Project work: A mini segmentation challenge

Imaging for the Life Sciences
MSLS / CO4: Project work

Student: ⇒ Mirco Blaser

University: ⇒ ZHAW

Semester: ⇒ 4th Semester

Date: ⇒ June 3rd 2024

Github repository: ⇒ https://github.com/denacem/waterbodies

Table of contents

Prerequisites / Setup

⇒ Special setup instructions, imports and configurations go here.

In [16]:
```
import numpy as np
import matplotlib.pyplot as plt
import cv2 as cv
import nibabel as nib
import pydicom
```

```
import PIL
from PIL import Image
import os
import pandas as pd
from scipy.optimize import linear_sum_assignment

# Smaller Matplotlib titles for PDF print
plt.rcParams['axes.titlesize'] = 'medium'

# Jupyter / IPython configuration:
# Automatically reload modules when modified
%load_ext autoreload
%autoreload 2

# Enable vectorized output (for nicer plots)
%config InlineBackend.figure_formats = ["svg"]

# Inline backend configuration
%matplotlib inline

# Enable this line if you want to use the interactive widgets
# It requires the ipympl package to be installed.
#%matplotlib widget

import sys
sys.path.insert(0, "../")
import tools

# Number of samples to create for the whole code
num_samples = 3
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Dataset

Title: Satellite Images of Water Bodies

Source: Kaggle

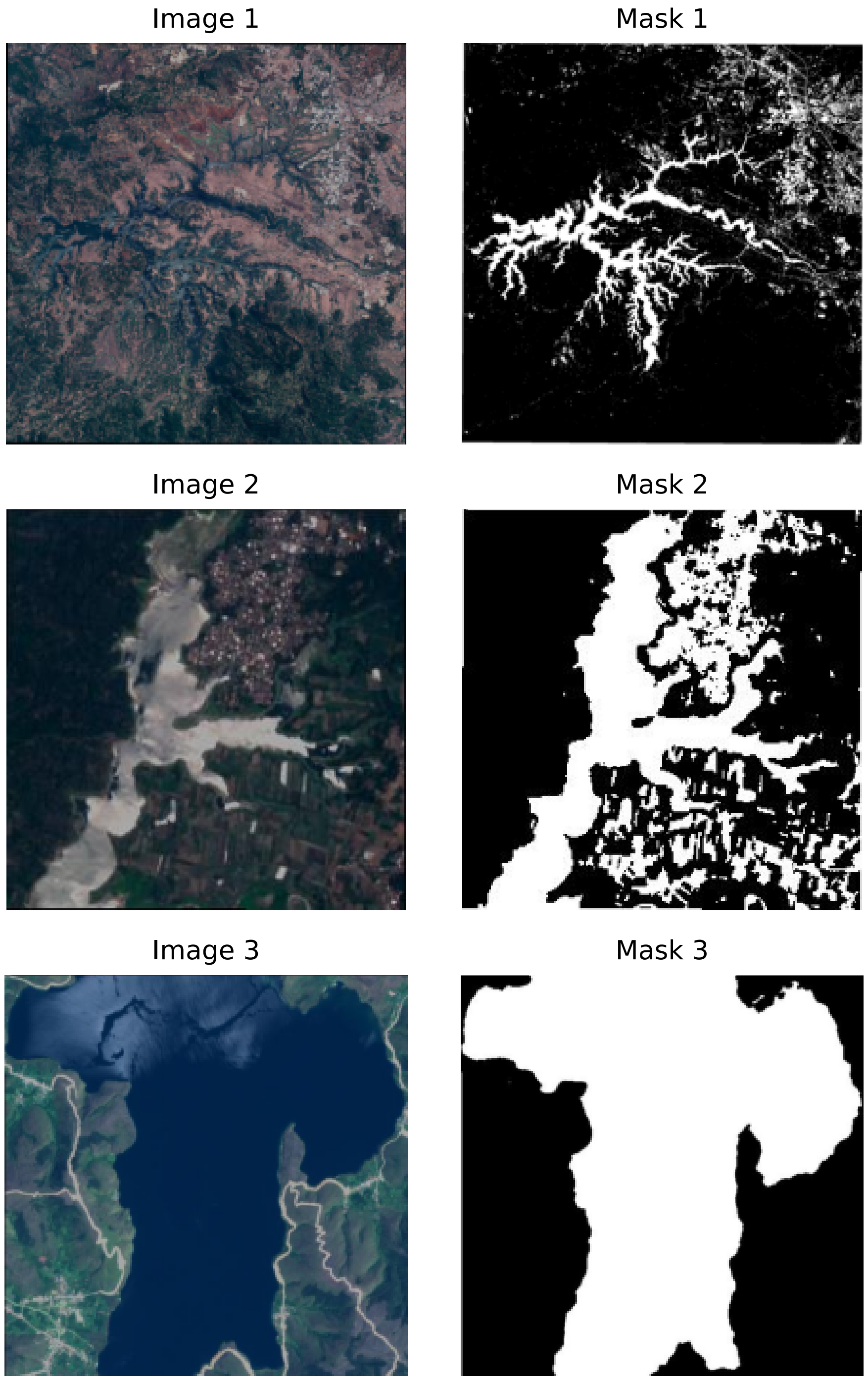
Description: A collection of water bodies images captured by the Sentinel-2 Satellite. Each image comes with a black and white mask where white represents water and black represents something else but water. The masks were generated by calculating the NWDI (Normalized Water Difference Index) which is frequently used to detect and measure vegetation in satellite images, but a greater threshold was used to detect water bodies.

- **Images**: These are the raw satellite images.
- **Masks**: These are the binary masks where water bodies are labeled.

Below are examples of the images and their corresponding masks:

In [8]:
```
# Paths to the directories containing images and masks
images_path = './data/Images'
masks_path = './data/Masks'

# Function to load images and masks
def load_images_masks(image_dir, mask_dir, num_samples=None, batch_size=100):
    image_files = sorted(os.listdir(image_dir))
    mask_files = sorted(os.listdir(mask_dir))

    num_samples = num_samples or len(image_files)

    images = []
    masks = []

    for i in range(0, num_samples, batch_size):
        image_batch_files = image_files[i:i+batch_size]
        mask_batch_files = mask_files[i:i+batch_size]

        image_batch = []
        mask_batch = []

        for image_file, mask_file in zip(image_batch_files, mask_batch_files):
            with Image.open(os.path.join(image_dir, image_file)) as image:
                image_batch.append(image.copy())

            with Image.open(os.path.join(mask_dir, mask_file)) as mask:
                mask_batch.append(mask.copy())

        images.extend(image_batch)
        masks.extend(mask_batch)

    return images, masks

# Load the first num_samples images and masks
images, masks = load_images_masks(images_path, masks_path, num_samples)

# Display the images and masks
fig, axs = plt.subplots(num_samples, 2, figsize=(5, num_samples*2.5))
```

```python
for i in range(num_samples):
    axs[i, 0].imshow(images[i])
    axs[i, 0].set_title(f'Image {i+1}')
    axs[i, 0].axis('off')

    axs[i, 1].imshow(masks[i], cmap='gray')
    axs[i, 1].set_title(f'Mask {i+1}')
    axs[i, 1].axis('off')

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



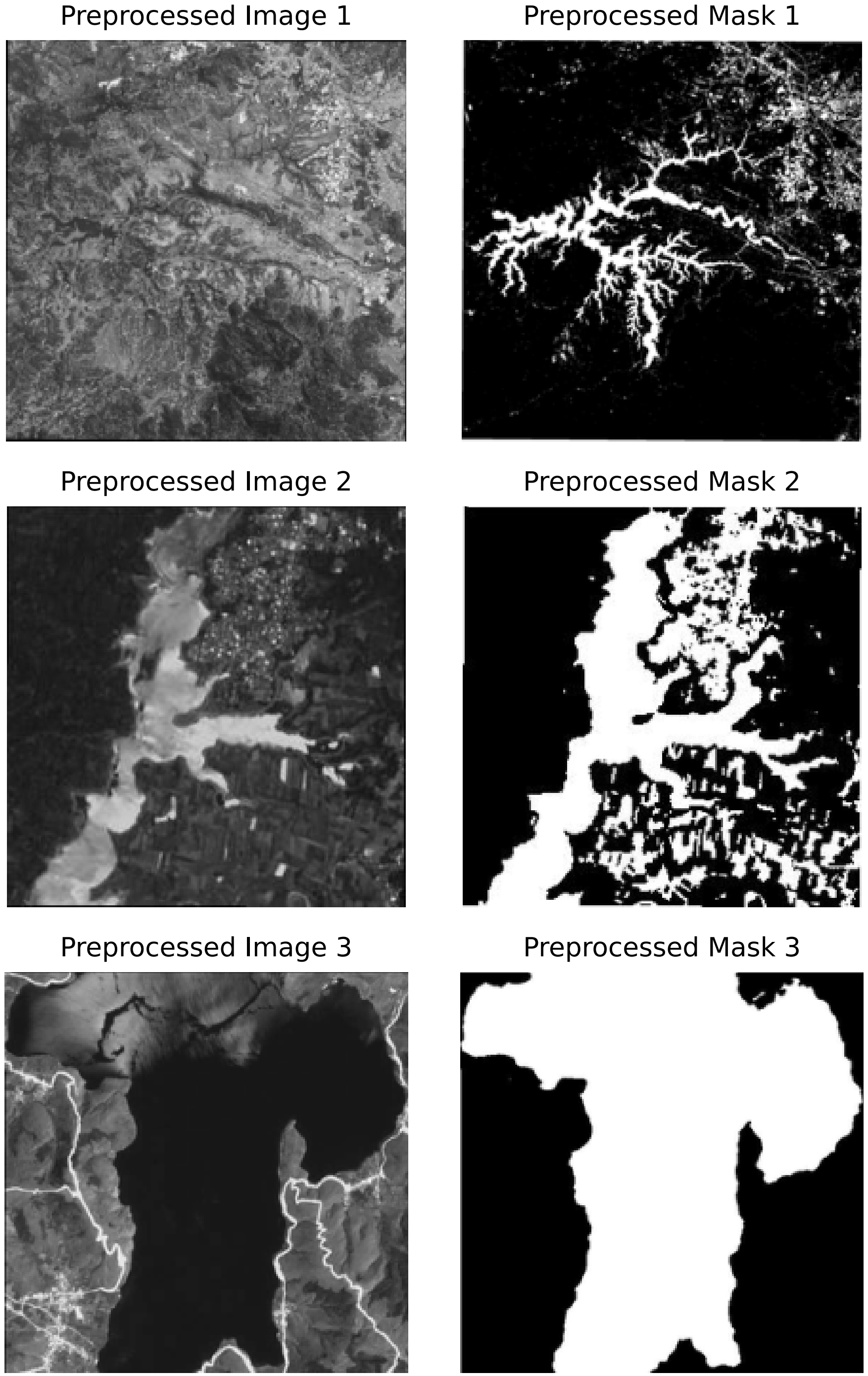
Preprocessing

1. **Resizing**: Standardized the image sizes to 256x256 pixels.
2. **Normalization**: Adjusted the pixel values to the range [0, 1].
3. **Contrast Enhancement**: Applied histogram equalization to enhance the contrast of the images.

Below are the examples of the preprocessed images and their corresponding masks:

In [9]:
```
import PIL
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt

def preprocess_image(image, ismask):
    target_width = 256

    # Check if the image has valid dimensions
    width, height = image.size

    # Calculate the target height to maintain the original aspect ratio
    target_height = int(height * (target_width / width))

    # Fix because some crazy distorted images get a height of 0 and mess things up
    if target_height == 0:
        target_height = 1

    # Resize the image while maintaining aspect ratio
    image_resized = image.resize((target_width, target_height), PIL.Image.LANCZOS)

    if not ismask:

        # Convert the image to a numpy array and then to grayscale directly
        image_array = np.array(image_resized)
        image_grayscale = cv.cvtColor(image_array, cv.COLOR_BGR2GRAY)
        image_enhanced = image_grayscale
    else:
        image_enhanced = np.array(image_resized.convert('L'))

    return image_enhanced

# Preprocess the first num_samples images and masks
preprocessed_images = [preprocess_image(images[i], ismask=False) for i in range(num_samples)]
preprocessed_masks = [preprocess_image(masks[i], ismask=True) for i in range(num_samples)]

# Display the preprocessed images and masks
fig, axs = plt.subplots(num_samples, 2, figsize=(5, num_samples*2.5))
```

```python
for i in range(num_samples):
    axs[i, 0].imshow(preprocessed_images[i], cmap='gray')
    axs[i, 0].set_title(f'Preprocessed Image {i+1}')
    axs[i, 0].axis('off')

    axs[i, 1].imshow(preprocessed_masks[i], cmap='gray')
    axs[i, 1].set_title(f'Preprocessed Mask {i+1}')
    axs[i, 1].axis('off')

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



Manual segmentation

For the manual segmentation, the Fiji (ImageJ) software was used. Fiji is an open-source image processing package that is widely used in the life sciences for its powerful and user-friendly tools.

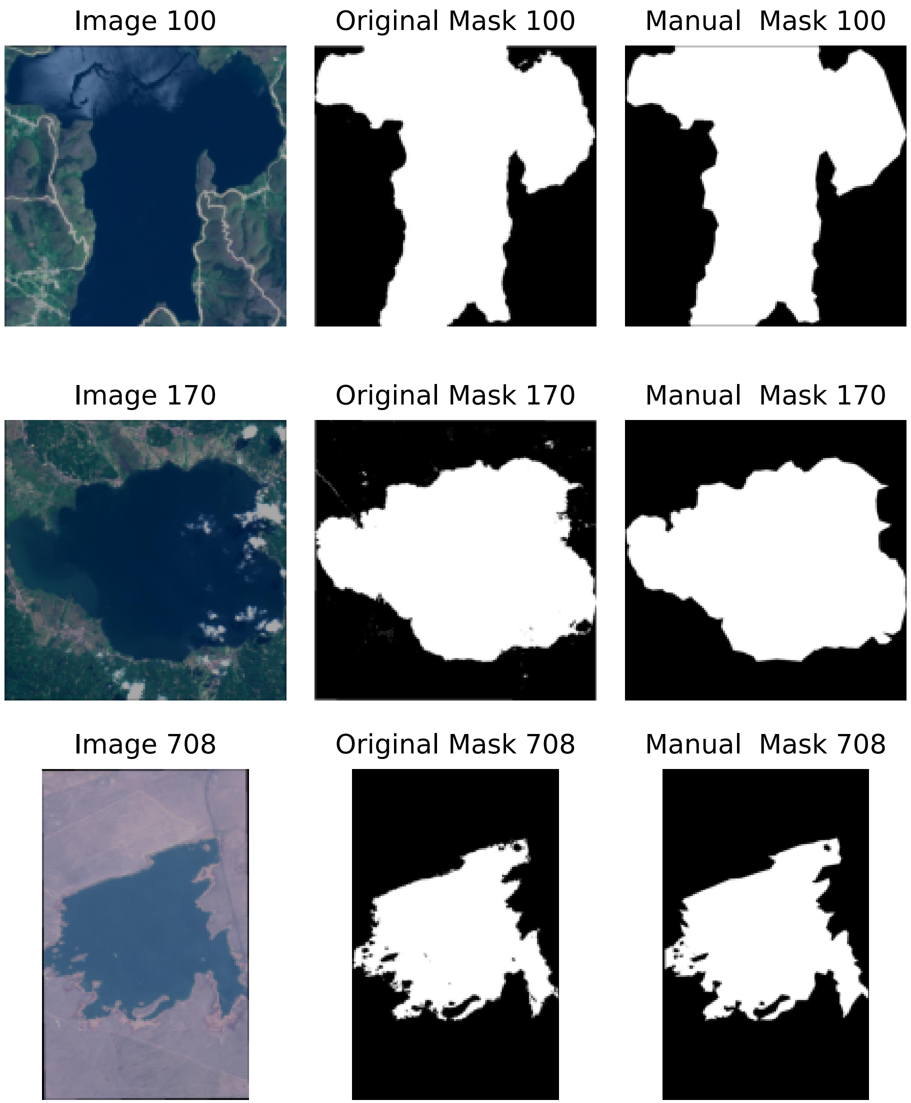
Steps for Manual Segmentation in Fiji:

1. **Open Image**: Load the image into Fiji by going to `File > Open...` and selecting the image file.
2. **Select the Region of Interest**: Use the Polygon selection tool to carefully outline the water body in the image.
3. **Create a Mask**: Once the water body is selected, go to Edit > Selection > Create Mask. This creates a binary mask where the selected region is white, and the rest is black.
4. **Save Mask**: Save the resulting mask by going to `File > Save As > PNG...`.

The following code displays the original images, the original masks, and the manually segmented masks for the selected images (100, 170, and 708).

In [10]:
```
# Paths to the directories containing images and masks
images_path = './data/Images/'
masks_path = './data/Masks/'
manual_masks_path = './manual/'

# List of specific image indices to load
image_indices = [100, 170, 708]

# Function to load specific images and masks
def load_specific_images_masks(image_dir, mask_dir, manual_mask_dir, indices):
    images = []
    masks = []
    manual_masks = []
    for index in indices:
        image_file = f'water_body_{index}.jpg'
        mask_file = f'water_body_{index}.jpg'
        manual_mask_file = f'water_body_{index}.png'
        image_path = os.path.join(image_dir, image_file)
        mask_path = os.path.join(mask_dir, mask_file)
        manual_mask_path = os.path.join(manual_mask_dir, manual_mask_file)
        images.append(Image.open(image_path))
        masks.append(Image.open(mask_path))
        manual_masks.append(Image.open(manual_mask_path))

    return images, masks, manual_masks

# Load the specific images and masks
images, masks, manual_masks = load_specific_images_masks(images_path, masks_path, manual_masks_path, image_indices)
```

```python
# Display the images and masks
fig, axs = plt.subplots(len(images), 3, figsize=(5, 2 * len(images)))

for i in range(len(images)):
    axs[i, 0].imshow(images[i])
    axs[i, 0].set_title(f'Image {image_indices[i]}')
    axs[i, 0].axis('off')

    axs[i, 1].imshow(masks[i], cmap='gray')
    axs[i, 1].set_title(f'Original Mask {image_indices[i]}')
    axs[i, 1].axis('off')

    axs[i, 2].imshow(manual_masks[i], cmap='gray')
    axs[i, 2].set_title(f'Manual  Mask {image_indices[i]}')
    axs[i, 2].axis('off')

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



Automated segmentation

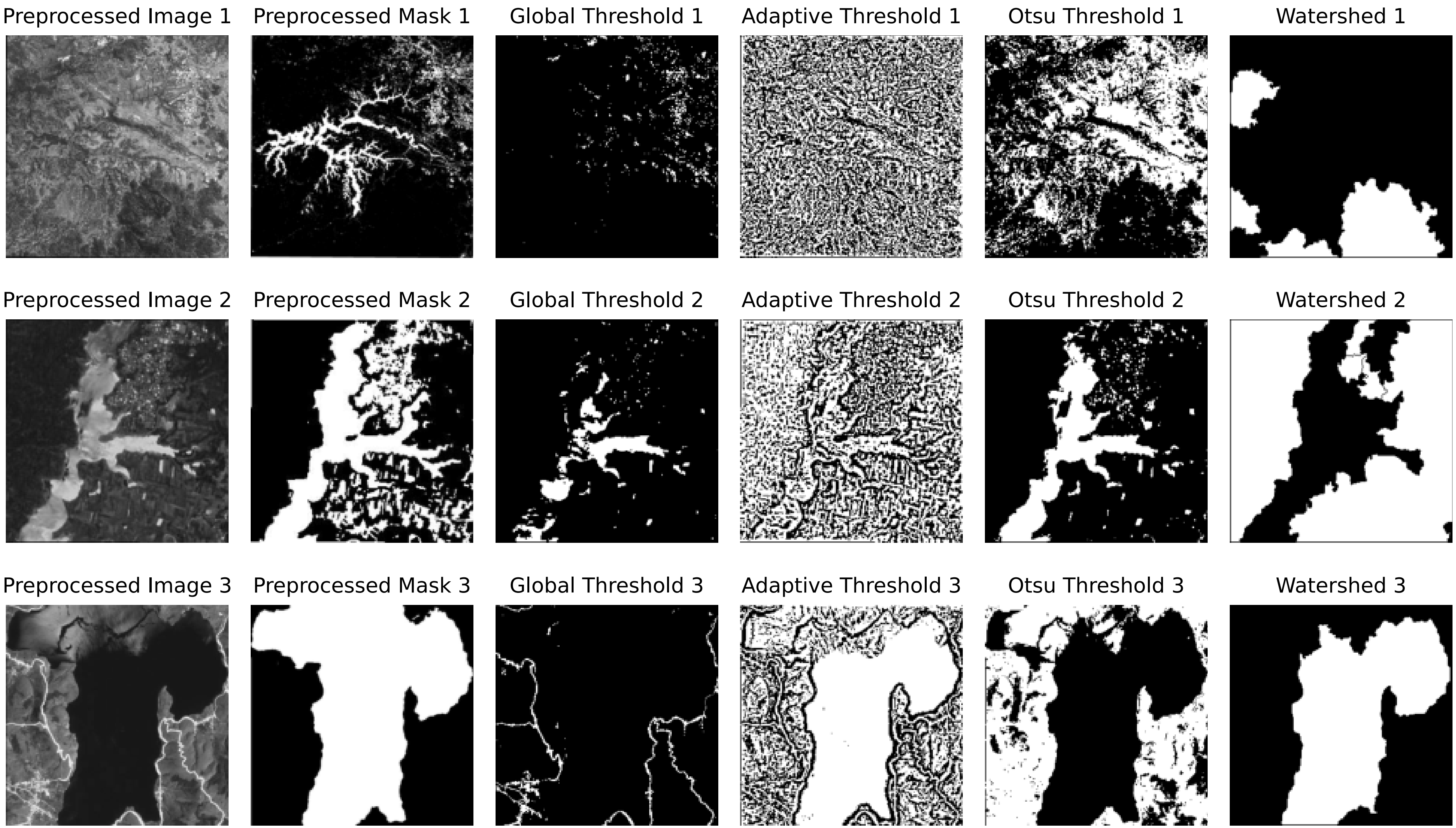
For the automated segmentation I created a segment() function that takes an image and a method and creates a segmented mask with the CV library. For watershed a separate function is created.

The function is called for the number of samples and run for all the four methods. Finally the results are visualised next to the original images and masks.

In [11]:
```python
# Define the segmentation function
def segment(image, method='global'):

    # Check the number of channels in the image
    if len(image.shape) == 3 and image.shape[2] == 3:
        # Convert the image to grayscale if it has 3 channels
        gray = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    else:
        # Image is already in grayscale
        gray = image

    if method == 'global':
        # Apply global thresholding
        _, mask = cv.threshold(gray, 127, 255, cv.THRESH_BINARY)
    elif method == 'adaptive':
        # Apply adaptive thresholding
        mask = cv.adaptiveThreshold(gray, 255, cv.ADAPTIVE_THRESH_GAUSSIAN_C, cv.THRESH_BINARY, 11, 2)
    elif method == 'otsu':
        # Apply Otsu's thresholding
        _, mask = cv.threshold(gray, 0, 255, cv.THRESH_BINARY + cv.THRESH_OTSU)
    elif method == 'watershed':
        # Apply watershed segmentation
        markers, img = segment_watershed(image)
        # Create a binary mask where the segmented regions are white
        mask = np.zeros_like(gray)
        mask[markers > 1] = 255  # We assume the labels greater than 1 correspond to foreground regions
    else:
        raise ValueError(f"Unknown method: {method}")

    return mask

def segment_watershed(img):
    img = cv.GaussianBlur(img, (5, 5), 0)
    img_blur = cv.medianBlur(img, 5)

    # Convert the image to grayscale
    if len(img_blur.shape) == 3 and img_blur.shape[2] == 3:
        gray = cv.cvtColor(img_blur, cv.COLOR_RGB2GRAY)
    else:
        gray = img_blur

    ret, thresh = cv.threshold(gray, 0, 255, cv.THRESH_BINARY_INV + cv.THRESH_OTSU)

    # Noise removal
    kernel = np.ones((3, 3), np.uint8)
    opening = cv.morphologyEx(thresh, cv.MORPH_OPEN, kernel, iterations=9)
```

```
    # Sure background area
    sure_bg = cv.dilate(opening, kernel, iterations=3)

    # Finding sure foreground area
    dist_transform = cv.distanceTransform(opening, cv.DIST_L2, 5)
    thr = 18
    ret, sure_fg = cv.threshold(dist_transform, thr, 255, 0)

    # Finding unknown region
    sure_fg = np.uint8(sure_fg)
    unknown = cv.subtract(sure_bg, sure_fg)

    # Marker labelling
    ret, markers = cv.connectedComponents(sure_fg)

    # Add one to all labels so that sure background is not 0, but 1
    markers = markers + 1

    # Now, mark the region of unknown with zero
    markers[unknown == 255] = 0

    # Ensure the image is in color
    if len(img.shape) == 2 or img.shape[2] != 3:
        img = cv.cvtColor(img, cv.COLOR_GRAY2BGR)

    markers = cv.watershed(img, markers)
    img[markers == -1] = [255, 0, 0]

    return markers, img

# Segment the preprocessed images using different methods
segmented_masks_otsu = [segment(image, method='otsu') for image in preprocessed_images]
segmented_masks_adaptive = [segment(image, method='adaptive') for image in preprocessed_images]
segmented_masks_global = [segment(image, method='global') for image in preprocessed_images]
segmented_masks_watershed = [segment(image, method='watershed') for image in preprocessed_images]

# Display the results
fig, axs = plt.subplots(num_samples, 6, figsize=(10, num_samples * 2))

for i in range(num_samples):
    axs[i, 0].imshow(preprocessed_images[i], cmap='gray')
    axs[i, 0].set_title(f'Preprocessed Image {i+1}')
    axs[i, 0].axis('off')

    axs[i, 1].imshow(preprocessed_masks[i], cmap='gray')
```

```
    axs[i, 1].set_title(f'Preprocessed Mask {i+1}')
    axs[i, 1].axis('off')

    axs[i, 2].imshow(segmented_masks_global[i], cmap='gray')
    axs[i, 2].set_title(f'Global Threshold {i+1}')
    axs[i, 2].axis('off')

    axs[i, 3].imshow(segmented_masks_adaptive[i], cmap='gray')
    axs[i, 3].set_title(f'Adaptive Threshold {i+1}')
    axs[i, 3].axis('off')

    axs[i, 4].imshow(segmented_masks_otsu[i], cmap='gray')
    axs[i, 4].set_title(f'Otsu Threshold {i+1}')
    axs[i, 4].axis('off')

    axs[i, 5].imshow(segmented_masks_watershed[i], cmap='gray')
    axs[i, 5].set_title(f'Watershed {i+1}')
    axs[i, 5].axis('off')

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



Evaluation

For the evaluation, first of all the segmented masks for a larger set is images is created.

The dice() function from the module repository is used to calculate the dice coefficients. The function is called for every mask, creating a dictionary with all the coefficients. Finally the mean and standard deviations for every method is calculated and visualised in an error bar chart.

In [13]:

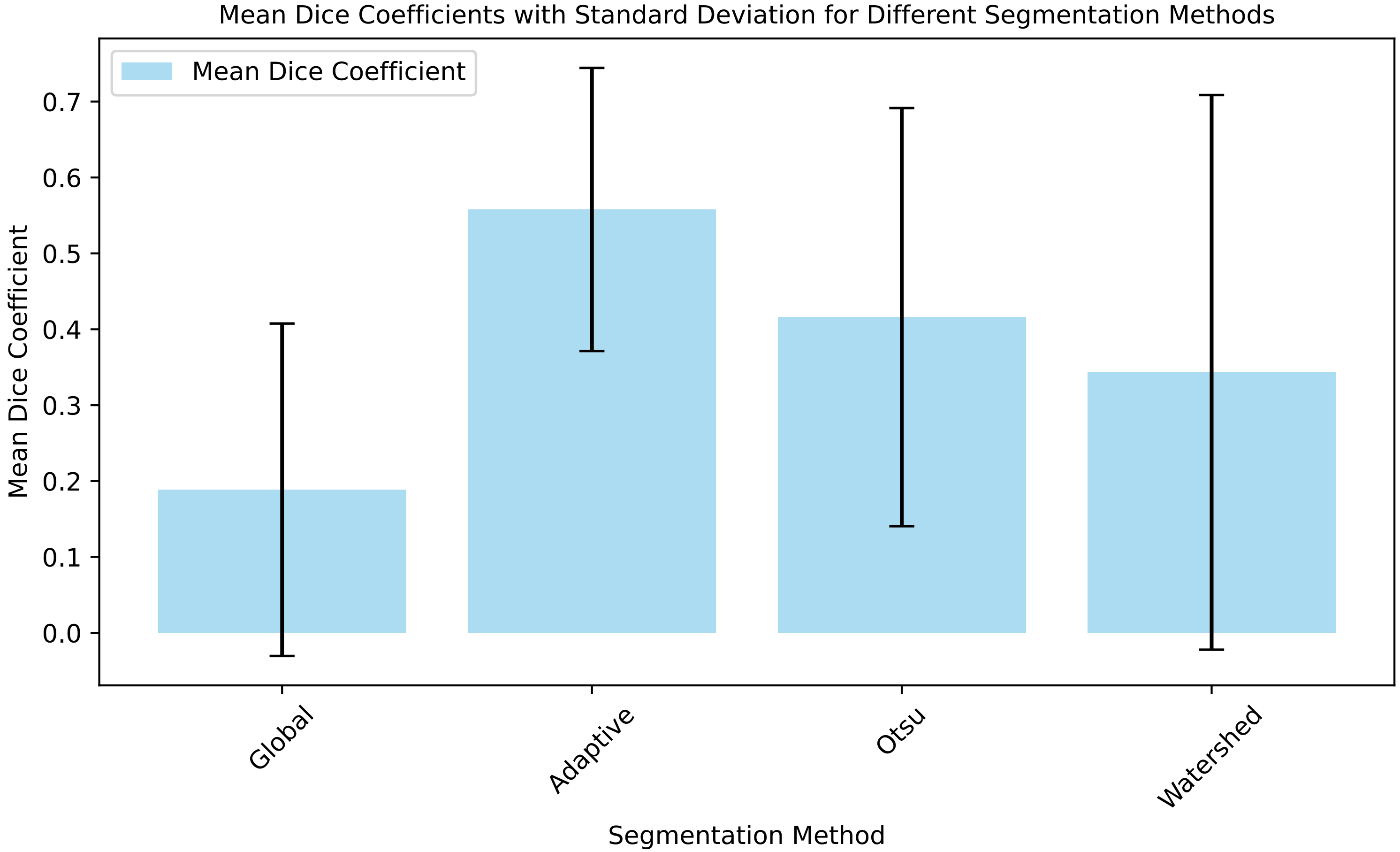
```
# Do the segmentation for the whole dataset now, without visualizing the result
```

```python
# Load the images and masks
# Only 200 works for now
images, masks = load_images_masks(images_path, masks_path, 200)

# Preprocess images and masks
preprocessed_images = [preprocess_image(image, ismask=False) for image in images]
preprocessed_masks = [preprocess_image(mask, ismask=True) for mask in masks]

# # Segment the preprocessed images using different methods
segmented_masks_otsu = [segment(image, method='otsu') for image in preprocessed_images]
segmented_masks_adaptive = [segment(image, method='adaptive') for image in preprocessed_images]
segmented_masks_global = [segment(image, method='global') for image in preprocessed_images]
segmented_masks_watershed = [segment(image, method='watershed') for image in preprocessed_images]
```

In [17]:

```python
# Define the function to compute the Dice coefficient
def dice(mask1, mask2):
    """Compute the Dice coefficient. The input masks should be binary."""
    assert mask1.shape == mask2.shape
    assert mask1.dtype == bool and mask2.dtype == bool
    intersection = mask1 & mask2  # Bitwise AND, equivalent to np.logical_and()
    return 2*np.sum(intersection)/(np.sum(mask1) + np.sum(mask2))

# Initialize dictionaries to store the Dice coefficients for each method
dice_scores = {}

# Compute the Dice coefficient for each pair of segmented masks and ground truth masks
for i in range(len(preprocessed_masks)):
    # Binarize the masks
    mask_gt_bin = preprocessed_masks[i] > 0
    mask_global_bin = segmented_masks_global[i] > 0
    mask_adaptive_bin = segmented_masks_adaptive[i] > 0
    mask_otsu_bin = segmented_masks_otsu[i] > 0
    mask_watershed_bin = segmented_masks_watershed[i] > 0

    # Compute the Dice coefficients
    dice_global = dice(mask_gt_bin, mask_global_bin)
    dice_adaptive = dice(mask_gt_bin, mask_adaptive_bin)
    dice_otsu = dice(mask_gt_bin, mask_otsu_bin)
    dice_watershed = dice(mask_gt_bin, mask_watershed_bin)

    # Store the Dice coefficients in the dictionary
    dice_scores[f"Sample {i+1}"] = {
        "Global": dice_global,
        "Adaptive": dice_adaptive,
        "Otsu": dice_otsu,
        "Watershed": dice_watershed
```

```
    }

# Display the Dice coefficients
dice_df = pd.DataFrame.from_dict(dice_scores, orient='index')

# Calculate the mean and standard deviation of the Dice coefficients for each segmentation method
mean_scores = dice_df.mean()
std_dev = dice_df.std()

# Plotting the mean Dice coefficients with error bars for standard deviation
plt.figure(figsize=(8, 5))

# Plotting the mean scores
plt.bar(mean_scores.index, mean_scores, yerr=std_dev, capsize=5, color='skyblue', alpha=0.7, label='Mean Dice Coefficient')

# Adding labels and title
plt.title('Mean Dice Coefficients with Standard Deviation for Different Segmentation Methods')
plt.xlabel('Segmentation Method')
plt.ylabel('Mean Dice Coefficient')
plt.xticks(rotation=45)
plt.legend()

# Showing the plot
plt.tight_layout()
plt.show()

print("Mean dice values: \n", mean_scores)
print("Std. dev. dice values: \n", std_dev)
```




```
Mean dice values:
 Global       0.188537
Adaptive     0.557905
Otsu         0.415973
Watershed    0.343198
dtype: float64
Std. dev. dice values:
 Global       0.219028
Adaptive     0.186522
Otsu         0.275443
Watershed    0.365412
dtype: float64
```

Discussion

Looking at the dataset by eye, it's very clear that on one side, it's quite easy to distinguish between water and non-water. However after looking closer, there are certain challenges such as the changing weather creating clouds around boders or depending on different factors, water simply isn't blue and land is green, brown. Sometimes the water is white because of reflecting clouds or it might be black because it's very deep.

Manually creating the segmentation by hand was quite simple with Fiji and I ended creating very similar masks as the ones already in the dataset.

Programatically creating the segmentation was of course a bit more challenging. I decided to go with simple, adaptive and Otsu's thresholding first. This already produced some promising results. To go a bit deeper I went on to include the watershed segmentation and put the results for a small sample all into one large graphic.

Global Thresholding does a good job when for images with a good contrast, but in other cases procuces bad results. Adaptive Thresholding is generally really good because it detects all the borders, no matter the varying lightnig conditions. Otsu is like global, but better because it's suitable for bimodal images. Watershed works really well with clearly distinguishable borders but fails when the images have lots of noise or small contrasts.

I went on and created the masks for a larger part of the dataset and finally used the dice method to compare the results between the different methods. The adaptive method produced the best results at a mean dice coefficient of 0.59, followed by otsu at 0.43, watershed at 0.37 and global at 0.18. It is clear that otsu and wathershed would have scored better if the dataset didn't include borders for some of the satellite images because these algorithms generally work better.

It is also worth mentioning that the reference masks included in the dataset, which are also used to create the coefficents, are not perfect. This further falsifies the results.

Issues

- The otsu and watershed algorithms couldn't deal well with some images containing black borders around them. I have tried cropping these borders away in the preprocesssing but after investing a lot of time, I didn't manage to create a working function for this issue.
- Some images created some problems because they were very distored, resulting in images with zero height in the preprocessing. Also unfortunately I could not get the evaluation to run with all the images because at some point an image would break the program at the binarizing mask step. I have tried to figure this out (see code snippet at the end) but after investing too much time I decided to proceed with only the first 200 images.

References

- Module Repository by N. Juchler (https://github.com/hirsch-lab/msls-co4-ss24)

- Used for code examples
- ChatGPT (https://chatgpt.com/)
    - Mainly used for generating basic (repetitive) code snippets and comments

In [ ]:
```
# Debugging of issue that with certain masks I couldn't calculate the dice coefficients
for i in range(len(preprocessed_masks)):
    # Binarize the masks
    mask_gt_bin = preprocessed_masks[i] > 0
    mask_otsu_bin = segmented_masks_otsu[i] > 0
    mask_adaptive_bin = segmented_masks_adaptive[i] > 0
    mask_global_bin = segmented_masks_global[i] > 0
    mask_watershed_bin = segmented_masks_watershed[i] > 0

    # print(f"Shape of ground truth mask: {mask_gt_bin.shape}")
    # print(f"Shape of Otsu segmented mask: {mask_otsu_bin.shape}")
    # print(f"Shape of Adaptive segmented mask: {mask_adaptive_bin.shape}")
    # print(f"Shape of Global segmented mask: {mask_global_bin.shape}")
    # print(f"Shape of Watershed segmented mask: {mask_watershed_bin.shape}")

    try:
        # Compute the Dice coefficients
        dice_otsu = dice(mask_gt_bin, mask_otsu_bin)
        dice_adaptive = dice(mask_gt_bin, mask_adaptive_bin)
        dice_global = dice(mask_gt_bin, mask_global_bin)
        dice_watershed = dice(mask_gt_bin, mask_watershed_bin)

        # Store the Dice coefficients in the dictionary
        dice_scores[f"Sample {i+1}"] = {
            "Otsu": dice_otsu,
            "Adaptive": dice_adaptive,
            "Global": dice_global,
            "Watershed": dice_watershed
        }
    except AssertionError as e:
        print(f"Error occurred for sample {i+1}: {e}")
```