HACETTEPE UNIVERSITY DEPARTMENT OF GEOMATICS ENGINEERING

GMT431- PHOTOGRAMMETRIC IMAGE ANALYSIS

ASSIGNMENT-1 REPORT

Egehan YAĞLICI

21967847

First of all, with the advantages of contemporary technology, photogrammetric image analysis plays a significant role in remote sensing and image processing. This study uses the Python programming language to address problems including converting world coordinates to pixel coordinates, identifying building borders in the image, and estimating a building's height from the ground.

The extensive library and module support of Python enhances photogrammetry research. The objective of this paper is to utilize image processing techniques, apply photogrammetric concepts, and examine the outcomes.

The process involves converting world coordinates to pixel coordinates, figuring out the building's boundaries, and estimating the building's height from the ground. The study uses Python programming to assist each stage to make these processes more comprehensible.

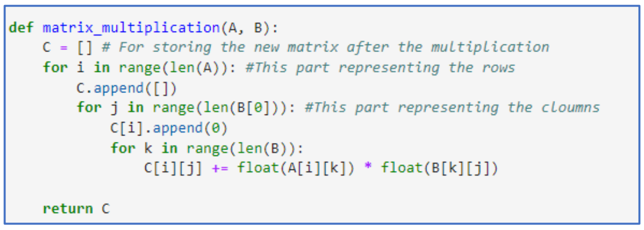
To make it clear ‘’the answers to the 2nd and 4th questions will be given together in the report and it will also explain why the 3rd question was not answered.’’

Q1) Please perform the Object Space to Image Space transformation with the help of the Projection Matrix in homogeneous coordinates.

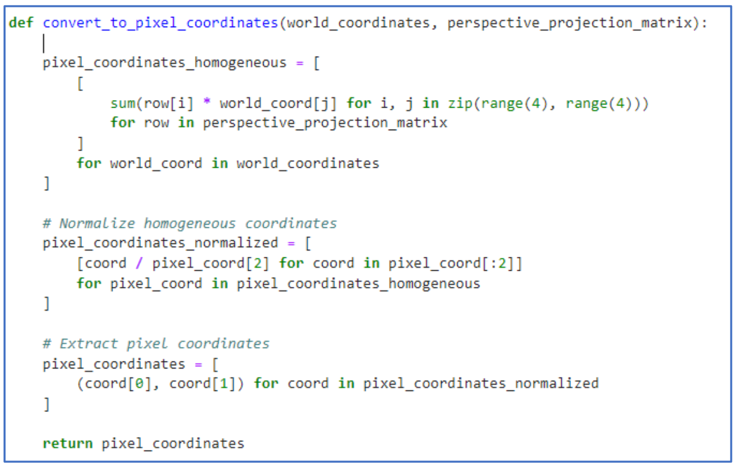
The use of libraries is forbidden for this phase. I have included the NumPy library as well because it makes the procedure easier overall.

The things we use for this question are the internal and external camera parameters, based on the information that has been provided. The transformation is carried out using nine coordinates that contain a building's roof corners. Placing the computed pixel coordinates on the image we have given us control over the transformation process. The operations carried out, the matrices represented in the Python code, and the graphics below are all obtained from the "Camera Projection" PDF file, which contains the lecture notes for the fourth week.

Since we did not have a library to use, I first started by writing the functions required for matrix multiplication and coordinate transformation.



```
def matrix_multiplication(A, B):
    C = [] # For storing the new matrix after the multiplication
    for i in range(len(A)): #This part representing the rows
        C.append([])
        for j in range(len(B[0])): #This part representing the cloumns
            C[i].append(0)
            for k in range(len(B)):
                C[i][j] += float(A[i][k]) * float(B[k][j])

    return C
```



```
def convert_to_pixel_coordinates(world_coordinates, perspective_projection_matrix):

    pixel_coordinates_homogeneous = [
        [
            sum(row[i] * world_coord[j] for i, j in zip(range(4), range(4)))
            for row in perspective_projection_matrix
        ]
        for world_coord in world_coordinates
    ]

    # Normalize homogeneous coordinates
    pixel_coordinates_normalized = [
        [coord / pixel_coord[2] for coord in pixel_coord[:2]]
        for pixel_coord in pixel_coordinates_homogeneous
    ]

    # Extract pixel coordinates
    pixel_coordinates = [
        (coord[0], coord[1]) for coord in pixel_coordinates_normalized
    ]

    return pixel_coordinates
```

The explanations of these functions are included in the code, but ı have put the visuals in their simplest form here so that they do not take up unnecessary space, and ı will explain the explanations later.

The main point to understand here is that Python stores the matrices ı create as a list, and operations such as matrix multiplication should be written taking this into consideration. Because NumPy saves matrices as arrays and performs operations easily with functions such as 'matmul'. Therefore, the term list comprehension must be known here to perform matrix operations correctly.

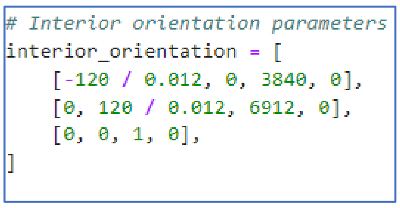
For pixel_coordinates_homogeneous = The matrix-vector multiplication between perspective_projection_matrix and each set of world_coordinates is performed in this section using a list comprehension. The outcome is kept in a homogeneous coordinates representing variable named pixel_coordinates_homogeneous.

pixel_coordinates_normalized = In this section, homogeneous coordinates are normalized by dividing each coordinate by its third element. A variable named pixel_coordinates_normalized holds the outcome.

pixel_coordinates = The first two elements of each normalized coordinate are extracted in this part to generate a list of tuples called pixel_coordinates. The generated pixel coordinates are these tuples.

Same explanations for the function include in the code too.

We first develop our functions and then utilize the available data to create the required matrices. These are the rotation matrix and camera's internal-external parameters that were given to us.



```
# Interior orientation parameters
interior_orientation = [
    [-120 / 0.012, 0, 3840, 0],
    [0, 120 / 0.012, 6912, 0],
    [0, 0, 1, 0],
]
```

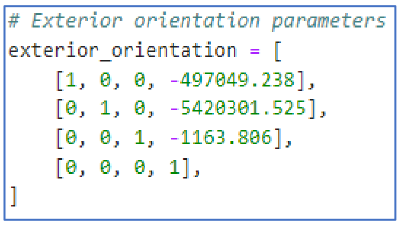
$$\begin{bmatrix} f/s_x & 0 & o_x & 0 \\ 0 & f/s_y & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

We use opposite version of this but just first row :) I figured out by trial and error.

$$\underbrace{\underbrace{\overset{\text{Film plane to pixels}}{\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix}}}_{\mathbf{M}_{\text{aff}}} \underbrace{\overset{\text{Perspective projection}}{\begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}}_{\mathbf{M}_{\text{proj}}}}_{\mathbf{M}_{\text{int}}}$$

This is an intrinsic matrix and what we do here is film plane to pixels and perspective projection written together. There are 6 parameters in the Affine Transformation: two translations, two scale factors and two rotation degrees. Focal length, pixel size, and translation are used in this matrix. This translation is the x and y values in the 'file coordinate system'. (focal length= 120mm, pixel size= 0.012mm, ox=3840(col), oy= 6912(row))

Let's examine exterior orientation parameters. The projection center coordinates are part of this exterior parameter's matrix. (X= 497049.238, Y= 5420301.525, Z= 1163.806)

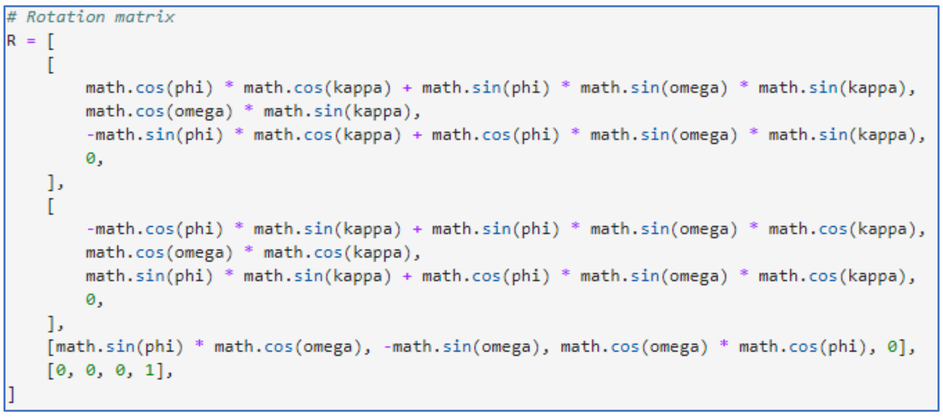


```
# Exterior orientation parameters
exterior_orientation = [
    [1, 0, 0, -497049.238],
    [0, 1, 0, -5420301.525],
    [0, 0, 1, -1163.806],
    [0, 0, 0, 1],
]
```

$$\begin{pmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Sometimes, the image and the camera coordinate systems have opposite orientations. That's why these are negative values.

We have a rotation matrix which given from the assignment pdf, this matrix refers to exact this one. After we see the matrix, we will look at our rotation angles for this question.



```
# Rotation matrix
R = [
    [
        math.cos(phi) * math.cos(kappa) + math.sin(phi) * math.sin(omega) * math.sin(kappa),
        math.cos(omega) * math.sin(kappa),
        -math.sin(phi) * math.cos(kappa) + math.cos(phi) * math.sin(omega) * math.sin(kappa),
        0,
    ],
    [
        -math.cos(phi) * math.sin(kappa) + math.sin(phi) * math.sin(omega) * math.cos(kappa),
        math.cos(omega) * math.cos(kappa),
        math.sin(phi) * math.sin(kappa) + math.cos(phi) * math.sin(omega) * math.cos(kappa),
        0,
    ],
    [math.sin(phi) * math.cos(omega), -math.sin(omega), math.cos(omega) * math.cos(phi), 0],
    [0, 0, 0, 1],
]
```

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```python
omega_d = (2.05968 / 400) * 360
phi_d = (0.67409 / 400) * 360
kappa_d = (199.23470 / 400) * 360

# Convert from degree to radian
omega = math.radians(omega_d)
phi = math.radians(phi_d)
kappa = math.radians(kappa_d)
```

So, our rotation angles are provided in grad format. We must convert it to degrees in order to use it, and since this is an easy operation, I thought it would be quicker to do this by hand rather than by building a function. These are the angles which we use in rotation matrix.

We now have all the information we need to convert the world coordinates of the building's roof into pixel coordinates.

```python
# Projection matrix
perspective_projection = matrix_multiplication(interior_orientation, R)
perspective_projection_final = matrix_multiplication(perspective_projection, exterior_orientation)
```

$$\begin{pmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 1 & -c_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
Perspective Projection
---------------------------------------
[10039.313845252786, -244.3625364179051, 3728.0107922739617, -3669853349.3550463]
[-50.47965979268584, -10217.632115389333, 6585.832503071403, 55400073181.98617]
[0.010582841934294953, -0.03234773381099907, 0.9994206459613922, 168911.14564244283]
```

As you can see, we use the matrix multiplication function we previously built to perform out these multiplication operations in this code from left to right. To establish pixel coordinates, we will process the perspective projection matrix that we acquired using world coordinates. I add the result of the Perspective Projection.

```python
a = np.array([0.123456789121212,2,3], dtype=np.float16)
print("16bit: ", a[0])

a = np.array([0.123456789121212,2,3], dtype=np.float32)
print("32bit: ", a[0])

b = np.array([0.1234567891212121212,2,3], dtype=np.float64)
print("64bit: ", b[0])
```

- 16bit: 0.1235
- 32bit: 0.12345679
- 64bit: 0.12345678912121212

This is the difference between float types

As additional information, if you did this with NumPy, you would see float64 type numbers. These contain more data than the float32 type numbers we obtained. However, in our current process, this detail does not mean anything because our numbers are not that complex. Source of the image is "The real difference between float32 and float64 - Stack Overflow".

```python
world_coordinates = [
    [497113.220, 5419946.461, 287.650, 1],
    [497130.081, 5419948.322, 287.650, 1],
    [497132.582, 5419926.619, 287.700, 1],
    [497128.209, 5419926.155, 287.650, 1],
    [497130.884, 5419901.053, 287.650, 1],
    [497141.373, 5419902.170, 287.300, 1],
    [497142.131, 5419895.066, 287.650, 1],
    [497118.956, 5419892.610, 287.650, 1],
    [497113.220, 5419946.461, 287.650, 1]
]
```

$$\begin{pmatrix} U \\ V \\ W \\ 1 \end{pmatrix}$$

The coordinates we see here are the corner coordinates of our building roof. Here I added an extra dimension by hand, this is because we always add a dimension when working with homogeneous coordinates. This refers to the picture next to the coordinates. After determining the real-world coordinates and our perspective projection matrix, we use the function we wrote (convert_to_pixel_coordinates) to calculate the pixel coordinates.

```
# Convert to pixel coordinates using the perspective projection matrix
pixel_coordinates = convert_to_pixel_coordinates(world_coordinates, perspective_projection_final)
```

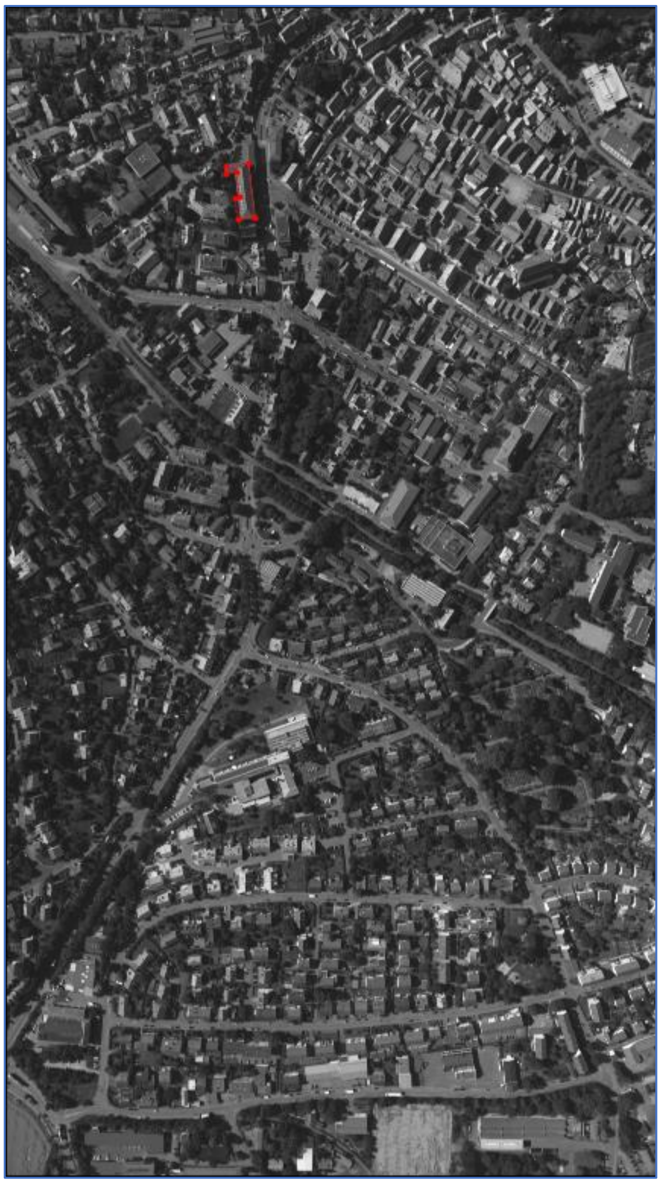
This section of the code handles the translation of 3D world coordinates to 2D pixel coordinates. To represent coordinates in homogeneous form, "pixel_coordinates_homogeneous" is a variable that holds the results of multiplying "perspective_projection_matrix" by each set of "world_coordinates" using list comprehension. The variable "pixel_coordinates_normalized" is then produced by normalizing these homogeneous coordinates by dividing by the third element of each coordinate. The next step is to generate a list of tuples named "pixel_coordinates" that contains the first two elements of each normalized coordinate. This list represents the final 2D pixel coordinates. The perspective projection transformation is largely captured by this set of procedures, which converts three-dimensional world points into equivalent two-dimensional pixel coordinates.

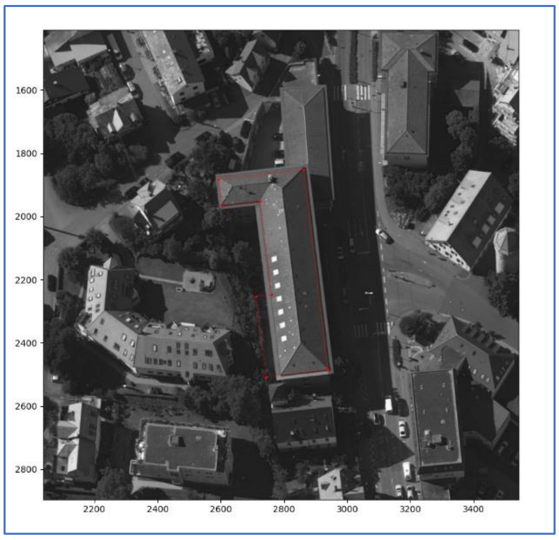
```
----------------------------------------
---> Pixel Coordinates: (2938.3452497870217, 2484.7347824591498)
----------------------------------------
---> Pixel Coordinates: (2743.213186527079, 2508.085141001929)
----------------------------------------
---> Pixel Coordinates: (2710.216496287416, 2253.0345881900257)
----------------------------------------
---> Pixel Coordinates: (2760.9366348076373, 2247.452531144176)
----------------------------------------
---> Pixel Coordinates: (2725.3505760252447, 1952.197743758674)
----------------------------------------
---> Pixel Coordinates: (2604.165932604981, 1968.1017329853264)
----------------------------------------
---> Pixel Coordinates: (2593.5796075680564, 1882.5592302232726)
----------------------------------------
---> Pixel Coordinates: (2862.391769630859, 1851.7166617644239)
----------------------------------------
---> Pixel Coordinates: (2938.3452497870217, 2484.7347824591498)
----------------------------------------
```

These are the pixel coordinates we obtained as a result of our last operation. We can check the accuracy of our result by trying it on the image. If the coordinates really fit on the roof corners of a building, we can consider our operation correct. Here is the code I built to make the visuals. It doesn't need an explanation, therefore I won't get into it here. The descriptions on the image allow for the desired adjustments to be performed.

```
import matplotlib.pyplot as plt


img = plt.imread('61.jp2') # read the image
fig = plt.figure(figsize=(20, 20)) # You can adjust size of the result from here etc 15 15
plt.imshow(img,cmap='gray') # It will show the image in gray scale

# Marking each pixel coordinate on the image
for coord in pixel_coordinates:
    plt.scatter(coord[0], coord[1], color='red', s=3) # You can adjust point visual from here
# Combine them and draw lines
for i in range(len(pixel_coordinates) - 1):
    x_vals = [pixel_coordinates[i][0], pixel_coordinates[i + 1][0]]
    y_vals = [pixel_coordinates[i][1], pixel_coordinates[i + 1][1]]
    plt.plot(x_vals, y_vals, color='red', linestyle='--', linewidth=0.5) #You can adjust the line visual from here

plt.show()
```

For the result scroll the page…



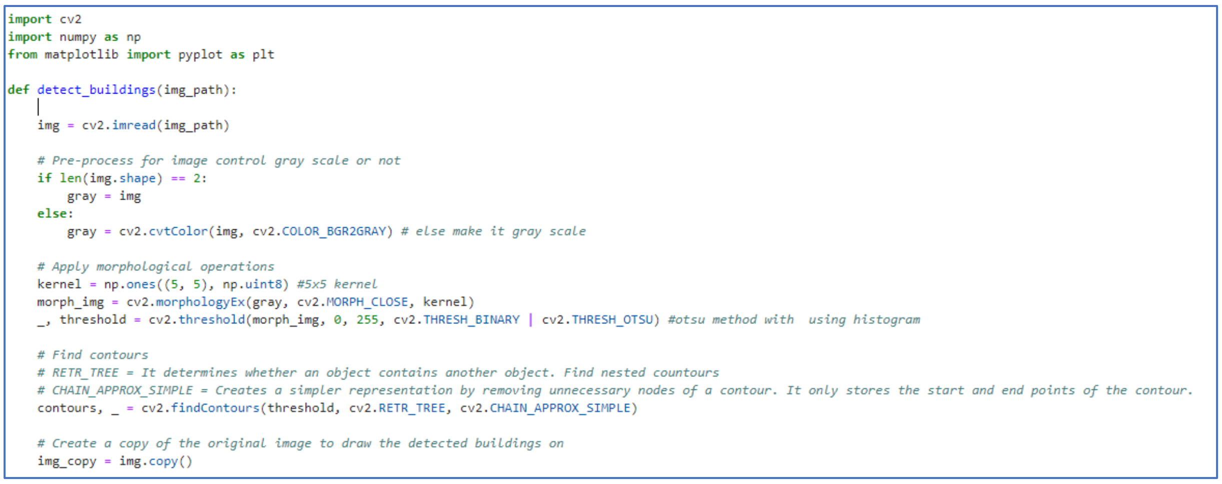


We can understand that our result is correct from the pixel coordinates in the images. Since the world coordinates we have contain minor errors, there has been a slight shift, but this is not due to the process.

Q2) You are required to plot the projected boundaries of all buildings on image 61.jp2.

As mentioned at the opening of the report, at this point, questions 2 and 4 will be addressed together. The code provided as an answer will be detailed first, followed by a discussion of the various approaches that were attempted.

Like with the last question, I wanted to build a function for myself because that's how I work. In addition, I made a library of my own functions so that I could use them quickly in the future. This function can also perform other desired tasks by changing the parameters within it. But it should not be forgotten that the settings should be changed according to the image itself.



```python
import cv2
import numpy as np
from matplotlib import pyplot as plt

def detect_buildings(img_path):

    img = cv2.imread(img_path)

    # Pre-process for image control gray scale or not
    if len(img.shape) == 2:
        gray = img
    else:
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # else make it gray scale

    # Apply morphological operations
    kernel = np.ones((5, 5), np.uint8) #5x5 kernel
    morph_img = cv2.morphologyEx(gray, cv2.MORPH_CLOSE, kernel)
    _, threshold = cv2.threshold(morph_img, 0, 255, cv2.THRESH_BINARY | cv2.THRESH_OTSU) #otsu method with  using histogram

    # Find contours
    # RETR_TREE = It determines whether an object contains another object. Find nested countours
    # CHAIN_APPROX_SIMPLE = Creates a simpler representation by removing unnecessary nodes of a contour. It only stores the start and end points of the contour.
    contours, _ = cv2.findContours(threshold, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)

    # Create a copy of the original image to draw the detected buildings on
    img_copy = img.copy()
```

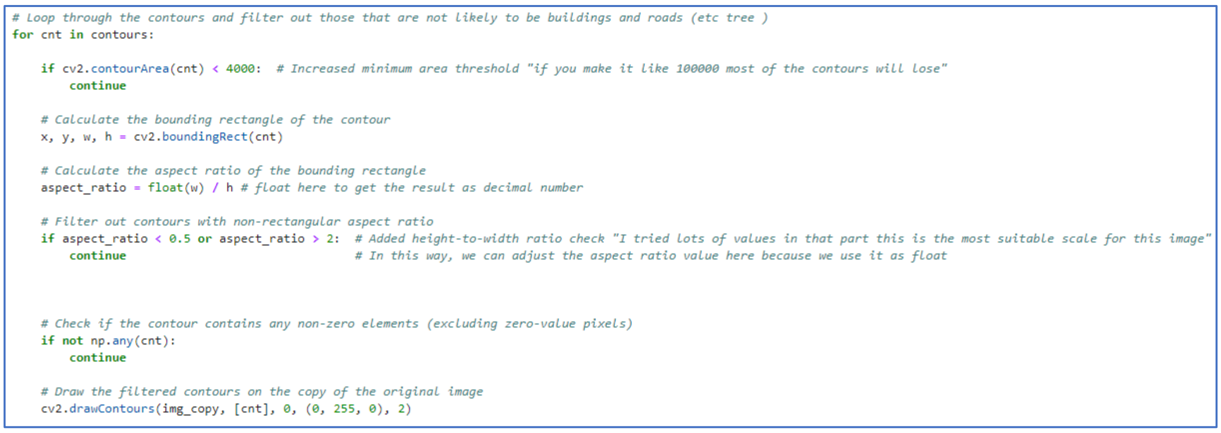
This function has full descriptions in the code; but, in order to keep the image size manageable, I have included it here without this information. It includes the initial portion of the function that we looked over earlier; we will go over the remaining portion after explaining this one.

In the first stage, we check whether the image we receive is gray scale, and if the image is RGB, we convert it to gray scale. Following that, we carry out morphological operations.

This function uses morphological operations along with a thresholding mechanism among other operations. Morphological operations, such as emphasizing edges or combining/dividing objects, are used to emphasize or correct forms and structures in the image. This code applies closure, which effectively closes minor gaps in the image and facilitates object combination. Through the use of 5x5 sized kernels, occlusion is applied to the image, making objects more distinct.

When applying the Otsu method in particular, thresholding is utilized to divide the pixel values in an image into two distinct zones by comparing them to a predetermined value. This technique examines at the image histogram to get the optimal threshold value. As a result, its goal is to get the picture ready for contour detection, Contours represent the boundaries of objects on an image. They consist of a series of points and are used in applications such as object detection and recognition. All the previous stages were done so that we can find the contours more easily and accurately. These preprocessing steps make the contour detection approach more accurate and effective. The study goes deeper into the technicalities of these methods, which include Canny or Harris corner detection, but were specifically excluded from being used due to the complex structure of this image assignment. To determine which method worked best, I tried a lot of them.

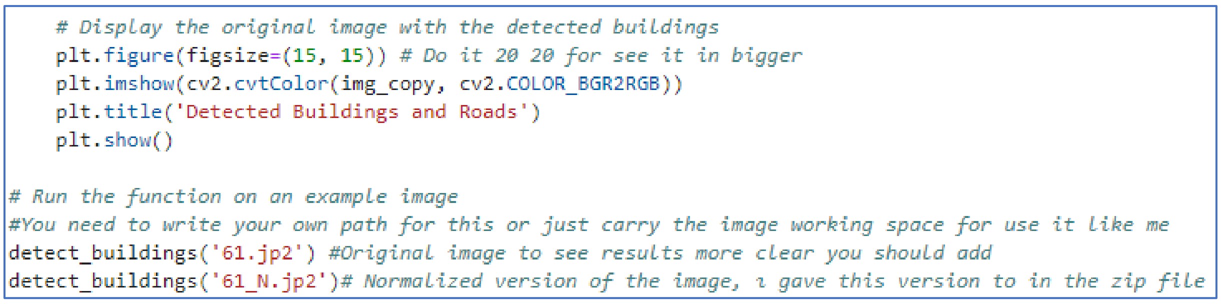
After analyzing this section of the code, we can move on to the rest of the function. Scroll the page…



```python
# Loop through the contours and filter out those that are not likely to be buildings and roads (etc tree )
for cnt in contours:

    if cv2.contourArea(cnt) < 4000:  # Increased minimum area threshold "if you make it like 100000 most of the contours will lose"
        continue

    # Calculate the bounding rectangle of the contour
    x, y, w, h = cv2.boundingRect(cnt)

    # Calculate the aspect ratio of the bounding rectangle
    aspect_ratio = float(w) / h # float here to get the result as decimal number

    # Filter out contours with non-rectangular aspect ratio
    if aspect_ratio < 0.5 or aspect_ratio > 2:  # Added height-to-width ratio check "I tried lots of values in that part this is the most suitable scale for this image"
        continue                               # In this way, we can adjust the aspect ratio value here because we use it as float



    # Check if the contour contains any non-zero elements (excluding zero-value pixels)
    if not np.any(cnt):
        continue

    # Draw the filtered contours on the copy of the original image
    cv2.drawContours(img_copy, [cnt], 0, (0, 255, 0), 2)
```

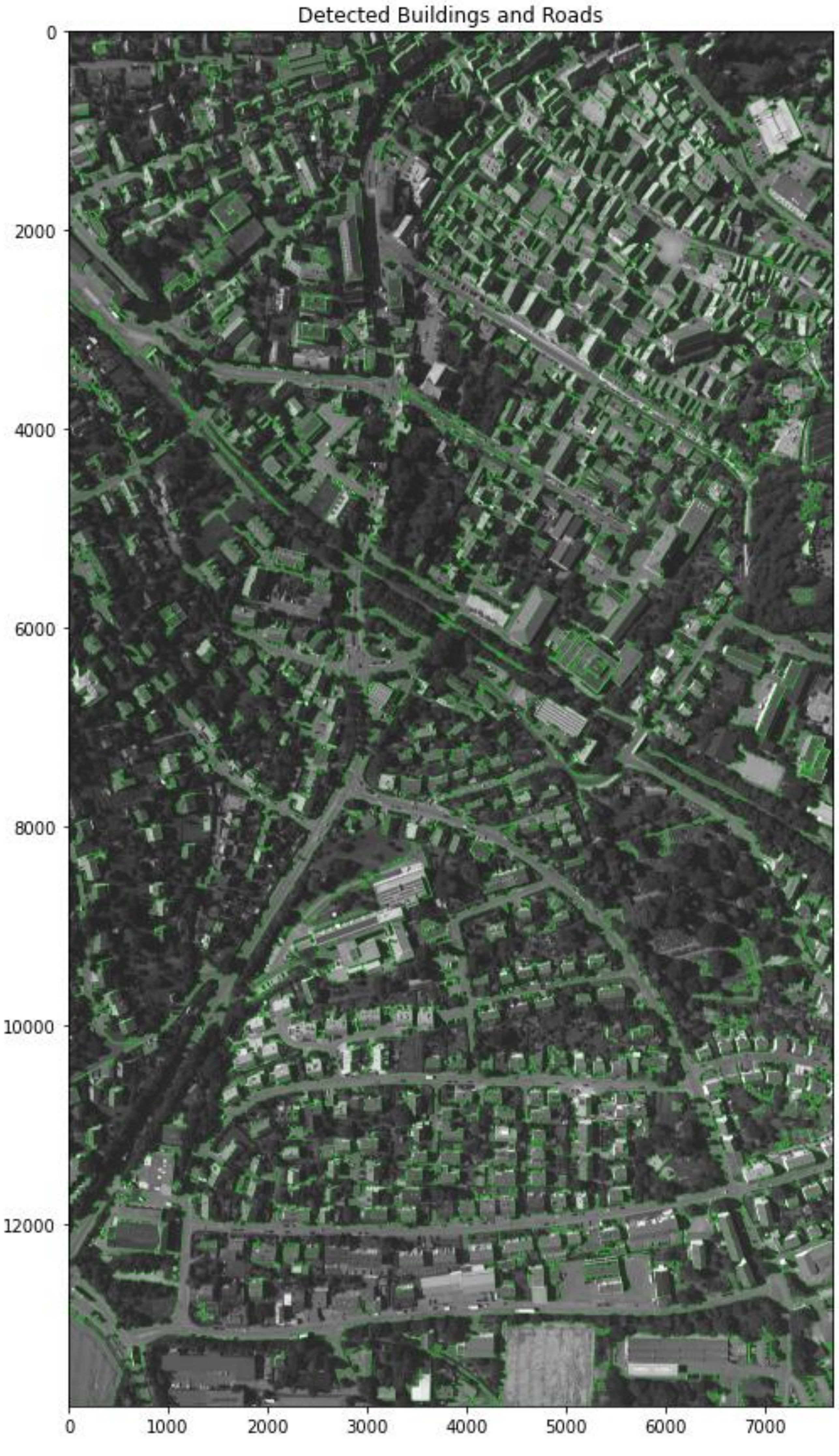
In this section of code, features that represent roads and buildings are identified and isolated, and features that represent trees are excluded. This is done by contour filtering. Unwanted contours are filtered out using a set of criteria as a loop moves through the contours. First, shapes smaller than 4000 are disregarded in favor of larger, more significant structures. To achieve increased accuracy, bounding rectangles are made for every contour, and the aspect ratios of these rectangles are found by dividing the width (w) by the height (h). Those contours that have an uneven rectangular ratio of 0.5 to 2 are removed. By doing this procedure, the shapes resembling standard building and road structures are protected. Furthermore, a filter to reject contours devoid of non-zero elements has been incorporated, therefore removing contours that symbolize empty spaces. By utilizing green and a pixel thickness of two, the filtered contours are visually emphasized on a copy of the original image. These filtering and sketching procedures improve the image's recognized structures' accuracy and significance, bolstering the results of further analysis and interpretation. It is flexible to modify the detection criteria in this filtering stage according to the features of the specific image by adjusting the parameters.

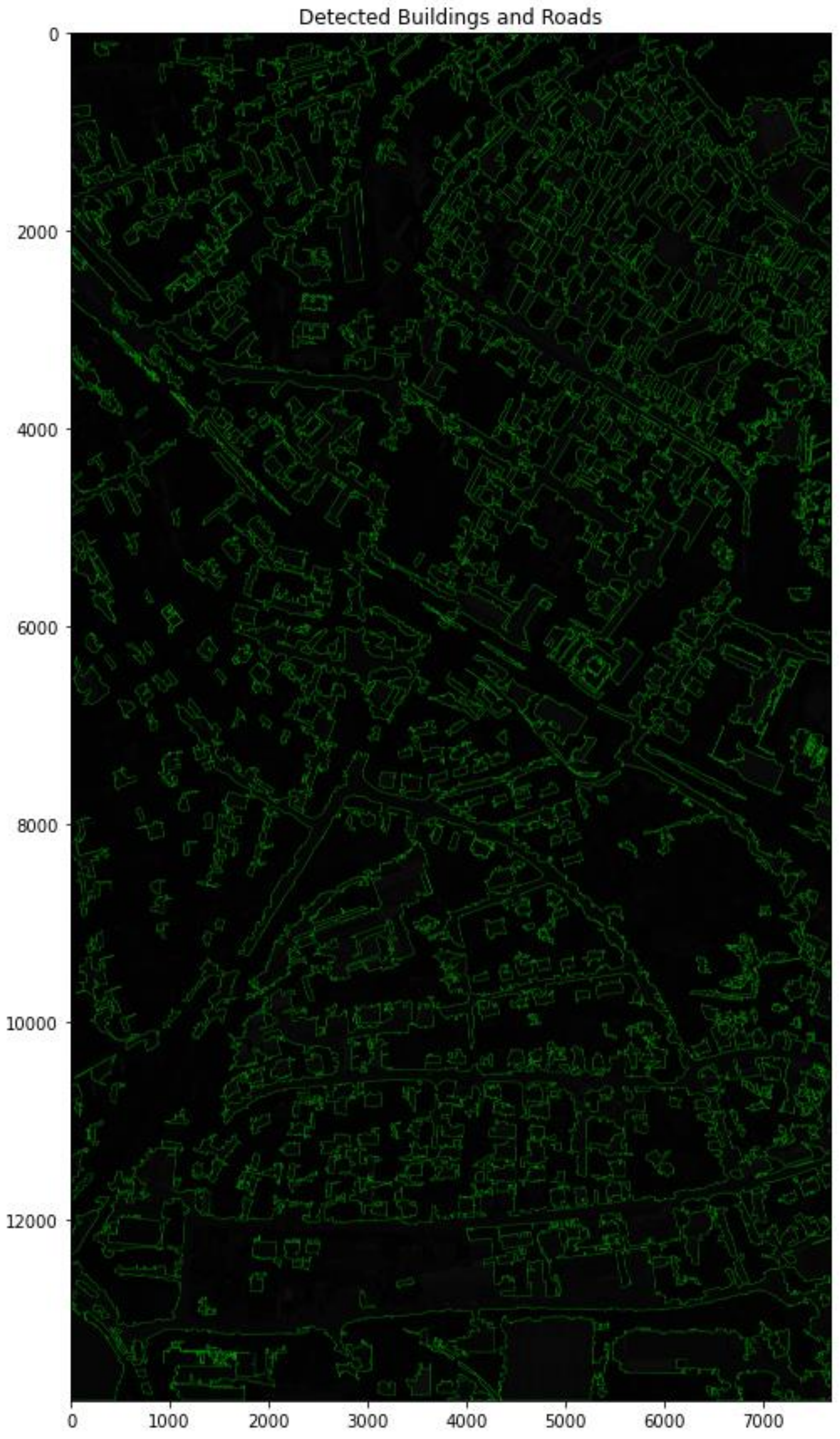
For a better visualization, I used two different visual outputs, one is the original version of the image we have and the other is its normalized version. I added this version to the zip file. If you open it in the same environment, you can use both without typing the path. Now that our function is complete, we can just call it with the image and see the results.



```python
    # Display the original image with the detected buildings
    plt.figure(figsize=(15, 15)) # Do it 20 20 for see it in bigger
    plt.imshow(cv2.cvtColor(img_copy, cv2.COLOR_BGR2RGB))
    plt.title('Detected Buildings and Roads')
    plt.show()

# Run the function on an example image
#You need to write your own path for this or just carry the image working space for use it like me
detect_buildings('61.jp2') #Original image to see results more clear you should add
detect_buildings('61_N.jp2')# Normalized version of the image, ı gave this version to in the zip file
```

You can see the results on the page below…


Detected Buildings and Roads
0
2000
4000
6000
8000
10000
12000
0
1000
2000
3000
4000
5000
6000
7000


Detected Buildings and Roads
0
2000
4000
6000
8000
10000
12000
0
1000
2000
3000
4000
5000
6000
7000

Now we can talk about the 4th question: Please comment/discuss/explain the problems/reasons/causes you face in the output building boundaries projected on image 61.jp2.

Firstly, The main reason why traditional methods cannot be used is that the image to be processed is monochromatic (1 band). If our image was in the RGB color space, it would be easier to process it with Canny and similar methods. A faster and more practical solution would be to convert this image to RGB through applications on the internet. However, I did not choose this alternative because I was extremely attracted to deep learning models.

Performing this process using deep learning models will enable similar operations to be performed more efficiently and quickly in the future. The decision to pursue research in this direction is directly related to my personal satisfaction. Because we had already learned about this subject in our lessons and knew the methods to be used in this field.

The main motivation of this approach is to explore the potential offered by deep learning models and understand the impact of this technology on future projects. Overcoming the limitations of traditional methods and turning to new generation technologies offers me both an enjoyable experience in the learning process and the chance to explore the potential in this field more closely.

We can now discuss the primary subject after I've mentioned these.

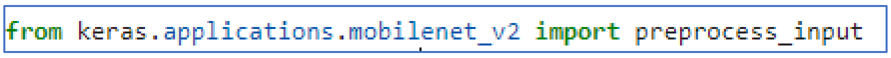
Realizing that traditional image processing methods were insufficient for the task of determining building boundaries due to the image we had, because the image was a 1-band image, I accessed it via GitHub and examined the codes with the post I found on the Hugging Face site of a Turkish student. However, I thought that it might not be appropriate to use this model directly and decided to do my own experiments on this subject. Since I do not think it is appropriate, I will leave the link for huggin face instead of sharing the link directly.

I have accessed many resources on the Kaggle platform about many pre-trained models that can be used with popular libraries such as TensorFlow and PyTorch. However, directly obtain and using these models was very difficult and it was not clear whether I could achieve a definitive result. That's why I decided to try one of the most popular ones.

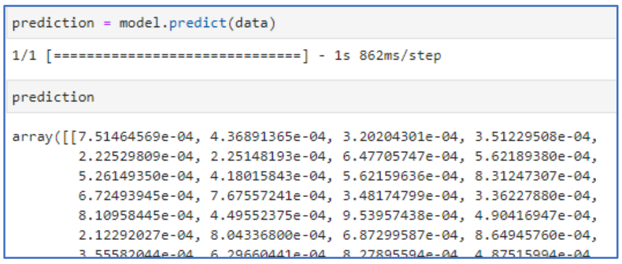
I used the MobileNet V2 model in my own tests. However, I chose not to share my results with this model because it would have detracted from the purpose of the assignment. However, thanks to this experience, I was able to understand the basic functioning of these models and developed them for a specific purpose for myself.

When using deep learning models, it is important to bring the visual into a format that the model can analyze. This step is generally performed with predefined operations, but in some special cases user intervention may be required.

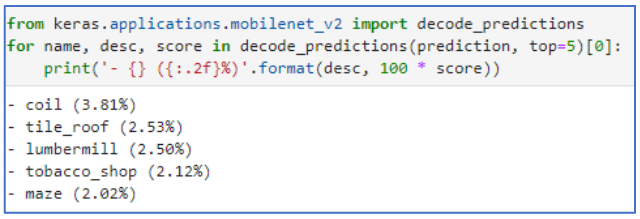
As a brief information, the operation can be performed quickly with these functions in MobileNet V2. However, in our scenario, a few operations were required beforehand. Because most models of this type require images with RGB color channels, so I had to add an additional channel to our image. The model was programmed to read the array consisting of 3 columns.



```
from keras.applications.mobilenet_v2 import preprocess_input
```



```
prediction = model.predict(data)

1/1 [==============================] - 1s 862ms/step

prediction

array([[7.51464569e-04, 4.36891365e-04, 3.20204301e-04, 3.51229508e-04,
        2.22529809e-04, 2.25148193e-04, 6.47705747e-04, 5.62189380e-04,
        5.26149350e-04, 4.18015843e-04, 5.62159636e-04, 8.31247307e-04,
        6.72493945e-04, 7.67557241e-04, 3.48174799e-04, 3.36227880e-04,
        8.10958445e-04, 4.49552375e-04, 9.53957438e-04, 4.90416947e-04,
        2.12292027e-04, 8.04336800e-04, 6.87299587e-04, 8.64945760e-04,
        3.55582044e-04, 6.29660441e-04, 8.27895594e-04, 4.87515994e-04,
```



```
from keras.applications.mobilenet_v2 import decode_predictions
for name, desc, score in decode_predictions(prediction, top=5)[0]:
    print('- {} ({:.2f}%)'.format(desc, 100 * score))

- coil (3.81%)
- tile_roof (2.53%)
- lumbermill (2.50%)
- tobacco_shop (2.12%)
- maze (2.02%)
```

These are just random examples for the understanding.

During my project process, I knew that the model was not specifically trained for the task of determining building boundaries. However, I chose to try it out of my curiosity about using and training such models, and it was an important step towards using this technology effectively in my own projects in the future. As a result, apart from detecting the building, we have detected different things that do not exist in the meaningless picture, but as I said, our main problem is our visual :D. We can collect and train one of these models and use it.

The operation of these models generally runs on the GPU and CPU base, and CUDA cores are used in this process. However, since the adjustments in this regard are laborious, I have not yet moved on to the model training phase. I would also like to point out that GPU usage is limited to NVDIA RTX series graphics cards and CUDA cores are only available in this generation graphics cards.

This experience, together with the knowledge I gained about using and understanding deep learning models, provided me with the opportunity to acquire a new hobby in line with my goal of using this technology more effectively in my future projects.

-For more information:

-Find Pre-trained Models | Kaggle

-TensorFlow

-Models - Hugging Face

I found an article about our subject when looking up the third question. I became enlightened after that.

I'm managed to find the answers to our second and third questions by reading this article. I saw the sentence [55] (ISPRS Vaihingen 2D Semantic Labeling Dataset) when I got to the article's source section. The task you assigned us represented a tiny portion of this location's data. Upon further investigation, I came to the conclusion that my research strategy and method for answering question 2 were definitely the best one. I'll go into detail about why I say this. But this part in the third questions answer. Yes, I know it will be a little complicated, but this is the order in which I experienced the events and this is my report :)

Q3) Please find the approximate above-ground height of the building. Find an appropriate logical way, to estimate the height of the building using just one image.

First, I found the exact most logical way for this question because this answer can be use whenever you want after when you can manage it. Remote Sensing | Free Full-Text | IM2ELEVATION: Building Height ... - MDPI. This is the source of that article I was talked about.

This article describes, in brief, how a deep learning model is developed to estimate building height from aerial photographs. Lidar point clouds and aerial photos undergo a number of pre-processing operations, and a Convolutional Neural Network (CNN) model is trained on them. A unique loss function is used to optimize the model's performance during training. Building height estimate using the created model and the suggested approach and preprocessing procedures yields successful results.

GitHub - speed8928/IMELE this link go to the code for this task and you can find the training data in here and the model. Additionally, the model trained here has an extremely high accuracy rate. You can find the data sets in GitHub link.

As we can see from here, the logical way for this job is a pre-trained deep learning model or training them with large datasets, which I also considered for the 2nd question. I didn't explain the article in detail because I did so much research that it would have gone on for pages.

Using the source section of this article as I explained at the end of question 2. I found the source of the image we have at https://www.isprs.org/education/benchmarks/UrbanSemLab/2d-sem-label-vaihingen.aspx

And from here I understand that the things I tried in the 2nd question are correct, but the model I use is wrong. Because in the source I will provide, there are models trained directly with the data in the region we have visuals of, and yes, I found the code part of it.

The purpose of collecting this data is; "This paper focuses on automatically identifying urban objects from weather sensor data. The very variable look of items in high-resolution data, such automobiles, streets, buildings, and trees, makes the task more difficult by decreasing intra-class variability and increasing inter-class variability. The primary focus is on precise 2D semantic segmentation that labels various item classes. It is challenging to conduct experimental comparisons between various methods since standard data sets for these machine learning-based processes are lacking. Two cutting edge aerial picture datasets provided by ISPRS WG III/4 are part of this "semantic labeling contest , Both datasets are digital surface models (DSM) created using dense image matching algorithms and very high resolution accurate orthophoto (TOP) tiles. Vaihingen is a tiny village with many of single-family homes as well as modest multi-story structures. On the other hand, Potsdam, with its big building blocks, winding streets, and compact settlement organization, is a good example of a typical historical city.".

And with the data they had, they created a deep learning model and trained it to distinguish and segment objects in the images. If you browse this site a little, you can see the results and scores.

While continuing my research, I came across another article and here was an article and resources written for their object detection model.

Large Selective Kernel Network for Remote Sensing Object Detection

Papers with Code - Large Selective Kernel Network for Remote Sensing ...

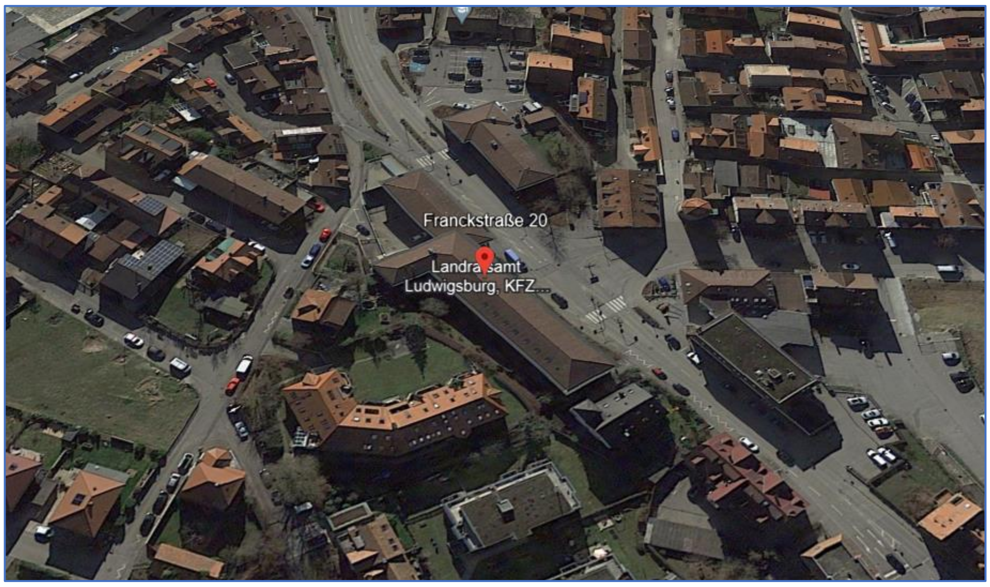
GitHub - zcablii/LSKNet: (ICCV 2023) Large Selective Kernel Network for ...

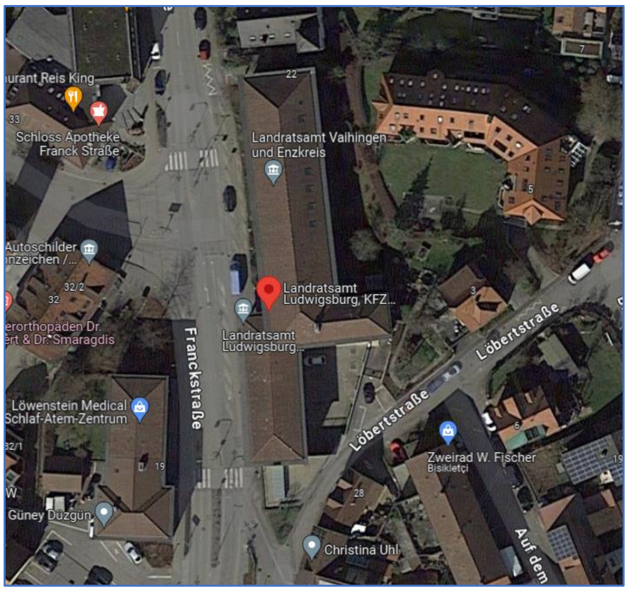
The name of the model used here is LSKNet. We can use this model with the recipes in the link. Scroll the page.

Now I have a guessed answer for question 3.

Adress of the building in the coordinates text 😉

Franckstraße 20, 71665 Vai hingen an der Enz, Almanya




urant Reis King
Schloss Apotheke Franck Straße
Landratsamt Vaihingen und Enzkreis
Autoschilder nzeichen /...
Landratsamt Ludwigsburg, KFZ...
Landratsamt Ludwigsburg...
erorthopäden Dr. ert & Dr. Smaragdis
Franckstraße
Löbertstraße
Löwenstein Medical Schlaf-Atem-Zentrum
Zweirad W. Fischer Bisikletçi
Güney Duzgun
Christina Uhl
Auf dem

It seems to be 12 meters long from wherever we look.

I took the pictures from Google Street View and Google Earth