```
import pandas as pd
import numpy as np
import matplotlib.pyplot  as plt
import os
import shutil

import geopandas as gpd

import rasterio
from rasterio.plot import reshape_as_image
from rasterio.features import rasterize

from shapely.geometry import Polygon, MultiPolygon
from shapely.ops import unary_union

import cv2
import glob

from tqdm import tqdm
import ast

from pyproj import CRS
```

Intro

To solve the second task "Computer vision. Sentinel-2 image matching", it was decided to use the provided dataset on Kaggle "Deforestation in Ukraine from Sentinel2 data" for matching satellite images.

Mask creation

The task recommendations included a website that explains how to read a satellite image based on geojson data and make a mask with the necessary polygons based on it. This will help to select only needed places on the image for comparison, rather than comparing the entire image. And since these are satellite images, their size is 10980x10980 and they weigh on average more than 100 mb, which requires huge computing power. Let's see how it is done in the example:

```
df = gpd.read_file(r'Sentinel2/Sentinel2/deforestation_labels.geojson')
```

```
df
```

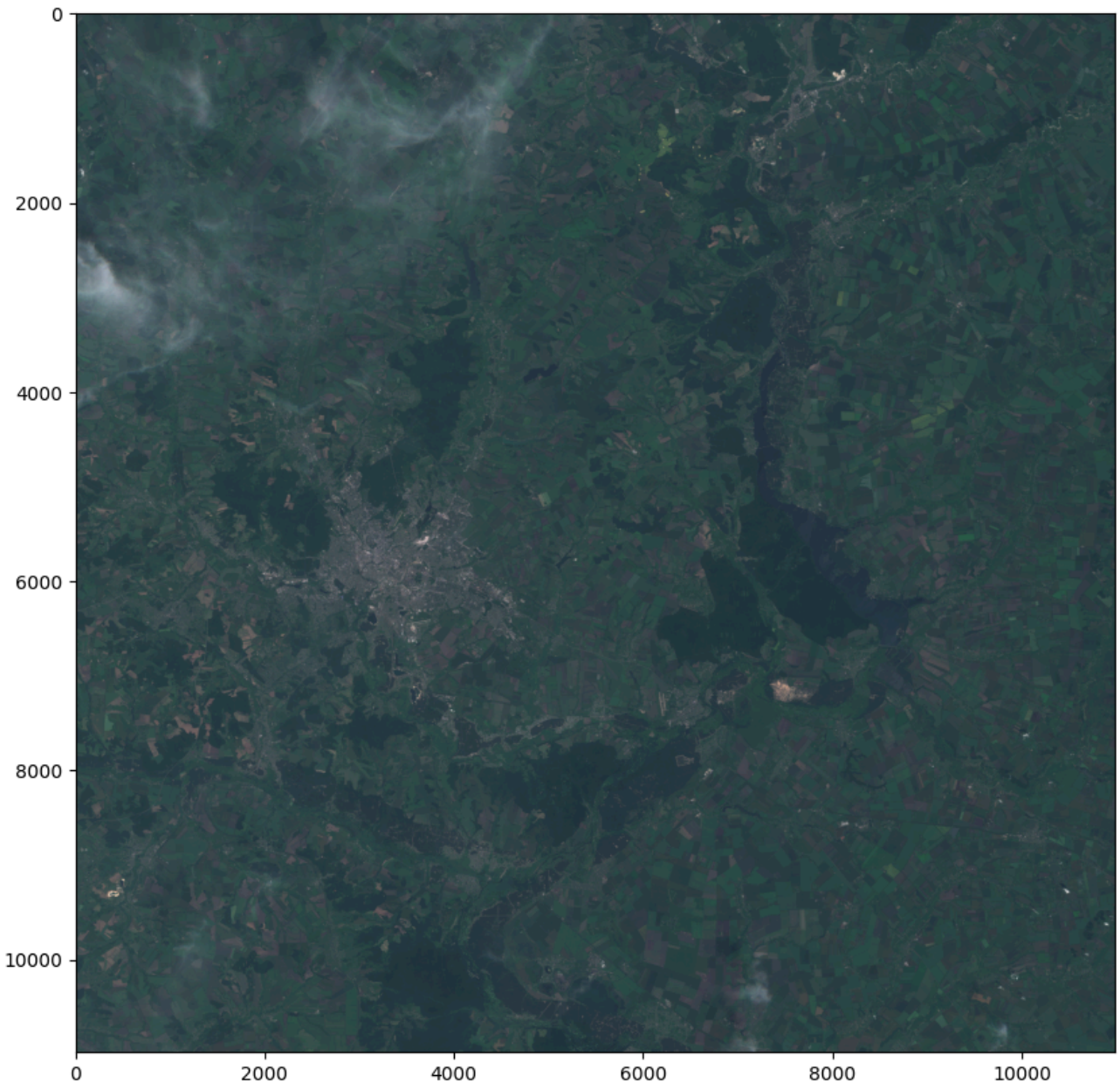
	img_date	tile	geometry
0	2016-04-09	36UXA	POLYGON ((35.7913 50.17406, 35.79277 50.17354,...
1	2016-04-09	36UXA	POLYGON ((35.77653 50.10271, 35.7781 50.10318,...
2	2016-04-09	36UXA	POLYGON ((35.78621 50.01277, 35.78616 50.01369...
3	2016-04-09	36UXA	POLYGON ((35.78007 50.00556, 35.78189 50.00557...
4	2016-04-09	36UXA	POLYGON ((35.79781 49.99568, 35.79908 49.99582...
...	...	...	...
5691	2019-09-09	36UYA	POLYGON ((36.37494 50.17378, 36.37506 50.17376...
5692	2019-09-09	36UYA	POLYGON ((36.37849 50.17354, 36.37851 50.17352...
5693	2019-09-09	36UYA	POLYGON ((35.91477 50.22407, 35.91472 50.22394...
5694	2019-10-26	36UYA	POLYGON ((36.88312 50.26589, 36.88408 50.26489...
5695	2019-10-26	36UYA	POLYGON ((36.88312 50.26589, 36.88408 50.26489...

5696 rows × 3 columns

```
# raster_path = r'Sentinel2/Sentinel2/S2A_MSIL1C_20160212T084052_N0201_R064_T36UYA_201602
raster_path = r'Sentinel2\Sentinel2\S2A_MSIL1C_20160621T084012_N0204_R064_T36UYA_20160621
# raster_path = r'Sentinel2\Sentinel2\S2B_MSIL1C_20180726T084009_N0206_R064_T36UXA_201807

with rasterio.open(raster_path, "r", driver='JP2OpenJPEG') as src:
    raster_image = src.read()
    raster_meta = src.meta
    raster_crs = src.crs


print(raster_image.shape)
print(raster_meta)
print(raster_crs)
```

```
(3, 10980, 10980)
{'driver': 'JP2OpenJPEG', 'dtype': 'uint8', 'nodata': None, 'width': 10980, 'height':
       0.0, -10.0, 5600040.0)}
EPSG:32636
```

```
raster_img = reshape_as_image(raster_image)
plt.figure(figsize=(10, 10))
plt.imshow(raster_img)
plt.show()
```



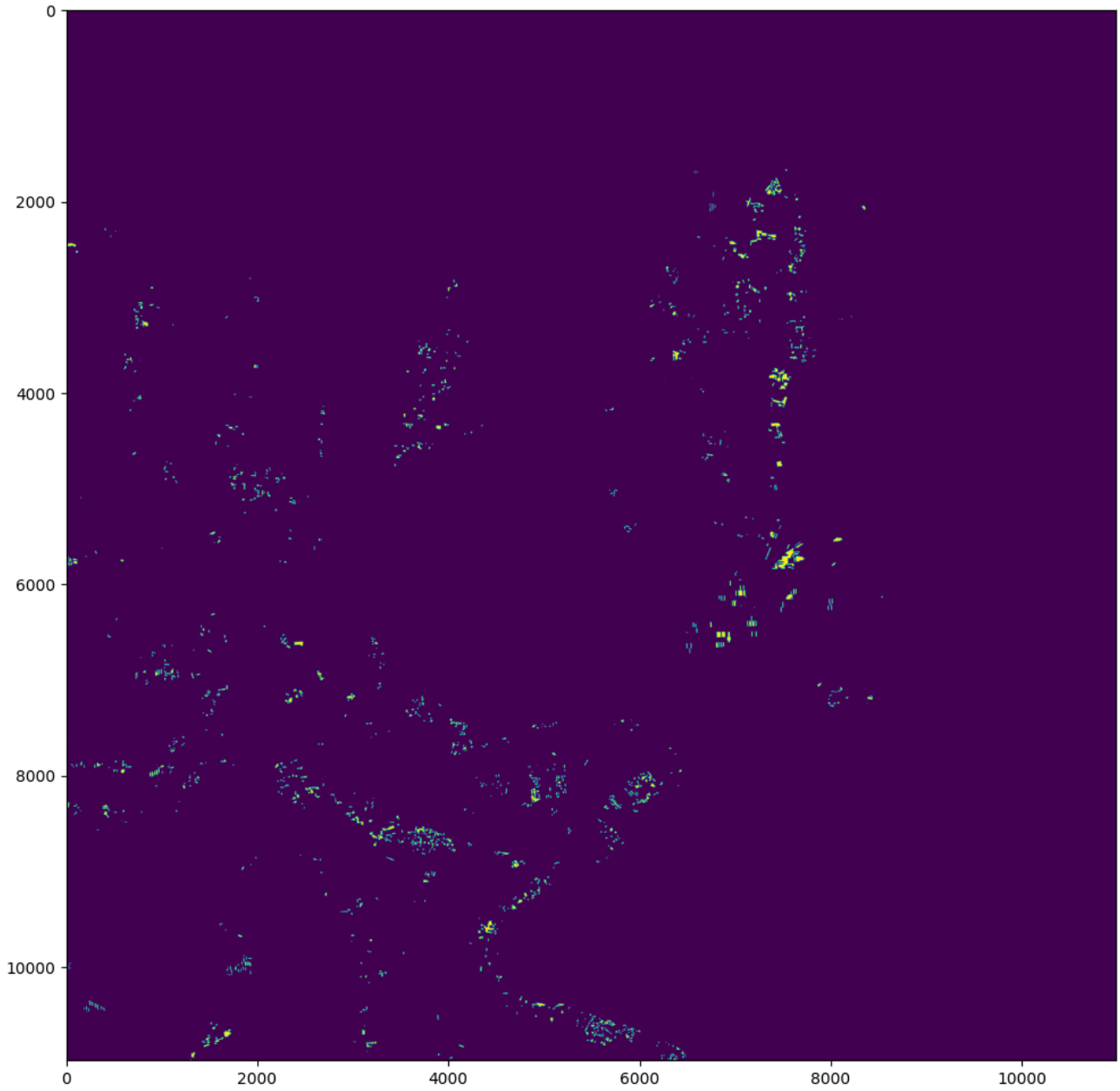

```
# assigning crs
# use 4236 for tiles from this dataset
df.crs = CRS.from_epsg(4236)  # Используем формат CRS.from_epsg
df = df.to_crs(CRS.from_string(raster_meta['crs'].to_string()))  # Современный способ


df
```

	img_date	tile	geometry
0	2016-04-09	36UXA	POLYGON ((699218.537 5562133.702, 699325.839 5...
1	2016-04-09	36UXA	POLYGON ((698459.426 5554162.662, 698569.358 5...
2	2016-04-09	36UXA	POLYGON ((699524.788 5544189.095, 699516.903 5...
3	2016-04-09	36UXA	POLYGON ((699114.519 5543371.364, 699244.558 5...
4	2016-04-09	36UXA	POLYGON ((700426.849 5542320.373, 700517.265 5...
...	...	...	...
5691	2019-09-09	36UYA	POLYGON ((740891.051 5563826.491, 740899.346 5...
5692	2019-09-09	36UYA	POLYGON ((741145.387 5563811.991, 741147.24 55...
5693	2019-09-09	36UYA	POLYGON ((707816.789 5568031.074, 707813.722 5...
5694	2019-10-26	36UYA	POLYGON ((776636.52 5575832.368, 776710.566 55...
5695	2019-10-26	36UYA	POLYGON ((776636.52 5575832.368, 776710.566 55...

5696 rows × 3 columns

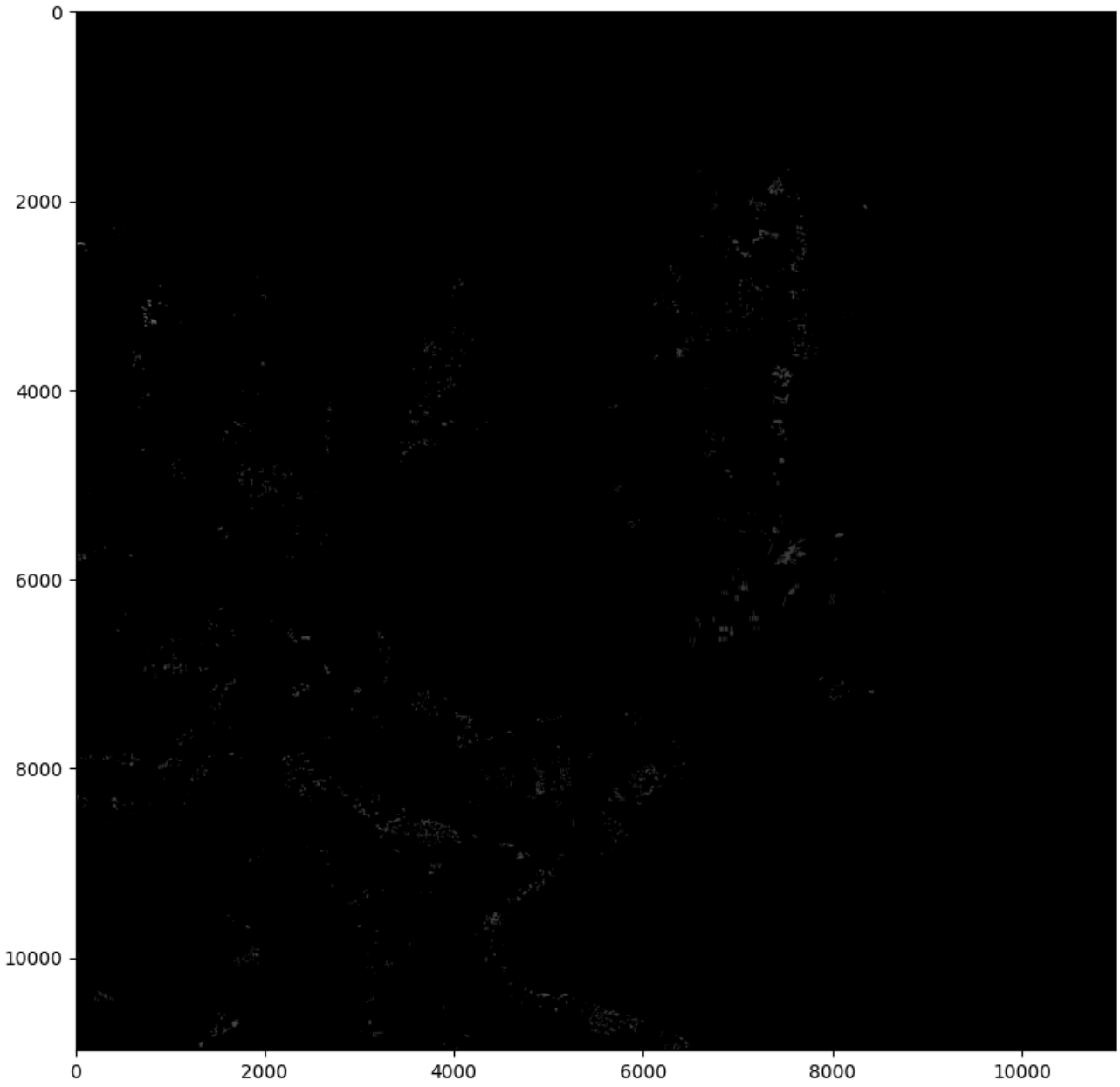
```
# rasterize works with polygons that are in image coordinate system
def poly_from_utm(polygon, transform):
    # make a polygon from multipolygon
    poly = unary_union(polygon)
    # transfrom each polygon to image crs, using raster meta
    poly_pts = [~transform * tuple(i[0:2]) for i in np.array(poly.exterior.coords)]
    return Polygon(poly_pts)

# creating binary mask
poly_shp = []
im_size = (src.meta['height'], src.meta['width'])

for num, row in df.iterrows():
    if row['geometry'].geom_type == 'Polygon':
        poly_shp.append(poly_from_utm(row['geometry'], src.meta['transform']))
    else:
        for p in row['geometry'].geoms:
            poly_shp.append(poly_from_utm(p, src.meta['transform']))


mask = rasterize(shapes=poly_shp, out_shape=im_size)
plt.figure(figsize=(12,12))
plt.imshow(mask)
```

<matplotlib.image.AxesImage at 0x18c05ac5ad0>



```
# Converting the 3-channel image to grayscale without changing the mask
raster_img_gray = cv2.cvtColor(raster_img, cv2.COLOR_RGB2GRAY)
print(f"Converted raster image shape (grayscale): {raster_img_gray.shape}")

# Ensure the mask is of type uint8 and scaled to 0-255
if mask.dtype != np.uint8 or mask.max() == 1:
    print(f"Converting mask to uint8 and scaling it to 0-255")
    mask_cv = (mask * 255).astype(np.uint8)

# Check if the mask has the same size as the grayscale image
if raster_img_gray.shape != mask.shape:
    print(f"Resizing mask from {mask.shape} to {raster_img_gray.shape}")
    mask_cv = cv2.resize(mask_cv, (raster_img_gray.shape[1], raster_img_gray.shape[0]))


# Apply the mask to the grayscale image
def apply_mask(image, mask):
    masked_image = cv2.bitwise_and(image, image, mask=mask)
    return masked_image


# Visualize the result
plt.figure(figsize=(10, 10))
plt.imshow(apply_mask(raster_img_gray, mask_cv), cmap='gray')
plt.show()
```

```
Converted raster image shape (grayscale): (10980, 10980)
Converting mask to uint8 and scaling it to 0-255
```



Got a picture with the mask overlaid in gray colors. In gray colors precisely because it will allow to highlight the boundaries of color changes more clearly, as well as to avoid the seasonal part, when forests are green or orange, thus confusing the algorithm. Also gray image reduces the weight of the image, as it contains only one channel instead of 3 and simplifies the calculations, thus reducing the computational load.

Algorithm building

Image extraction

First of all it is necessary to get the images from the provided dataset into a new folder, where only the necessary images will be stored. We go through all files in the source directory. The required images will always have the name “_TCI.jp2” at the end. That's how we do the search. And then copy to the newly created folder. We could do this already in the process of mask creation, but since these images will be needed again, it makes sense to search and delete unnecessary mb right away

```
def extract_tci_images(source_dir, target_dir, max_images=None):
    # Create a folder for the extracted images if it does not already exist
    if not os.path.exists(target_dir):
        os.makedirs(target_dir)

    # Counter to limit the number of files
    count = 0

    # Go through all the files in the source directory
    for root, dirs, files in os.walk(source_dir):
        for file in files:
            if file.endswith("_TCI.jp2"):  # Check that the file ends with “_TCI.jp2”.
                full_path = os.path.join(root, file)
                shutil.copy(full_path, target_dir)  # Copy the file to a new folder
                count += 1

                if max_images and count >= max_images:
                    print(f"Extracted {count} of images, stop.")
                    return

    print(f"A total of {count} images have been extracted.")

# getting pictures
source_directory = 'Sentinel2\Sentinel2'
target_directory = 'Sentinel2 IMG'
extract_tci_images(source_directory, target_directory)
```

Image Processing and Masking

Now, based on the provided guide for creating a mask, let's make functions that will do this automatically. The only exception is the function for resizing pictures. Although they will be grayed out, but later they will have to be compared 1225 times, which will be a heavy load for the computer. That's why it was decided to cut the size of pictures by half. All images will be converted from JP2 to JPG because JPEG is an easier format to work with and takes up less disk space. In addition, the JPG format is widely used and supported by many programs and platforms, while JP2 cannot be opened by a basic toolkit.

```
# Loading an image and resizing
def load_and_preprocess_image(image_path, max_dimension=5490):
    img = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)  # for reading in gray
    height, width = img.shape
    scale = max_dimension / max(height, width)
    resized_img = cv2.resize(img, (int(width * scale), int(height * scale)))
    return resized_img


# Convert polygons to image coordinates
def poly_from_utm(polygon, transform):
    poly = unary_union(polygon)
    poly_pts = [~transform * tuple(coord[:2]) for coord in np.array(poly.exterior.coords)
    return Polygon(poly_pts)


# Create a mask for an image based on geodata
def create_mask_from_geodata(image_path, geodata_path):
    # Open the image with rasterio
    with rasterio.open(image_path) as src:
        transform = src.meta['transform']
        img_size = (src.meta['height'], src.meta['width'])

    # Reading geodata (.geojson file)
    df = gpd.read_file(geodata_path)

    # Set the coordinate system
    df.crs = CRS.from_epsg(4236)  # Use the CRS.from_epsg format
    df = df.to_crs(CRS.from_string(src.crs.to_string()))  # Transforming polygons to the

    # Convert polygons to image coordinates
    poly_shp = []
    for _, row in df.iterrows():
        geom = row['geometry']
        if geom.geom_type == 'Polygon':
            poly_shp.append(poly_from_utm(geom, transform))
        elif geom.geom_type == 'MultiPolygon':
            for p in geom.geoms:
                poly_shp.append(poly_from_utm(p, transform))

    # Creating a mask using the rasterize function
    mask = rasterize(shapes=poly_shp, out_shape=img_size)

    # Convert the mask to uint8 format
    mask = (mask * 255).astype(np.uint8)

    return mask


# Creating a mask for pictures
def preprocess_and_save_images_with_masks(source_images_path, dest_dataset_path, geodata_
    if not os.path.exists(dest_dataset_path):
        os.makedirs(dest_dataset_path)

    # Getting all TCI images from a specified folder
```

```
    tci_image_files = glob.glob(os.path.join(source_images_path, "*_TCI.jp2"))

    for image_file in tci_image_files:
        # Getting the file name
        base_name = os.path.basename(image_file)

        # Constructing a path to save a preprocessed image
        dest_image_path = os.path.join(dest_dataset_path, base_name.replace('.jp2', '.jpg

        # Creating an image mask based on geodata
        mask = create_mask_from_geodata(image_file, geodata_path)

        # Image loading and preprocessing
        preprocessed_image = load_and_preprocess_image(image_file, max_dimension)

        # Resize the mask to fit the size of the preprocessed image
        mask_resized = cv2.resize(mask, (preprocessed_image.shape[1], preprocessed_image.

        # Applying a mask to a preprocessed image
        masked_image = cv2.bitwise_and(preprocessed_image, preprocessed_image, mask=mask_

        # Saving masked image
        cv2.imwrite(dest_image_path, masked_image, [int(cv2.IMWRITE_JPEG_QUALITY), 90])


# Start functions
source_images_path = 'Sentinel2 IMG'
dest_dataset_path = 'Sentinel2 IMG Mask 5k'
geodata_path = 'Sentinel2/Sentinel2/deforestation_labels.geojson'

preprocess_and_save_images_with_masks(source_images_path, dest_dataset_path, geodata_path
```

Searching for keypoints

There are many different methods to find keypoints (SIFT, SURF, KAZE, AKAZE, FAST, BRIEF, ORB, ... etc.). Unfortunately not all methods are available for public use. Therefore, it was decided to use the AKAZE method to solve the problem. AKAZE applies methods based on fuzzy Gaussian blurring to find key points. This process involves creating multiple images at different scales and finding points that have a significant difference compared to the surrounding pixels. The method itself is based on KAZE and is considered relatively new. To use the AKAZE method, we will use the OpenCV library. After finding the keypoints and descriptors, we write everything to a dataframe along with the path to the image to save memory space.

```
# Function to process images and find keypoints using AKAZE
def process_images(images_folder, output_csv_path):

    # List to store image paths, keypoints, and descriptors
    data = []

    # Find all images with the "_TCI" suffix
    image_paths = glob.glob(os.path.join(images_folder, '*_TCI.*'))

    # Initialize AKAZE
    akaze = cv2.AKAZE_create()

    for image_path in image_paths:
        print(f'Processing {image_path}')

        # Open the image
        image = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)  # grayscale

        # Detect keypoints and descriptors using AKAZE
        keypoints, descriptors = akaze.detectAndCompute(image, None)

        # Save keypoints and descriptors if they are found
        if keypoints is not None and descriptors is not None:
            # Convert keypoints to a simpler format for storage
            kp_list = [(kp.pt, kp.size, kp.angle, kp.response, kp.octave, kp.class_id) fo

            # Add the data to the list (image path, keypoints, descriptors)
            data.append({
                "image_path": image_path,
                "keypoints": kp_list,  # List of keypoint coordinates
                "descriptors": descriptors.tolist()  # Convert descriptors to list
            })

    # Create a DataFrame from the collected data
    df = pd.DataFrame(data)

    # Save the DataFrame to a CSV file for easy inspection
    df.to_csv(output_csv_path, index=False)
    print(f'Keypoints and descriptors saved to {output_csv_path}')

# Example function call
images_folder = 'Sentinel2 IMG Mask 5k'
geodata_path = 'Sentinel2/Sentinel2/deforestation_labels.geojson'
output_csv_path = 'keypoints_data_mask.csv'

process_images(images_folder, output_csv_path)
```

```
Processing Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UXA_20180731T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UXA_20180805T083559_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UXA_20180810T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UXA_20180815T084009_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UXA_20180820T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UXA_20180825T083549_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UXA_20180830T083601_TCI.jpg
```

```
Processing Sentinel2 IMG Mask 5k\T36UXA_20180904T083549_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UXA_20180919T083621_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UXA_20190427T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UXA_20190606T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20160212T084052_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20160330T082542_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20160405T085012_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20160502T083602_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20160509T082612_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20160618T082602_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20160621T084012_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20160830T083602_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20161026T083032_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20161121T085252_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20161205T083332_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190313T083739_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190318T083701_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190328T084011_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190402T083559_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190407T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190412T083609_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190417T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190422T083609_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190427T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190517T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190601T083609_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190606T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190611T083609_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190616T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190621T083609_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190626T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190701T083609_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190706T083611_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190721T083609_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190726T083611_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190805T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190810T083609_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190815T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190825T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190830T083609_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190904T083601_TCI.jpg
Processing Sentinel2 IMG Mask 5k\T36UYA_20190909T083559_TCI.jpg
Keypoints and descriptors saved to keypoints_data_mask.csv
```

```
df_k = pd.read_csv('keypoints_data_mask.csv')
```

```
df_k
```

	image_path	keypoints	descriptors
0	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	[((2153.186279296875, 35.134918212890625), 4.8...	[[72, 237, 3, 64, 16, 242, 135, 100, 150, 180,...
1	Sentinel2 IMG Mask 5k\T36UXA_20180731T083601_T...	[((2153.9580078125, 35.295082092285156), 4.800...	[[97, 253, 73, 38, 1, 255, 252, 54, 203, 250, ...
2	Sentinel2 IMG Mask 5k\T36UXA_20180805T083559_T...	[((2504.82421875, 92.6819076538086), 4.8000001...	[[64, 237, 3, 108, 1, 242, 229, 44, 146, 244, ...
3	Sentinel2 IMG Mask 5k\T36UXA_20180810T083601_T...	[((2504.86376953125, 92.72518157958984), 4.800...	[[72, 237, 139, 76, 1, 242, 229, 44, 146, 244,...
4	Sentinel2 IMG Mask 5k\T36UXA_20180815T084009_T...	[((2153.229736328125, 35.38895034790039), 4.80...	[[72, 237, 3, 64, 18, 242, 135, 100, 150, 244,...
5	Sentinel2 IMG Mask 5k\T36UXA_20180820T083601_T...	[((2504.85595703125, 92.71087646484375), 4.800...	[[72, 237, 139, 108, 1, 242, 229, 44, 146, 244...
6	Sentinel2 IMG Mask 5k\T36UXA_20180825T083549_T...	[((2504.841796875, 92.72445678710938), 4.80000...	[[72, 237, 139, 108, 1, 242, 229, 44, 146, 244...
7	Sentinel2 IMG Mask 5k\T36UXA_20180830T083601_T...	[((2153.181640625, 35.21105194091797), 4.80000...	[[72, 237, 3, 64, 18, 242, 135, 100, 150, 180,...
8	Sentinel2 IMG Mask 5k\T36UXA_20180904T083549_T...	[((2153.185302734375, 35.24109649658203), 4.80...	[[72, 237, 3, 64, 16, 242, 135, 100, 150, 180,...
9	Sentinel2 IMG Mask 5k\T36UXA_20180919T083621_T...	[((2153.172119140625, 35.198665618896484), 4.8...	[[64, 237, 3, 64, 16, 242, 135, 100, 150, 180,...
10	Sentinel2 IMG Mask 5k\T36UXA_20190427T083601_T...	[((2808.027099609375, 80.68900299072266), 4.80...	[[96, 237, 3, 4, 0, 255, 252, 116, 154, 214, 1...
11	Sentinel2 IMG Mask 5k\T36UXA_20190606T083601_T...	[((2504.831298828125, 92.7143783569336), 4.800...	[[72, 237, 3, 76, 1, 242, 229, 44, 146, 244, 1...
12	Sentinel2 IMG Mask 5k\T36UYA_20160212T084052_T...	[((3765.326904296875, 837.7035522460938), 4.80...	[[72, 237, 139, 206, 17, 254, 231, 60, 146, 24...
13	Sentinel2 IMG Mask 5k\T36UYA_20160330T082542_T...	[((3765.386474609375, 837.7262573242188), 4.80...	[[97, 253, 1, 68, 1, 255, 167, 102, 211, 252, ...
14	Sentinel2 IMG Mask 5k\T36UYA_20160405T085012_T...	[((203.08712768554688, 1148.499755859375), 4.8...	[[72, 237, 3, 72, 18, 242, 167, 108, 150, 180,...
		[((3306.0068359375	[[64, 13, 11, 4, 0

15	Sentinel2 IMG Mask 5k\T36UYA_20160502T083602_T...	[((3765.[illegible], 847.4597778320312), 4.8000...	[[64, 16, 11, 4, 0, 176, 240, 116, 154, 222, 2...
16	Sentinel2 IMG Mask 5k\T36UYA_20160509T082612_T...	[((3765.163330078125, 837.5535278320312), 4.80...	[[96, 253, 1, 68, 1, 255, 183, 118, 219, 250, ...
17	Sentinel2 IMG Mask 5k\T36UYA_20160618T082602_T...	[((3765.244384765625, 837.67724609375), 4.8000...	[[72, 237, 139, 207, 81, 254, 231, 124, 146, 2...
18	Sentinel2 IMG Mask 5k\T36UYA_20160621T084012_T...	[((3765.195556640625, 837.5310668945312), 4.80...	[[72, 237, 139, 207, 17, 254, 231, 60, 130, 24...
19	Sentinel2 IMG Mask 5k\T36UYA_20160830T083602_T...	[((3765.205322265625, 837.5341796875), 4.80000...	[[72, 237, 139, 206, 17, 254, 231, 60, 130, 24...
20	Sentinel2 IMG Mask 5k\T36UYA_20161026T083032_T...	[((3765.222412109375, 837.2945556640625), 4.80...	[[72, 237, 139, 199, 1, 254, 247, 60, 130, 240...
21	Sentinel2 IMG Mask 5k\T36UYA_20161121T085252_T...	[((200.6117401123047, 1141.7449951171875), 4.8...	[[97, 253, 1, 68, 1, 255, 245, 54, 203, 254, 1...
22	Sentinel2 IMG Mask 5k\T36UYA_20161205T083332_T...	[((3765.326904296875, 837.7035522460938), 4.80...	[[72, 237, 139, 206, 17, 254, 231, 60, 146, 24...
23	Sentinel2 IMG Mask 5k\T36UYA_20190313T083739_T...	[((3765.072998046875, 837.54052734375), 4.8000...	[[96, 253, 1, 68, 1, 255, 183, 118, 219, 250, ...

Comparison of keypoints

We will need descriptors to compare key points. Since they were saved in string format, we convert them back to numpy array. In the comparison function itself, we will create a small bar to view the comparison progress using the tqdm library.

There are 1225 images to be compared in total. Thanks to the OpenCV library, comparisons can be done using built-in functions. And to filter out, so to say, “bad” points, let's set the threshold at 80%. Filter all matches to leave only those that have a distance less than a certain threshold value, calculated on the basis of the maximum distance among all found matches multiplied by the threshold.

For comparative analysis, we record the similarity in the images based on the ratio of all found similar points to the total number of points. And at the end create a new dataframe to store the paths to the matched images and their similarity percentage.

```
# Function to convert descriptors from string format to numpy array
def convert_descriptors(descriptor_string):
    try:
        # Convert the string back into a list of numbers
        descriptors = np.array(ast.literal_eval(descriptor_string), dtype=np.uint8)  # Se
        return descriptors
    except:
        return None  # Return None if no descriptors are available

# Function to compare images using keypoints and descriptors
def compare_images(df, threshold=0.8, max_matches=None):

    # List to store matching image pairs and similarity percentage
    matches = []

    # Progress bar
    total_comparisons = (len(df) * (len(df) - 1)) // 2  # C(n, 2) = n * (n - 1) / 2
    progress_bar = tqdm(total=total_comparisons, desc="Comparing images", unit="compariso

    # Brute-Force matcher for feature matching
    # bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=False)
    # bf = cv2.BFMatcher()
    matcher = cv2.DescriptorMatcher_create(cv2.DescriptorMatcher_BRUTEFORCE_HAMMING)

    # Loop through the DataFrame to compare each image with every other image
    for i in range(len(df)):
        for j in range(i + 1, len(df)):
            # Check if the limit of matches has been reached
            if max_matches is not None and len(matches) >= max_matches:
                print(f"Limit of {max_matches} matches reached. Stopping process.")
                progress_bar.close()
                return pd.DataFrame(matches, columns=["image1", "image2", "similarity (%)

            # Get descriptors of the two images to compare
            descriptors1 = convert_descriptors(df.loc[i, "descriptors"])
            descriptors2 = convert_descriptors(df.loc[j, "descriptors"])

            # Skip if no descriptors are found for either image
            if descriptors1 is None or descriptors2 is None:
                continue

            # Perform feature matching
            matches_found = matcher.knnMatch(descriptors1, descriptors2, 2)
            # matches_found = bf.knnMatch(descriptors1, descriptors2, k=2)

            # Sort matches based on distance (lower distance means better match)
            # matches_found = sorted(matches_found, key=lambda x: x.distance)

            # Calculate the percentage of good matches (ratio of matches below a threshol
            good_matches = [m for m, n in matches_found if m.distance < threshold * n.dis
            # good_matches = [m for m, n in matches_found if m.distance < threshold * n.d
            match_ratio = len(good_matches) / max(len(matches_found), 1)  # Avoid divisio
            match_percentage = match_ratio * 100  # Convert ratio to percentage

            # If match ratio is above the threshold, add the match pair to the list along
```

```
            # if len(good_matches) >= 80:
            matches.append([df.loc[i, "image_path"], df.loc[j, "image_path"], round(match

            # Update progress bar
            progress_bar.update(1)

    progress_bar.close()

    # Convert the list of matches to a DataFrame for further analysis or saving
    matches_df = pd.DataFrame(matches, columns=["image1", "image2", "similarity (%)"])

    print(f'Found {len(matches)} matching image pairs.')
    return matches_df

# Example usage
df = pd.read_csv('keypoints_data_mask.csv')

# Compare images in the DataFrame with a limit of 10 matches
matches_df = compare_images(df, threshold=0.8)

# Save the matching pairs to a CSV for later visualization
matches_df.to_csv('image_matches_mask.csv', index=False)
print("Image matches saved to image_matches.csv")
```

```
Comparing images: 100%|██████████| 1225/1225 [25:08<00:00,  1.23s/comparison]Found 12
Image matches saved to image_matches.csv
```

```
df_match = pd.read_csv('image_matches_mask.csv')


df_match
```

	image1	image2	similarity (%)
0	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UXA_20180731T083601_T...	60.13
1	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UXA_20180805T083559_T...	57.82
2	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UXA_20180810T083601_T...	53.21
3	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UXA_20180815T084009_T...	66.76
4	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UXA_20180820T083601_T...	59.78
...	...	...	...
1220	Sentinel2 IMG Mask 5k\T36UYA_20190825T083601_T...	Sentinel2 IMG Mask 5k\T36UYA_20190904T083601_T...	86.61
1221	Sentinel2 IMG Mask 5k\T36UYA_20190825T083601_T...	Sentinel2 IMG Mask 5k\T36UYA_20190909T083559_T...	85.63

Match visualization

Moving on to visualization, there are a couple of additional functions that need to be made. The first one is a function that would pull key points from the dataframe and convert them to cv2.KeyPoint. The second is a similar function for resizing the image, but here all colors are used, as it is preferable for visualizing results than looking at matching points by gray masks on a black background.

```
def convert_to_keypoints(kp_string):
    kp_list = ast.literal_eval(kp_string)

    keypoints = []
    for (pt, size, angle, response, octave, class_id) in kp_list:
        # Create an object cv2.KeyPoint
        keypoint = cv2.KeyPoint(x=pt[0], y=pt[1], size=size, angle=angle, octave=octave,
        keypoints.append(keypoint)
    return keypoints
```

```
def load_and_preprocess_image_color(image_path, max_dimension=5490):
    img = cv2.imread(image_path, cv2.IMREAD_COLOR)
    height, width, _ = img.shape
    scale = max_dimension / max(height, width)
    resized_img = cv2.resize(img, (int(width * scale), int(height * scale)))
    return resized_img
```

The visualization function itself pulls the necessary index information from pre-prepared dataframes of keypoints with descriptors and matches. For original images, the title is converted back to jp2 format, the desired image is pulled and resized. After all procedures the graphs are plotted.

```
# Function to visualize matches between two images using keypoints from the DataFrame
def visualize_matches(df_keypoints, df_matches, match_index=0):

    # Extract the paths of the matching images from the df_matches DataFrame
    image1_path = df_matches.loc[match_index, 'image1']
    image2_path = df_matches.loc[match_index, 'image2']

    # Retrieve the keypoints and descriptors for the two images from df_keypoints
    kp1 = convert_to_keypoints(df_keypoints[df_keypoints['image_path'] == image1_path]['k
    kp2 = convert_to_keypoints(df_keypoints[df_keypoints['image_path'] == image2_path]['k

    des1 = convert_descriptors(df_keypoints[df_keypoints['image_path'] == image1_path]['d
    des2 = convert_descriptors(df_keypoints[df_keypoints['image_path'] == image2_path]['d

    # Load the images
    img1 = cv2.imread(image1_path)
    img2 = cv2.imread(image2_path)

    # Load original images (format .jp2)
    original_img1_path = os.path.join('Sentinel2 IMG', os.path.basename(image1_path).repl
    original_img2_path = os.path.join('Sentinel2 IMG', os.path.basename(image2_path).repl

    original_img1 = load_and_preprocess_image_color(original_img1_path)
    original_img2 = load_and_preprocess_image_color(original_img2_path)

    # Match descriptors using BFMatcher
    bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)
    matches = bf.match(des1, des2)

    # Sort matches based on their distance (best matches first)
    matches = sorted(matches, key=lambda x: x.distance)

    # Plot images and matches

    # First row: Original images
    plt.figure(figsize=(25, 20))

    plt.subplot(2, 2, 1)
    plt.imshow(cv2.cvtColor(original_img1, cv2.COLOR_BGR2RGB))
    plt.title('Original Image 1')

    plt.subplot(2, 2, 2)
    plt.imshow(cv2.cvtColor(original_img2, cv2.COLOR_BGR2RGB))
    plt.title('Original Image 2')

    plt.tight_layout()
    plt.savefig('check.jpg')


    # Second row: Processed (gray) images
    plt.figure(figsize=(25, 20))

    plt.subplot(2, 2, 1)
    plt.imshow(cv2.cvtColor(img1, cv2.COLOR_BGR2RGB))
    plt.title('Processed Image 1')
```

```
    plt.subplot(2, 2, 2)
    plt.imshow(cv2.cvtColor(img2, cv2.COLOR_BGR2RGB))
    plt.title('Processed Image 2')

    plt.tight_layout()


    # Third row: Matches between images
    plt.figure(figsize=(20, 10))

    # Display matches between images
    match_img = cv2.drawMatches(original_img1, kp1, original_img2, kp2, matches[:20], Non

    plt.imshow(cv2.cvtColor(match_img, cv2.COLOR_BGR2RGB))
    plt.title('Keypoint Matches on Original Images')

    plt.tight_layout()

    plt.show()

    print(df_matches.loc[match_index, 'similarity (%)'])
```

First, let's display a dataframe of results where the matches are greater than 50%.

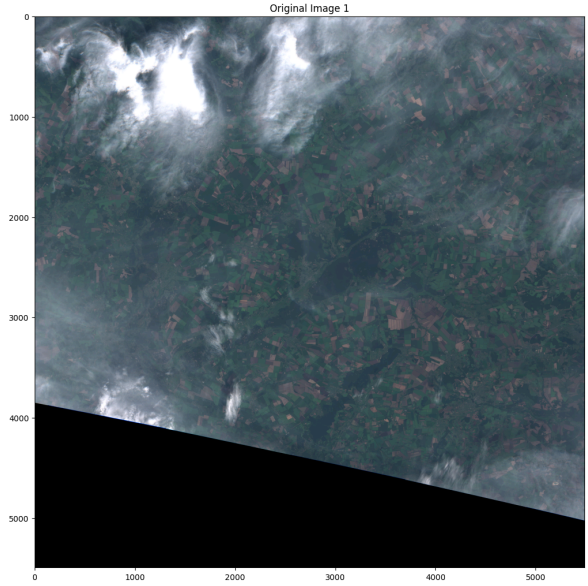
```
df_match[df_match['similarity (%)'] >= 50]
```

	image1	image2	similarity (%)
0	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UXA_20180731T083601_T...	60.13
1	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UXA_20180805T083559_T...	57.82
2	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UXA_20180810T083601_T...	53.21
3	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UXA_20180815T084009_T...	66.76
4	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UXA_20180820T083601_T...	59.78
...	...	...	...
1220	Sentinel2 IMG Mask 5k\T36UYA_20190825T083601_T...	Sentinel2 IMG Mask 5k\T36UYA_20190904T083601_T...	86.61
1221	Sentinel2 IMG Mask 5k\T36UYA_20190825T083601_T...	Sentinel2 IMG Mask 5k\T36UYA_20190909T083559_T...	85.63

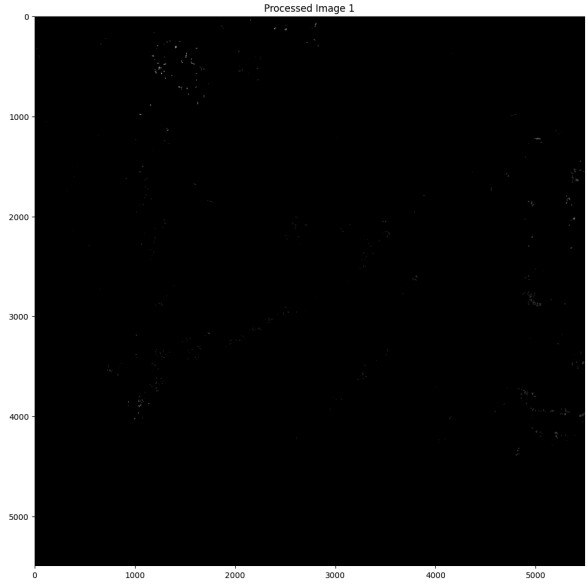
By briefly analyzing the result we can surely say that there are no matches between two different places (UXA does not match UYA). It is great that the algorithm does not find any matches in them.

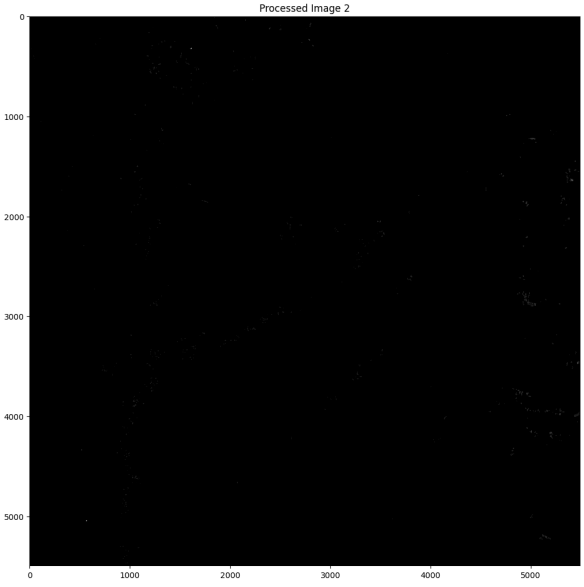
Let's run the visualization function on a few examples from the obtained list

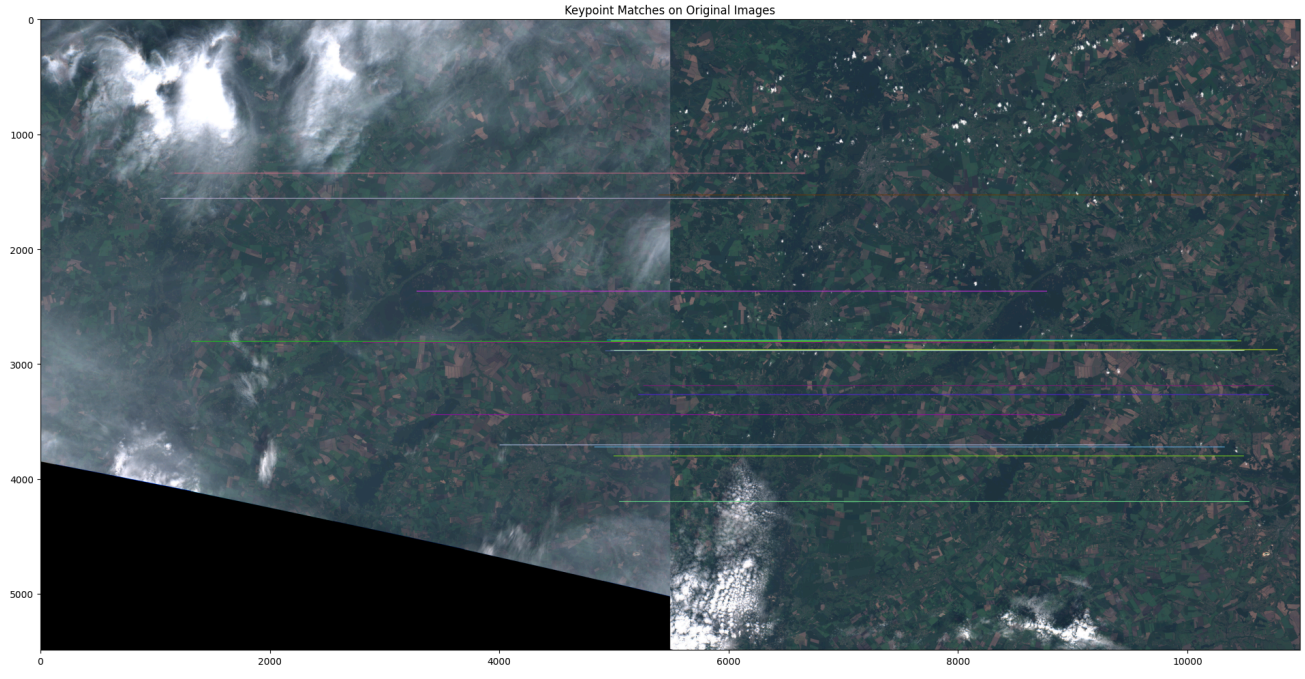
```
# Visualization of matched results:
visualize_matches(df_k, df_match, match_index=0)
```


Original Image 1
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
5000


Original Image 2
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
5000

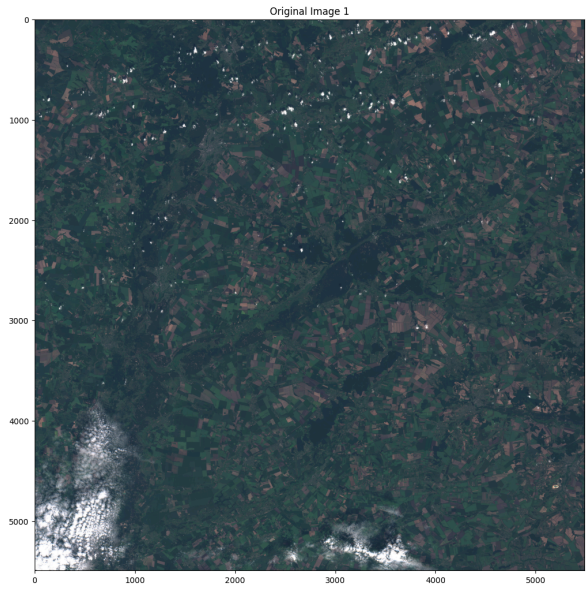

Processed Image 1
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
5000

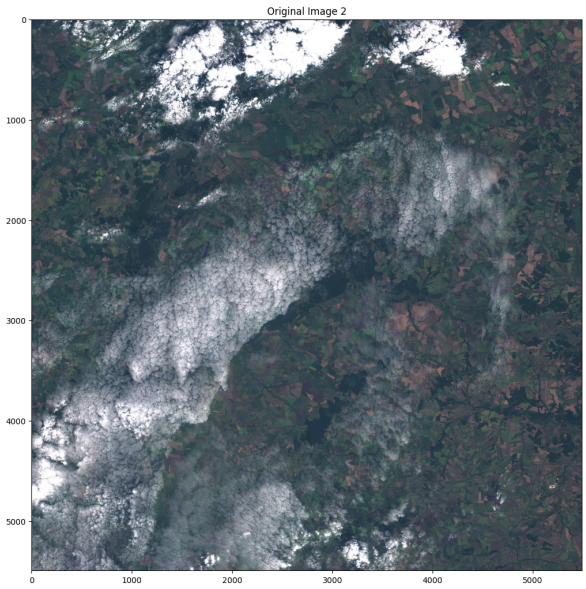

Processed Image 2
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
5000

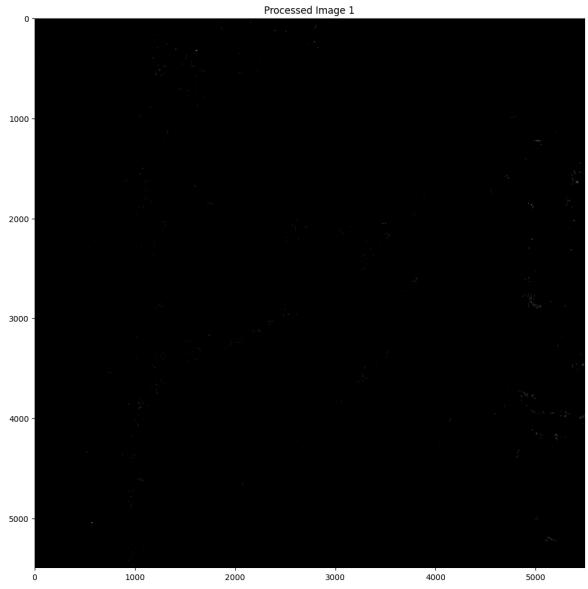

Keypoint Matches on Original Images
0
1000
2000
3000
4000
5000
0
2000
4000
6000
8000
10000

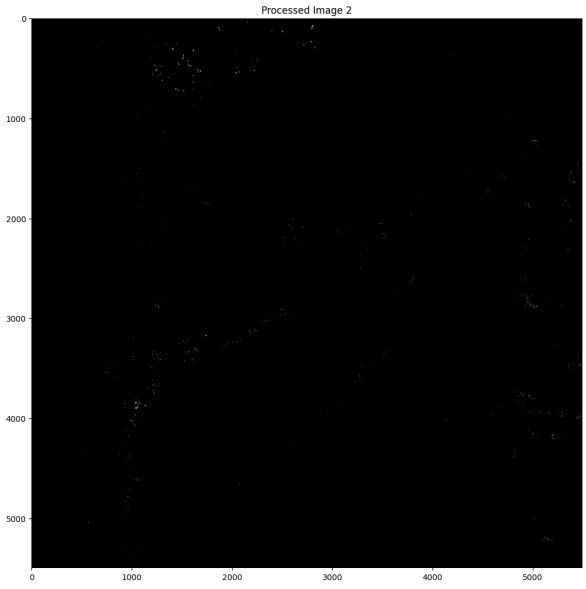
60.13

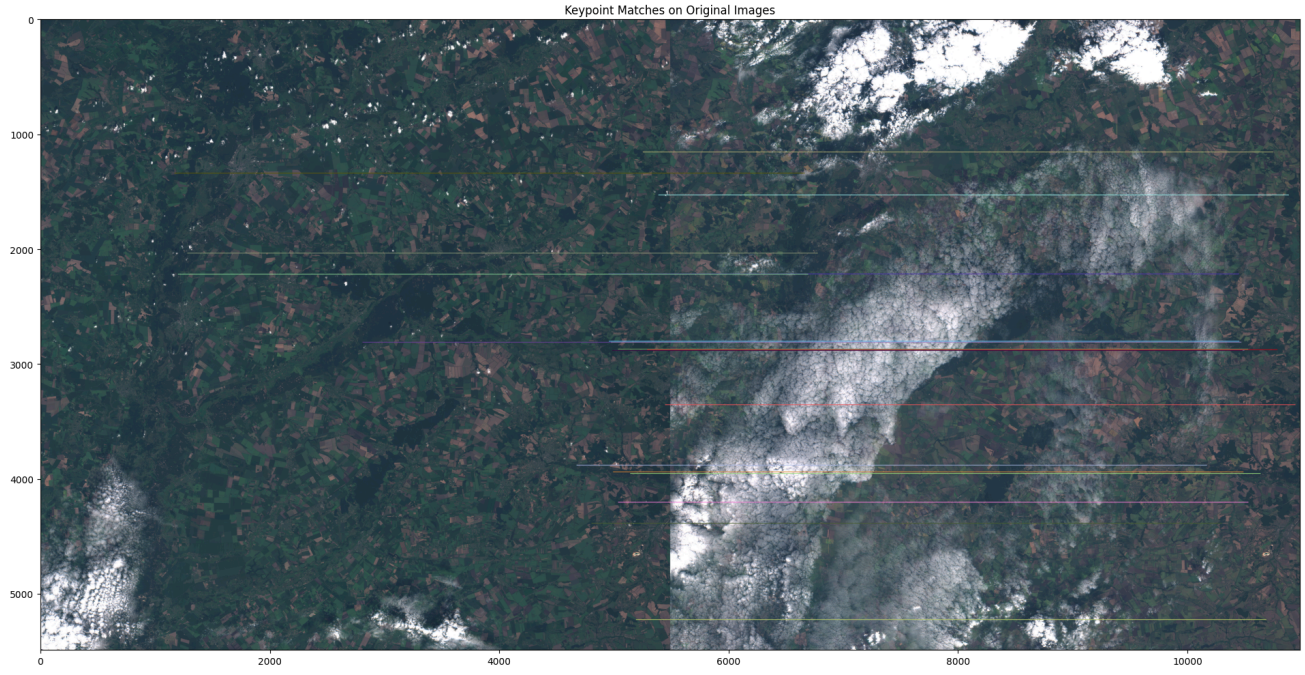
```
# Visualization of matched results:
visualize_matches(df_k, df_match, match_index=51)
```


Original Image 1
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
5000


Original Image 2
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
5000


Processed Image 1
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
5000


Processed Image 2
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
5000


Keypoint Matches on Original Images
0
1000
2000
3000
4000
5000
0
2000
4000
6000
8000
10000

72.72

Now let's display the dataframe where the correlation is less than 50%.

```
df_match[df_match['similarity (%)'] < 50]
```

	image1	image2	similarity (%)
11	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UYA_20160212T084052_T...	19.62
12	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UYA_20160330T082542_T...	12.78
13	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UYA_20160405T085012_T...	21.51
14	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UYA_20160502T083602_T...	21.37
15	Sentinel2 IMG Mask 5k\T36UXA_20180726T084009_T...	Sentinel2 IMG Mask 5k\T36UYA_20160509T082612_T...	12.78
...	...	...	...
1210	Sentinel2 IMG Mask 5k\T36UYA_20190810T083609_T...	Sentinel2 IMG Mask 5k\T36UYA_20190815T083601_T...	38.48
1211	Sentinel2 IMG Mask 5k\T36UYA_20190810T083609_T...	Sentinel2 IMG Mask 5k\T36UYA_20190825T083601_T...	32.49

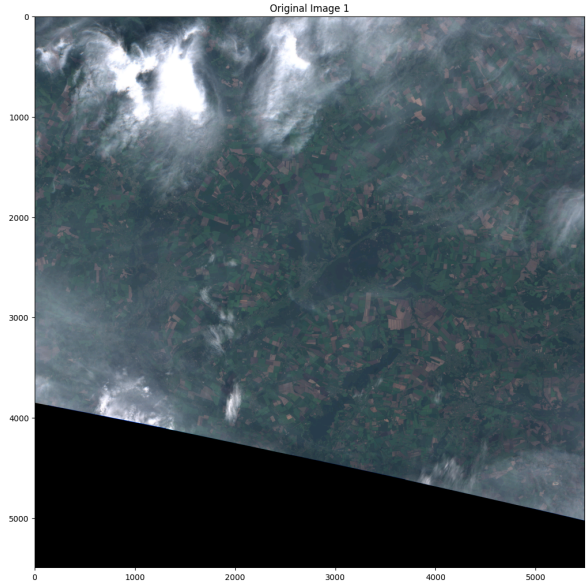
Here, UXA doesn't match only UYA, which is also very good. But the result of UYA with itself is already worse. This may be due to some reasons:

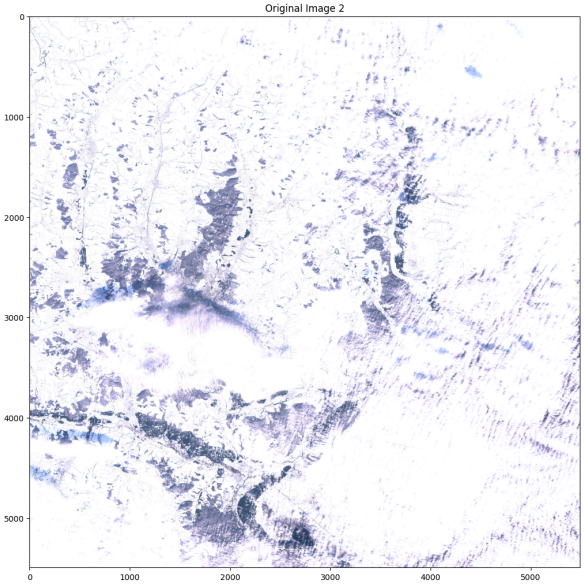
- UYA has more varieties than UXA
- UYA has more “problem” images, where there is no part of the image, cloudiness or highly highlighted images.

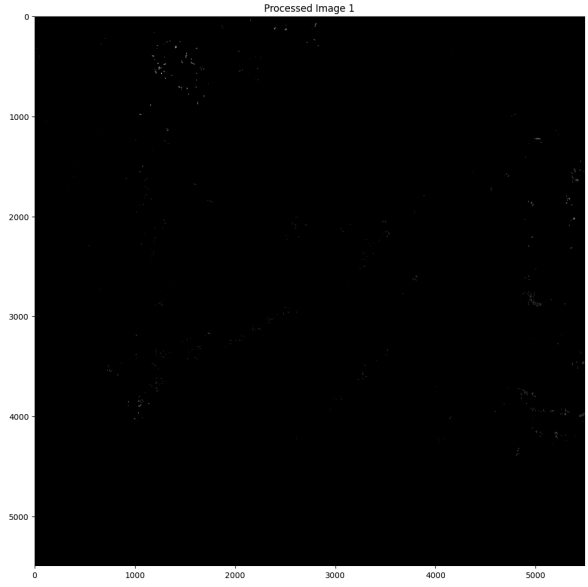
Nevertheless - even at 40% matching the algorithm is pretty good at finding the same keypoints.

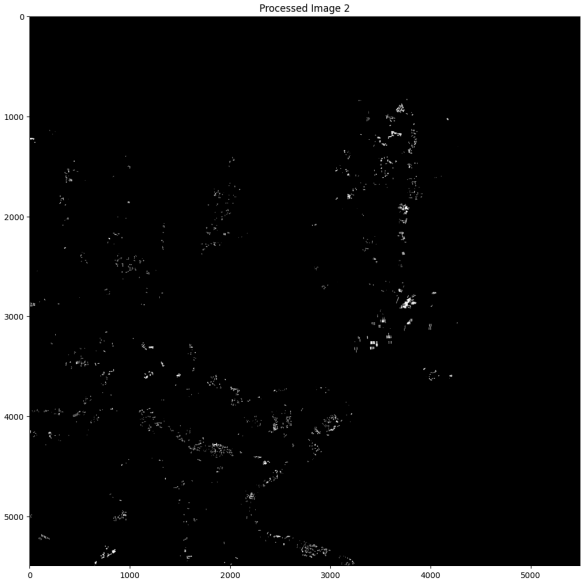
Also it should be noted that IDE in which the project was made (DataSpell) for some reason badly displays very bright images in the output. Moreover, such images change the color of neighboring images. When saving the image to a folder, the colors are saved in the correct variant.

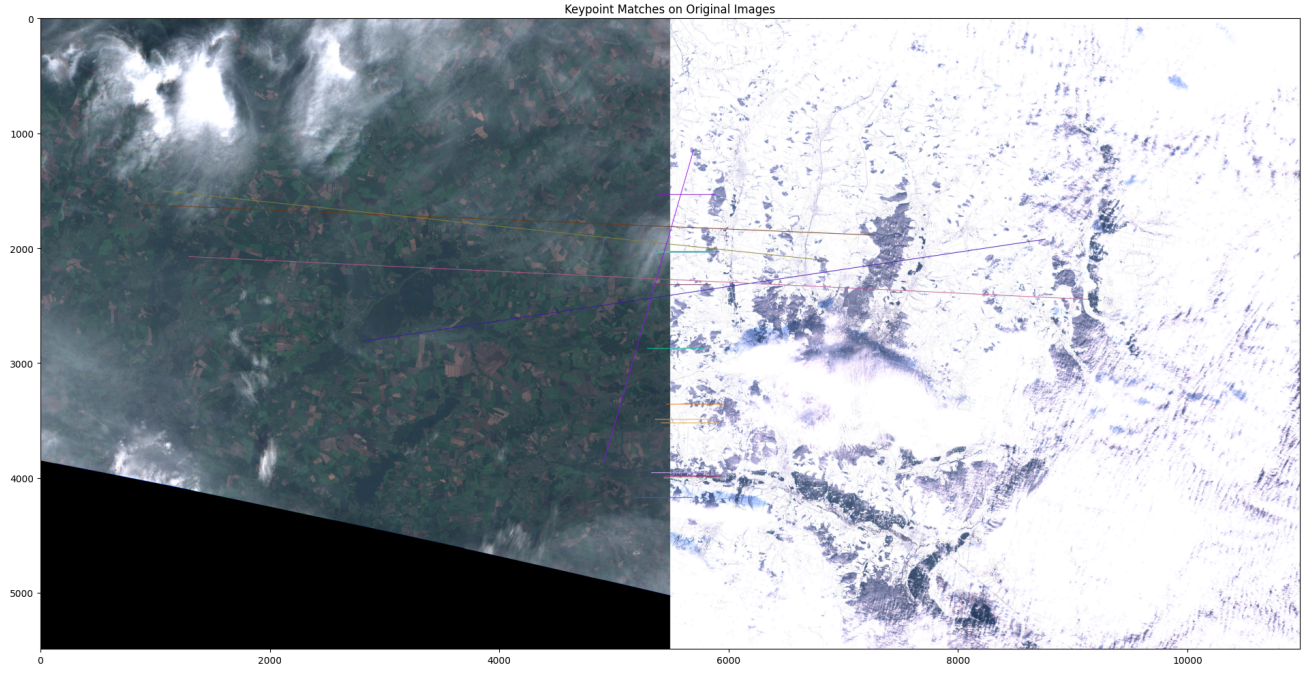
```
# Visualization of unmatched results:
visualize_matches(df_k, df_match, match_index=11)
```


Original Image 1
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
5000


Original Image 2
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
5000


Processed Image 1
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
5000


Processed Image 2
0
1000
2000
3000
4000
5000
0
1000
2000
3000
4000
5000


Keypoint Matches on Original Images
0
1000
2000
3000
4000
5000
0
2000
4000
6000
8000
10000

19.62

```
# Visualization of unmatched results:
visualize_matches(df_k, df_match, match_index=1191)
```