                            x, 5, 255, cv2.THRESH_BINARY
                        )[1]
                    )
                )(z)
            )
        ),
    },
    {
        "name": "Detect green - blue",
        "pipe": (
            pipe
            | (
                lambda z: (
                    pipe
                    | (
                        lambda x: np.array(
                            np.round(
                                cv2.split(x)[2] * 0.75
                                + (
                                    pipe
                                    | (
                                        lambda y: cv2.fastNlMeansDenoisi
                                            y, None, 10
                                        )
                                    )
                                    | (
                                        lambda y: cv2.threshold(
                                            y,
                                            240,
                                            255,
                                            cv2.THRESH_BINARY,
                                        )[1]
                                    )
                                )(cv2.split(x)[0])
                                * 0.25
                            ),
                            dtype=np.uint8,
                        )
                    )
                    | (lambda x: cv2.fastNlMeansDenoising(x, None, 40))
                    | (lambda x: blur(x, 6))
                    | (
                        lambda x: cv2.threshold(
                            x,
                            85,
                            255,
                            cv2.THRESH_BINARY,
                        )[1]
```

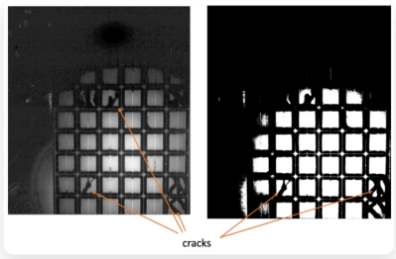
```
                        )
                        | (lambda x: blur(x, 5))
                        | (
                            lambda x: cv2.threshold(
                                x, 5, 255, cv2.THRESH_BINARY
                            )[1]
                        )
                        | (
                            lambda x: find_faults(
                                cv2.cvtColor(x, cv2.COLOR_GRAY2RGB),
                                background=z,
                                image_is_mask=True,
                            )
                        )
                    )(z)
                )
            ),
        },
    ]

    images = []

    for o in originals:
        for p in pipelines_split_detect:
            images.append(
                {
                    "label": f"{o['name']} | {p['name']}",
                    "image": p["pipe"](o["image"]),
                }
            )

    show(
        images,
        scale=4,
        cols=len(pipelines_split_detect),
        savepath="./images/exercise-2-color-split-detect.jpg",
    )
```

Resources

- OpenCV Thresholding
- How to increase contrast (answer from StackOverflow)

Exercise 3

We want to produce two images that facilitate spotting cracks on a solar panel, like so:

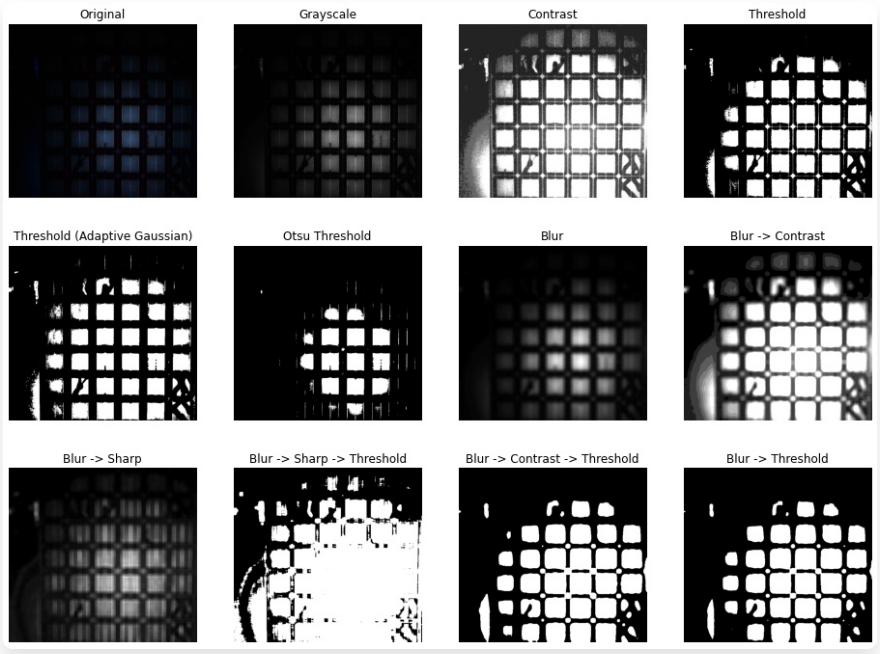


Our starting point is very dark, and features are hard to spot.

To get the original image to look like the ones above, we have to:

1. Grayscale the image
    i. Contrast the image
    Contrasting is increasing the difference between dark and light values in an image with continous values.
    ii. Threshold the image
    Thresholding is splitting dark and light values in an image to binary values, with a given threshold.
    Note: The values are usually split into the minimum- (dark) and the maximum (white) value, but functions can be used in place of minimum and maximum.

As a bonus, we'll explore some more methods and discuss their usability.



It is clear that `Contrast` is the easiest way to make cracks legible for humans, while a combination of blur, contrast and threshold might be a possible foundation to algorithmically find cracks. As there are established ways to classify images, especially binary problems like this, it would probably be best to approach this with an existing solution even though the problem would be possible to handle manually or with a few lines of code.

Code

Code with more detailed comments below:

```
# %%

"""
INF250 2020 Fall
Exam Exercise 3
"""

import numpy as np
import matplotlib.image as mpimg
import cv2

from exercise_1 import show
from exercise_2 import blur, brightness_contrast, sharpen


def compress(
    image: np.ndarray,
    comp_factor=0.25,
    print_resolution=False,
) -> np.ndarray:
    compressed = cv2.resize(
```

```
    compressed = cv2.resize(
        np.copy(image), (0, 0), fx=comp_factor, fy=comp_factor
    )
    if print_resolution:
        print(
            f"{'Original resolution:':24}"
            + f"{len(image):>6}"
            + f"{len(image[0]):>6}"
        )
        print(
            f"{'Compressed resolution:':24}"
            + f"{len(compressed):>6}"
            + f"{len(compressed[0]):>6}"
        )
    return compressed


def threshold(image: np.ndarray) -> np.ndarray:
    _, output = cv2.threshold(image, 8, 255, cv2.THRESH_BINARY)
    return output



def threshold_adaptive_gaussian(image: np.ndarray) -> np.ndarray:
    return cv2.adaptiveThreshold(
        image,
        1,
        cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
        cv2.THRESH_BINARY,
        255,
        -2.5,
    )


def threshold_otsu(image: np.ndarray) -> np.ndarray:
    _, output = cv2.threshold(
        image, -255, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU
    )
    return output


if __name__ == "__main__":
    # Import original image
    original = mpimg.imread("./resources/UVfluor.tif")

    # Compress to speed up processing, and remove redundant data, as the
    # features we are looking for are approximately 50-200 pixels wide in th
    # original image
    comp_factor = 0.25
    compressed = compress(
        original,
        comp_factor=comp_factor,
        print_resolution=True,
    )

    # Grayscale, as we don't need color
```

```
    gray = cv2.cvtColor(compressed, cv2.COLOR_BGR2GRAY)

    blur_radius = 30

    # List of image dictionaries to organize and easily loop through
    images = [
        {
            "label": "Original",
            "image": compressed,
        },
        {
            "label": "Grayscale",
            "image": gray,
        },
        {
            "label": "Contrast",
            "image": brightness_contrast(gray, -35, 35),
        },
        {
            "label": "Threshold",
            "image": threshold(gray),
        },
        {
            "label": "Threshold (Adaptive Gaussian)",
            "image": threshold_adaptive_gaussian(gray),
        },
        {
            "label": "Otsu Threshold",
            "image": threshold_otsu(gray),
        },
        {
            "label": "Blur",
            "image": blur(gray, radius=int(comp_factor * blur_radius)),
        },
        {
            "label": "Blur -> Contrast",
            "image": brightness_contrast(
                blur(gray, radius=int(comp_factor * blur_radius)), 0, 25
            ),
        },
        {
            "label": "Blur -> Sharp",
            "image": blur(
                sharpen(gray), radius=int(comp_factor * blur_radius)
            ),
        },
        {
            "label": "Blur -> Sharp -> Threshold",
            "image": threshold(
                blur(sharpen(gray), radius=int(comp_factor * blur_radius))
            ),
        },
        {
            "label": "Blur -> Contrast -> Threshold",
            "image": threshold(
```

```
                brightness_contrast(
                    blur(gray, radius=int(comp_factor * blur_radius)),
                    contrast=1.1,
                )
            ),
        },
        {
            "label": "Blur -> Threshold",
            "image": threshold(
                blur(gray, radius=int(comp_factor * blur_radius))
            ),
        },
    ]

    show(images, scale=4, cols=4, savepath="./images/exercise-3.jpg")
```

Exercise 4

Here we wil perform spectrum analysis on a band sequential raster file. The code used to carry this out is attached at the bottom of this section.

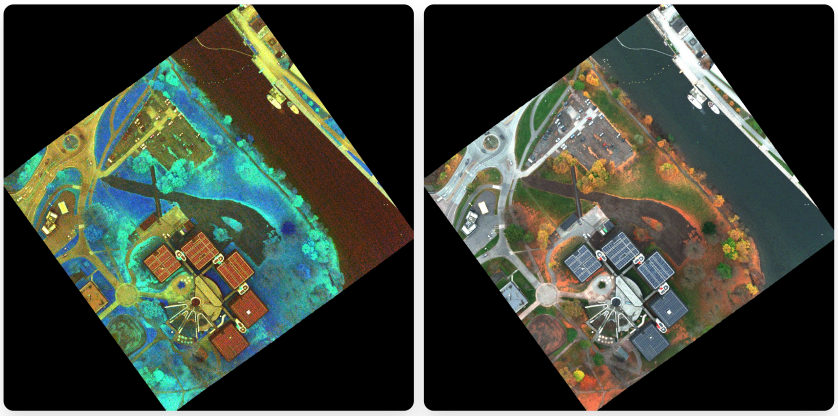
The answer is heavily based on the script `hyperspecdemo_updated.py` available at INF250's Canvas (Fall 2020), but the code is heavily tweaked.

Part 1 - Extract and display RGB-image

As we have 186 wavebands that range outside visible light, the simplest way to emulate an RGB-image is to pick one band in the typical range for each of the three normal color channels, red, green and blue.

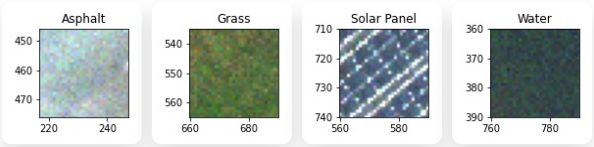
The function to carry this out is inspired by Jon Nordy's implementation, which picks out the nearest band to a given wavelength.

Wavelengths used are derived by taking the center value in each corresponding range.

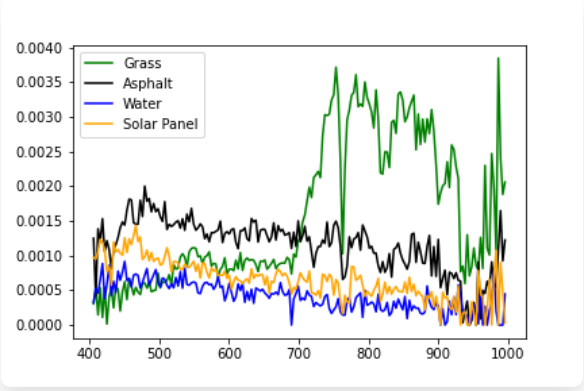


Part 2 - Display spectrum for grass, solar panels, asphalt and water

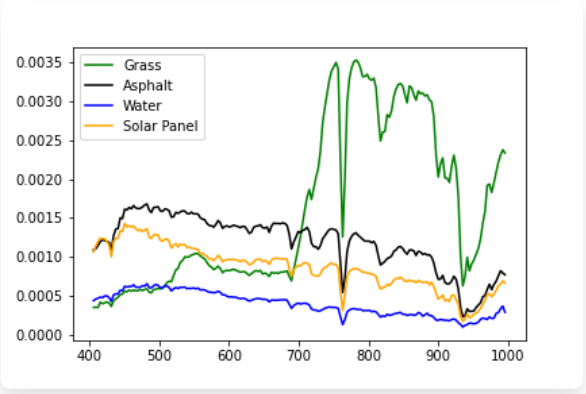
To display spectra for materials, we need to first find coordinates were the material can be found.



Then, we can display the values of all the different wavelengths from those points in a plot.



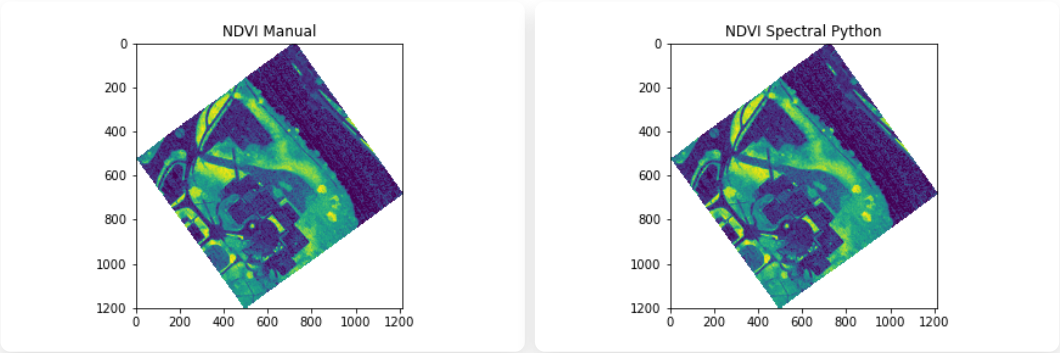
Part 3 - Compute mean spectrum (20x20px) | Comment differences



The mean spectra are smoothed out, as we reduced the noise by taking the mean of a patch of material. The bigger the patch, the smoother we can expect the spectrum to be, as we are reducing detail.

By taking the mean we are generalizing better what the different materials look like, so we are effectively widening the concept of what each material is. In part 2 we were only focusing on one single pixel, which is by far not enough data to define something as general as grass for example (it would be like looking at the tip of a straw and generalizing that to an entire field - not a great idea). This can make it easier to differentiate between the different classes by just looking at the spectra, as illustrated by the graphs.

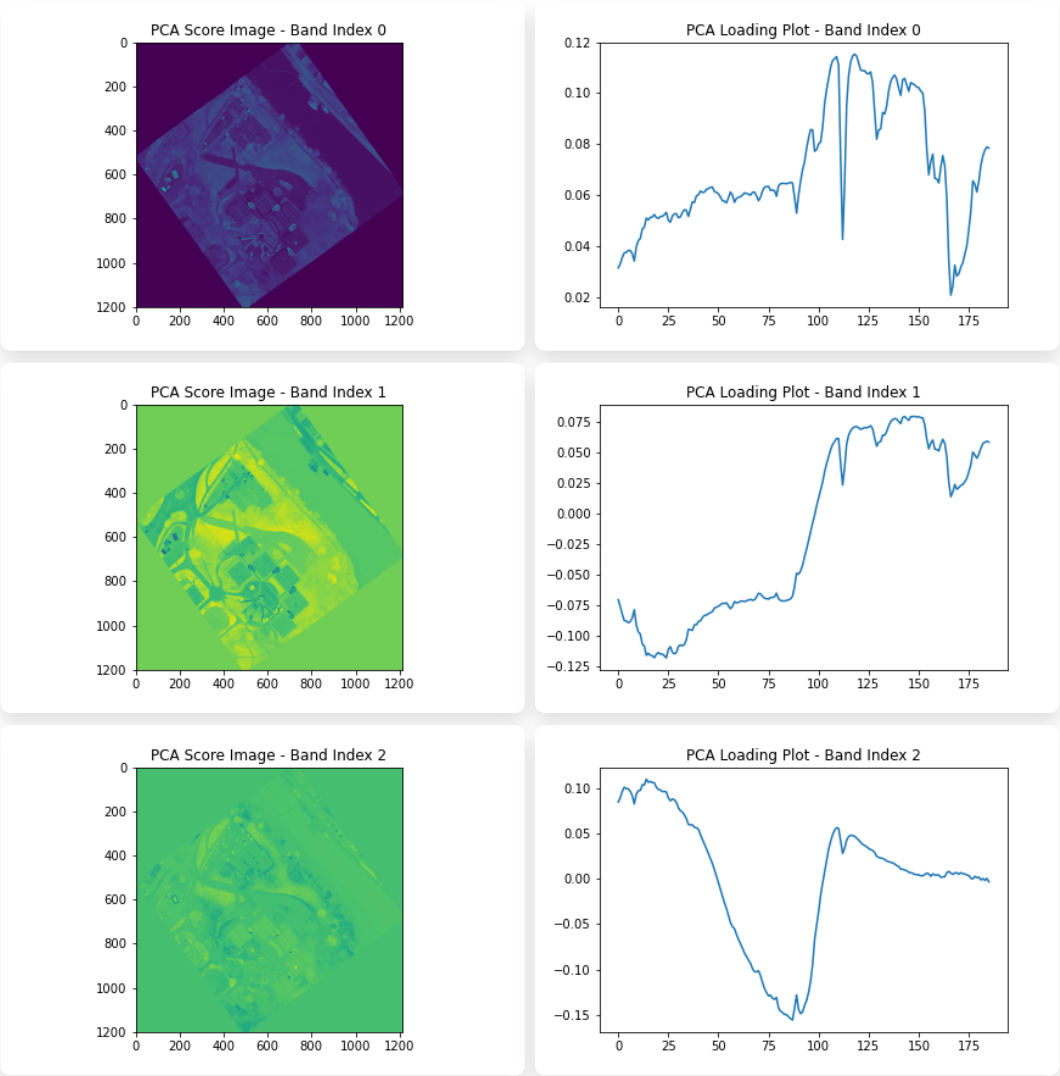
Part 4 - NDVI-image



On the left is a manually computed NDVI-image, and we can see it matches well with Spectral Python's implementation.

NOTE: The manual result had to be inversed to look the same.

Part 5 - PCA | Display 3 first score images and loading plots | Comment | Solar panels?



As PCA is a way to standardize and emphasize variation, and by that a way to extract features, we can see what elements in the image is most strongly represented by the the different bands.

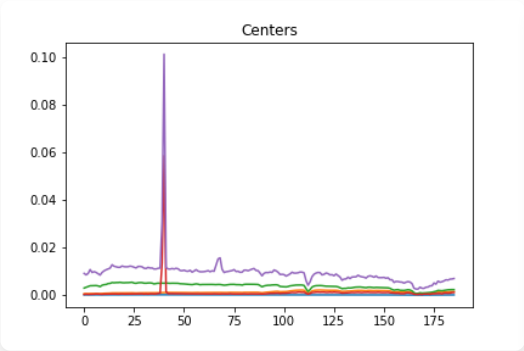
The higher the variety, the more information we can extract from the band.

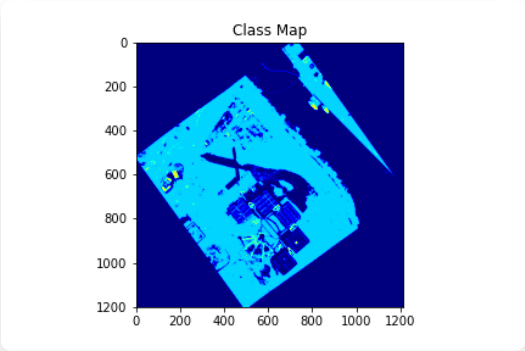
The solar panels aren't especially clear in any of these images (based on the contrast to other elements), but if we go through more of the bands, we might be lucky and find bands where they are more pronounced.

Part 6 - K-means clustering

K-means clustering is a way of automatically finding features or categories in data. We have previously done this manually by picking spots we conceptualize as a feature (material), but now we will ask the algorithm to find distinct features for us.

We will use an arbitrary number of features, 5, and see what comes out.





The results doesn't reflect categories we would choose, as water and dark road are grouped together and grass and pavement are seen as the same thing. The solar panels seems to be pretty distinct, but generally it looks like the K-means result has differentiated between elements in a way that is really not that useful.

This is an image, and as useful classes most likely will be normal concepts for humans, it might make more sense to label manually based on the intended application. Note that K-means can be very useful when looking at complex data where we do not have an intuitive understanding and want to explore and understand more about what we are working with.

What would be interesting is to only perform k-means on wavelengths outside the visible range, as that could reveal something - but we could also just look at the same wavelengths mapped to RGB, which may be faster and more intuitive.

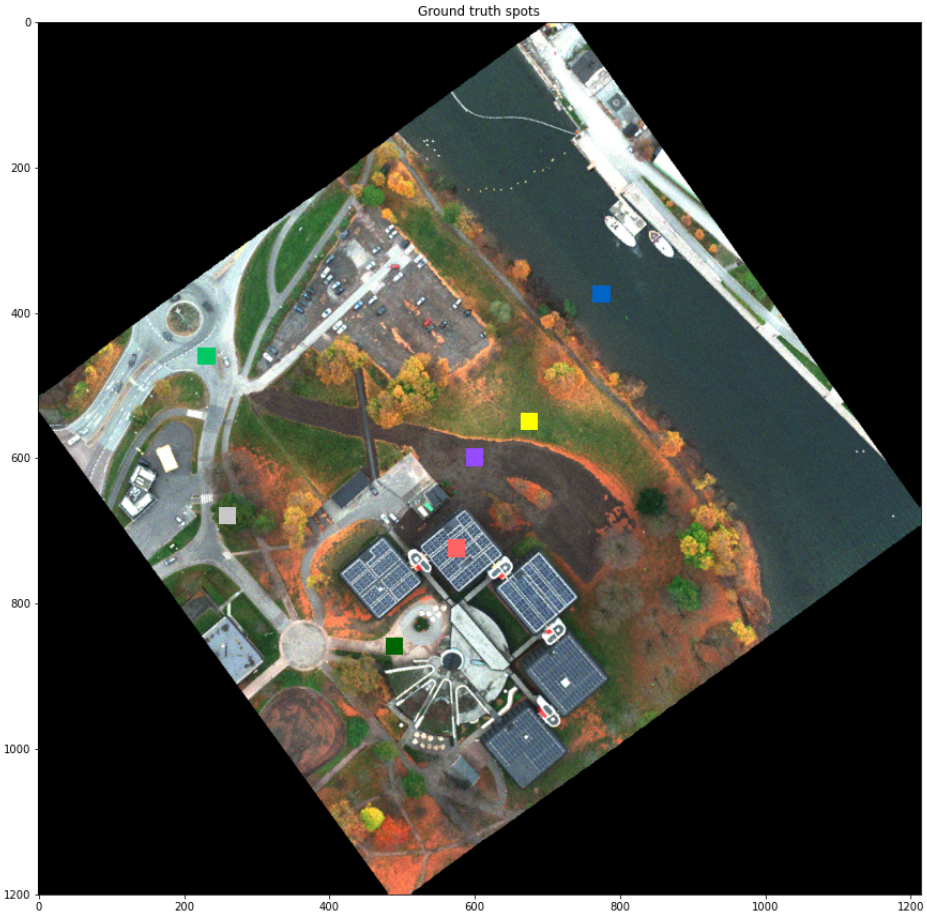
Part 7 - Gaussian maximum likelihood classification | Comment

To perform a classification, we first have to define a ground truth a classifier can base predictions on.

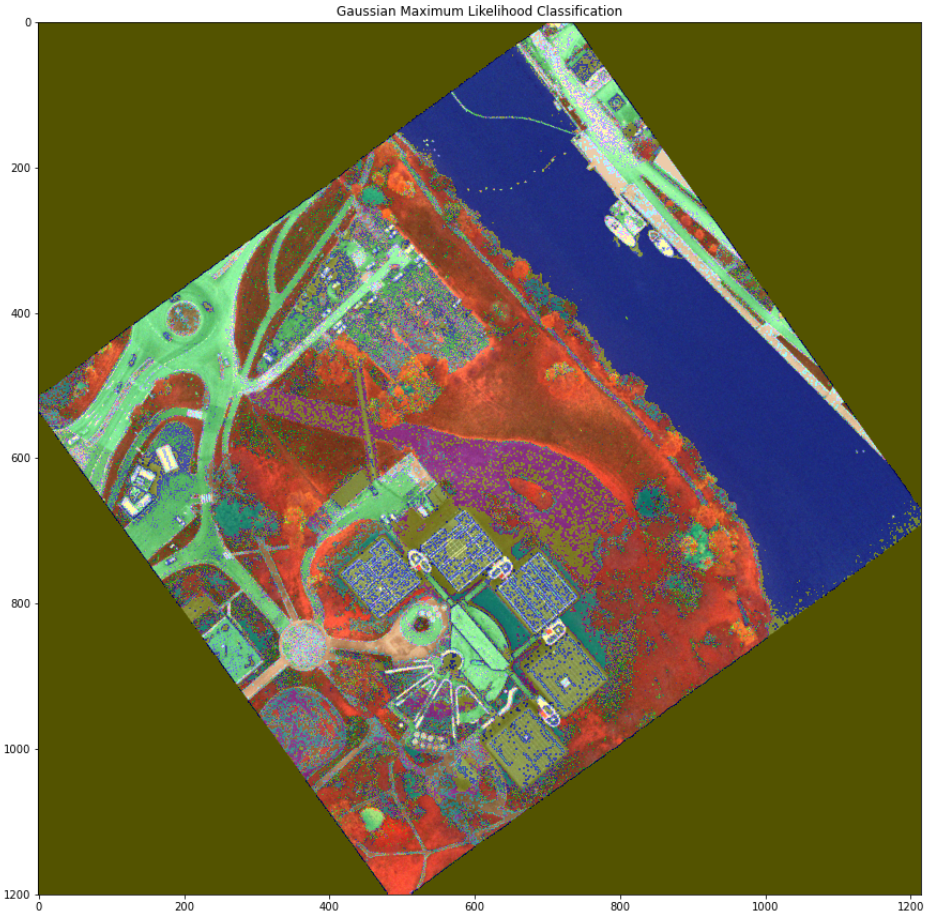
Here we use the following classes:

```
{
    1: 'Grass',
    2: 'Asphalt',
    3: 'Water',
    4: 'Solar Panel',
    5: 'Dark road',
    6: 'Tree',
    7: 'Dirt'
}
```

Classes are based on the following squares of data:



We give the classes as training labels to a gaussian classifier, and predict classes based on the maximum likelihood. The predicted classes are defined as an image we can overlay the original, so we can see how well it performed.



It obviously needs some further tweaking, but all in all that is a pretty good result for a first try.

What we can do to improve detection:

- Make ground truths more general
  - Increase number of patches for ground truths for each class (material)
  - Increase size of patches
  - A caveat here is overfitting - the model will do really well if the whole image is ground truths, but that will probably not generalize across new images
- Post-process the result
  - As it's safe to assume that most classes are a greater continous area, and not noisy like we see in the image above. The classifier has no way to understand this, but we can implement it by substituting noise with the dominant class in an area post-classification

Code

Code with more detailed comments below:

```
# %%
# Info

"""
INF250 2020 Fall
Exam Exercise 4
"""

# %%
# Part 1 - Extract and display RGB-image
# ------------------------------------------------------------------------

from spectral import *
import spectral.io.envi as envi
import matplotlib.pyplot as plt
from typing import List, Tuple, Union
import random

base_path = "./resources/"
header_path = f"{base_path}hyperspectral_vnir.hdr"
output_path = "./images/exercise-4-"

bsq = envi.open(header_path)
print(bsq)

wavelengths = envi.read_envi_header(header_path)["wavelength"]
wavelengths = [float(i) for i in wavelengths]  # list[str] -> list[float]
stretch = (0.02, 0.98)
uni_stretch = (stretch, stretch, stretch)

figsize = (15, 15)

# Throughout the code there will be args used only one place.
# This is intentional, as it makes it easier to export just the image if
# needed.
args = {"data": bsq, "stretch": uni_stretch}
imshow(figsize=figsize, **args)
save_rgb(filename=f"{output_path}original.jpg", **args)


def get_waveband_index(
    band_sequential: io.bsqfile.BsqFile,
    wavelength: Union[int, float],
) -> int:
    """

    Based on Jon Nordby's find_band():
    https://tinyurl.com/yb8tzacx
    """
    diffs = [
```

```
        (i, abs(v - wavelength))
        for i, v in enumerate(band_sequential.bands.centers)
    ]
    nearest = sorted(diffs, key=lambda x: x[1])[0]

    return nearest[0]


def avg(
    elements: Union[
        List[Union[int, float]],
        Tuple[Union[int, float]],
    ],
    return_integer=False,
) -> Union[float, int]:
    average = sum(elements) / len(elements)
    return int(average) if return_integer else average


ri = get_waveband_index(bsq, avg((625, 700)))
gi = get_waveband_index(bsq, avg((500, 565)))
bi = get_waveband_index(bsq, avg((450, 485)))

rgb_bands = (ri, gi, bi)

args = {
    **args,
    "bands": rgb_bands,
}
imshow(figsize=figsize, **args)
save_rgb(filename=f"{output_path}rgb.jpg", **args)

# %%
# Part 2 - Display spectrum for grass, solar panels, asphalt and water
# ----------------------------------------------------------------------
import numpy as np


def get_spectrum(
    band_sequential: io.bsqfile.BsqFile,
    coordinates: Tuple[int, int],
) -> np.ndarray:
    return np.array(
        (band_sequential[coordinates[0], coordinates[1], :].reshape(-1, 1))
    )


materials = [
    {
        "label": "Grass",
        "color": "green",
        "coordinates": (550, 675),
    },
    {
        "label": "Asphalt",
```

```
        "color": "black",
        "coordinates": (461, 232),
    },
    {
        "label": "Water",
        "color": "blue",
        "coordinates": (375, 775),
    },
    {
        "label": "Solar Panel",
        "color": "orange",
        "coordinates": (725, 575),
    },
]

# Find spots with materials
for material in materials:
    view = imshow(figsize=(2, 2), **args)
    view.pan_to(material["coordinates"][0], material["coordinates"][1])
    view.zoom(40)
    view.set_title(material["label"])
    plt.tight_layout()
    plt.savefig(
        output_path
        + "find-materials-"
        + material["label"].lower().replace(" ", "-")
        + ".jpg",
    )


# Plot spectra
plt.figure()

for material in materials:
    plt.plot(
        wavelengths,
        get_spectrum(bsq, material["coordinates"]),
        material["color"],
        label=material["label"],
    )

plt.legend(loc="upper left")
plt.savefig(f"{output_path}material-spectrums.png")

# %%
# Part 3 - Compute mean spectrum (20x20px)
# ----------------------------------------------------------------------


def get_mean_spectrum(
    band_sequential: io.bsqfile.BsqFile,
    coordinates: Tuple[int, int],
    size: Tuple[int, int],
) -> np.ndarray:
    c0i, c0f = (
```

```
        coordinates[0] - int(size[1] / 2 + 0.5),
        coordinates[0] + int(size[1] / 2 + 0.5),
    )
    c1i, c1f = (
        coordinates[1] - int(size[1] / 2 + 0.5),
        coordinates[1] + int(size[1] / 2 + 0.5),
    )

    area = band_sequential[
        c0i:c0f,
        c1i:c1f,
        :,
    ]

    mean = area.mean(axis=0).mean(axis=0)

    return np.array(mean)


# Plot spectra
plt.figure()

for material in materials:
    plt.plot(
        wavelengths,
        get_mean_spectrum(bsq, material["coordinates"], (20, 20)),
        material["color"],
        label=material["label"],
    )

plt.legend(loc="upper left")
plt.savefig(f"{output_path}material-spectrums-mean-20x20.png")

# %%
# Part 4 - NDVI-image
# ---------------------------------------------------------------------------

ni = get_waveband_index(bsq, avg((750, 2500)))
ndvi_plot_args = {
    "vmin": -0.5,
    "vmax": 0.7,
}

# ndvi manually
hyper_img = bsq[:, :, :]
ndvi_img = (
    (hyper_img[:, :, ri] - hyper_img[:, :, ni])
    / (hyper_img[:, :, ri] + hyper_img[:, :, ni])
    * -1  # Inverse values to match spectral python
)
plt.figure()
plt.title("NDVI Manual")
plt.imshow(ndvi_img, **ndvi_plot_args)
plt.savefig(f"{output_path}ndvi-manual.png")
```

```python
# # ndvi with spectral python
ndvi_img_spy = ndvi(hyper_img, ri, ni)

plt.figure()
plt.title("NDVI Spectral Python")
plt.imshow(ndvi_img_spy, **ndvi_plot_args)
plt.savefig(f"{output_path}ndvi-spectral-python.png")

# %%
# Part 5 - PCA | Display 3 first score images and loading plots
# ----------------------------------------------------------------------------
pca = principal_components(hyper_img)
pca_reduced = pca.reduce(fraction=0.99)

pca_img = pca_reduced.transform(hyper_img)
loadings = pca_reduced.eigenvectors

for i in range(3):
    pca_plot_args = {
        # Disabled colormap normalization to improve eligibility of image
        # "vmin": -0.1,
        # "vmax": 0.15,
    }
    plt.figure()
    plt.title(f"PCA Score Image - Band Index {i}")
    plt.imshow(pca_img[:, :, i], **pca_plot_args)
    plt.savefig(f"{output_path}pca-score-{i + 1}.png")

    plt.figure()
    plt.title(f"PCA Loading Plot - Band Index {i}")
    plt.plot(loadings[:, i])
    plt.savefig(f"{output_path}pca-loading-{i + 1}.png")

# Loadings plots

# %%
# Part 6 - K-means clustering
# ----------------------------------------------------------------------------
class_map, centers = kmeans(bsq, 5, 30)

# %%
plt.figure()
plt.title("Class Map")
plt.imshow(class_map, "jet")
plt.savefig(f"{output_path}kmeans-class-map.png")

plt.figure()
plt.title("Centers")
for i in range(centers.shape[0]):
    plt.plot(centers[i])
plt.savefig(f"{output_path}kmeans-centers.png")

# %%
# Part 7 - Gaussian maximum likelihood classification
# ----------------------------------------------------------------------------
```

```
# Ground truth
class_materials = np.concatenate(
    (
        np.copy(materials),
        np.array(
            [
                {"label": "Dark road", "coordinates": (600, 600)},
                {"label": "Tree", "coordinates": (680, 260)},
                {"label": "Dirt", "coordinates": (860, 490)},
            ]
        ),
    )
)


def get_ground_truth(materials, hyper_img, size=(20, 20)) -> None:
    ground_truth = np.zeros([hyper_img.shape[0], hyper_img.shape[1]])
    ground_truth_dict = {}

    for i, material in enumerate(materials):
        c0i, c0f = (
            material["coordinates"][0] - int(size[1] / 2 + 0.5),
            material["coordinates"][0] + int(size[1] / 2 + 0.5),
        )
        c1i, c1f = (
            material["coordinates"][1] - int(size[1] / 2 + 0.5),
            material["coordinates"][1] + int(size[1] / 2 + 0.5),
        )

        # value = round((i + 0.37) / len(materials), 2)
        value = i + 1
        ground_truth[c0i:c0f, c1i:c1f] = value
        ground_truth_dict[value] = material["label"]

    return ground_truth, ground_truth_dict


ground_truth, ground_truth_dict = get_ground_truth(
    class_materials, hyper_img, (25, 25)
)

print(ground_truth_dict)

view = imshow(
    figsize=figsize, **args, classes=(ground_truth) / (len(materials) * 2.3)
)
view.set_title("Ground truth spots")
view.set_display_mode("overlay")
view.class_alpha = 1
plt.savefig(f"{output_path}ground-truth.png")

# %%
classes = create_training_classes(hyper_img, ground_truth)
gmlc = GaussianClassifier(classes)
```

```
clmap = gmlc.classify_image(hyper_img)

# %%
view = imshow(figsize=figsize, **args, classes=clmap)
view.class_alpha = 0.333
view.set_title("Gaussian Maximum Likelihood Classification")
view.set_display_mode("overlay")
plt.savefig(f"{output_path}classification.png")
```

Resources

- Spectral Python
- BIL, BIP and BSQ raster files