Reverse search in Mandelbrot set

Kristijonas Silius and Julio Torres Fernández

December 2024

1 Mandelbrot set and assignment.

The Mandelbrot set is a two-dimensional set described by the iteration of a simple function on the complex plane. Within this set, there are convergent and divergent values, with the former laying inside of the set while the latter outside of it. This divergence is visible with the description of colour maps alongside the set, with different colours for different rates of divergence.

The boundary of the Mandelbrot set, for what is most known, is a fractal curve of great complexity that exhibits ever more mesmerizing detail as it is amplified, showing a similar structure on multiple boundary areas or even being replicated at different depths and coordinates.

The set is constructed in the complex set over the iteration of the simple function,

$$f_c(z) = z^2 + c$$

that does not diverge to infinity when an iteration starting at $z = 0$ for its coordinates in the complex plane, c as complex number, takes place:

$$f_c(0),\ f_c(f_c(0)),\ f_c(f_c(f_c(0)))...$$

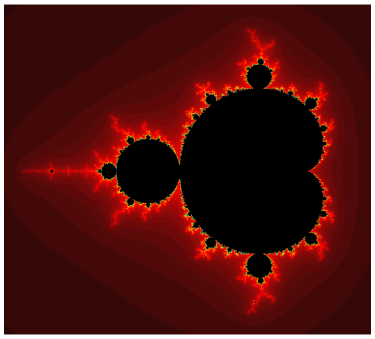
Figure 1: Mandelbrot set.

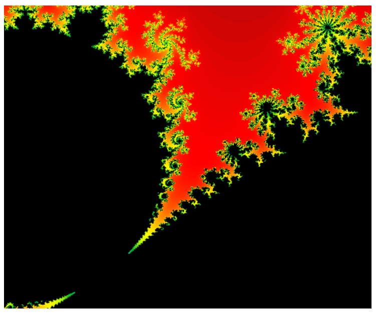
Figure 2: Mandelbrot fragment.

2 Assignment.

The assignment is quite straightforward. Given an image input of a fragment of the Mandelbrot set, identify its coordinates and zoom on the location of the given input from the complete Mandelbrot set. The input given will be a random snippet of the set. Information about its location or zoom level will not be provided. Hence, an analysis or training process is to be created to compare the input and the complete set to identify the coordinates of the given image.

The analytical procedure, *backward exploration process*, would consist on a mathematical analysis of the parameters that would generate the structure of the given image and locate those coordinates within our generated Mandelbrot set. For the training process, *forward exploration process*, a number of images is generated of random fragments and zoom levels of the Mandelbrot set. These images, and their coordinate information, are used to train a CNN model to identify the coordinates of the provided image.

3 Procedures: CNN

As mentioned, the goal of this project is to create a program that can identify the structure of a given fragment of the Mandelbrot set and identify its coordinates. To that end, both exploration processes described previously are applied in this project and are described below.

3.1 Forward exploration process. *CNN.*

The first exploration process to discuss in this project is the training of a model to do image analysis tasks. A Convolutional neural network, **CNN**, architecture of deep learning that learns directly from images. A CNN is made up of several layers that processes and transforms and input to produce an output, coordinates in our case.

Within the neural network we differenciate different layers of neurons: input layer, hidden layer and output layer. In a typical neural network all the neurons of the input layer are connected to all neurons of the hidden layer. On the other hand, in a CNN each neuron of the hidden layer is connected to a small (or limited) number of input layer neurons. This, in turn, affects the way weights and biases are described in the learning process of hidden neurons. For typical NNs each hidden neuron has its own weights that are updated upon each new input of training data. Whereas for CNNs, all hidden neurons share the same weights and biases. Thus, all neurons are capable of detecting the same features. With that in mind, a CNN can have multiple hidden layers and each has the capacity of detecting a specific feauture. The more hidden layers, the more complexity is learned from the images. Furthermore, with activation and pooling steps the output of each neuron is simplified, reducing the number of parameters that the model needs to learn.

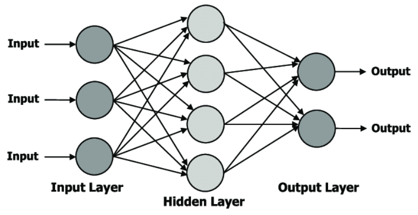

Figure 3: Typical Neural Network.

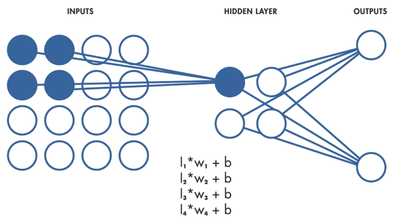

Figure 4: CNN.

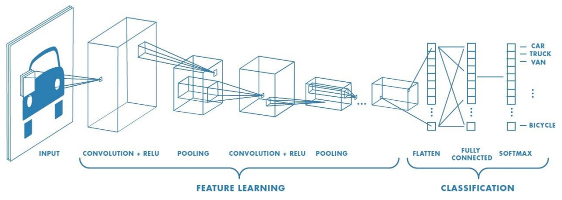

Figure 5: Reduction of dimensionality at CNN with pooling.

Now, using this model (*CNN*) we have two very similar working programs, one with *PyTorch* and the other using *tensorflow*. This is as result that at the beginning of the pair project we gave each other a week in order to study the topic and come up with ideas of how to solve it. After that week we both had a similar idea, use a CNN, although had implemented it using a different python library. We were happy with the output so we continued until obtaining working programs.

Bellow, both models are described in a concise manner explaining and referencing the different functions and variables in order to make them understandable and easier to follow.

3.2 CNN model using PyTorch.

The PyTorch CNN model code can be found, with more details and instructions in Github together with an approach that replaced it, Feature Matching (can be found in section 4 Procedures: Feature Matching.

Since both CNN model essentially work the same, I wanted to try another approach to see the comparison, but the original work on the PyTorch CNN for this problem still exists, just requires refinement and some ”cleaning up”. With the goal of avoiding redundant information this model is not discussed in more detail here.

3.3 CNN model using TensorFlow.

How to run the model.

First and foremost, to run the model simply call the '*mandelbrot_CNN_tensorflow.py*' file and call the .png file of the fragment of the mandelbrot, as:

```
    > python  ./mandelbrot_CNN_tensorflow.py  {file_name}.png
```

This will generate an output .gif file in the same directory with the zoom-in over the coordinates determined by the search model from the starting viewpoint of the Mandelbrot set with the name "zoom_animation.gif".

Additionally, aside from the usual numpy or matplotlib libraries, other ones used are:

1. tensorflow
2. imageio
3. argparse
4. PIL
5. glob

Training dataset.

'Mandelbrot_CNN.py' is the file where the model is stored, not trained, and used to find the coordinates of our input image. Both the model as well as the set of images used to train it are defined in 'Mandelbrot_final.ipynb'. And from this last one we start.

Within generating the program to recognize the location of the given fragment of the Mandelbrot set, we ought to generate a set of images from where out model can be trained. In order to generate the images, we define the Mandelbrot set and its characteristics in 'Mandelbrot'.

```
def mandelbrot(h, w, max_iter, x_min, x_max, y_min, y_max):
y, x = np.ogrid[y_min:y_max:h*1j, x_min:x_max:w*1j]
c = x + y*1j
z = c
divtime = max_iter + np.zeros(z.shape, dtype=int)

for i in range(max_iter):
    z = z**2 + c
    diverge = z * np.conj(z) > 2**2
    div_now = diverge & (divtime == max_iter)
    divtime[div_now] = i
    z[diverge] = 2

return divtime
```

Listing 1: Definition of Mandelbrot.

Here, the dimensions of our images ,with height and width parameters (**h, b**), as well as the complex number and the value of $f(z=0) = z^2 + c$ in the first iteration are defined. Then, a loop over a maximum number of iterations, 100, defines the number of iterations at each point.

This function, alongside with *'save_with_colormap'*, are used consequently in *'generate_data'* to create the images to train our model.

```
    def generate_data(output_dir, num_samples=number_samples,
    img_size=(image_size, image_size), colormap='magma'):
    os.makedirs(output_dir, exist_ok=True)
    for i in range(num_samples):
        # Randomize bounding box within the Mandelbrot set domain
        x_min = np.random.uniform(-2, -0.8)
        x_max = np.random.uniform(x_min + 0.1, x_min + 0.6)
        y_min = np.random.uniform(-1.4, 1.4 - (x_max - x_min))
        y_max = y_min + (x_max - x_min)

        # Generate Mandelbrot set data
        mandel = mandelbrot(img_size[1], img_size[0], max_iter=100,
                            x_min=x_min, x_max=x_max, y_min=y_min,
    y_max=y_max)

        # Save the image with the specified colormap
        output_path = (os.path.join(output_dir, f"{i}_{x_min:.6f}_{
    x_max:.6f}_{y_min:.6f}_{y_max:.6f}.png")) # Name of the image.
        save_with_colormap(mandel, output_path, colormap=colormap)
```

Listing 2: Generation of images for training data.

Here the most important part is the fact that images are generated randomly but also ensure them to be small enough as to observe detailed patterns. That is why 'x_min' is focused on one area of the set, where most intricate patterns appear, and 'x_max' is not fully random, but close enough to the minimum parameter to enhance small sections of the set. Then, the distance between 'y_min' and 'y_max' is the same as that of x values, so the complete region remains a square. This coordinates are then used to generate its respective Mandelbrot section, and the image name file is engraved with these coordinates. Later the name of the file will take part on the training of the data. Overall, the data is generated to fit the boundaries of the Mandelbrot set domain: [-2, 1.5] for x and [-1.4, 1.4] for x.

Lastly, a set of 1000 images is generated for the training data. This number has been deliberate. Initially I ran the model using only 30 images of size 64*64 and the dataset took around a minute to generate. Improving the quality of the image to 480*480, and the number of images to 1000, it takes around 30 minutes to generate the dataset.

```
    # Generate 1000 samples with magma colormap
generate_data("mandelbrot_training_highQuality", num_samples=
    number_samples, img_size=(image_size, image_size), colormap='
    magma')
```

Listing 3: Generate training data

Model.

Once the training data is generated, available in "mandelbrot_training_highQuality" directory, we access it and use it to train our model. In order to access the data as well as its coordinates, we use the function *"load_data"*.

```
    def load_data(data_dir, img_size=(image_size, image_size)):
    images, labels = [], []
    for file in glob.glob(f"{data_dir}/*.png"):
        base_name = os.path.basename(file).replace(".png", "")
        parts = base_name.split("_")  # Split the name by "_"
        parts.pop(0) # Eliminate index
        x_min = float(parts[0])
        x_max = float(parts[1])
        y_min = float(parts[2])
        y_max = float(parts[3])

        img = Image.open(file).convert('L')  # Convert to grayscale
        img = np.array(img.resize(img_size)) / 255.0
        img = img.reshape(img_size[0], img_size[1], 1)  # Reshape
    to (480, 480, 1)

        images.append(img)
        labels.append([x_min, x_max, y_min, y_max])

    return np.array(images), np.array(labels)
```

Listing 4: Generate training data

Remembering the function where the images are generated, its important to remember that the coordinates of each of the images is saved in the .png name file as (index _ x_min _ x_max _ y_min _ y_max.png). Hence, each of these parameters are stored in as labels alongside an array of images. Once the dataset can be accessed, we proceed with the description and the training of the model. The model has been defined using some of the steps described in the introductory part of CNN with activation and pooling steps, in different layers. The model that I initially had defined obtained decent results but tended to overfit (it didn't had dropouts included). With the help of Kristijonas, introducing dropouts and increasing the number of epochs the coordination prediction accuracy are considerably low while the training metric, Loss and MAE, are higher than the ones defined initially. We'll get to the analysis of these results a bit later on.

```
    # Load training data
X_train, y_train = load_data("mandelbrot_data_highQuality")
# Build the CNN model
model = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(
    image_size, image_size, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.1),  # Add dropout

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.1),  # Add dropout

    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Dropout(0.1),  # Add dropout

    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dropout(0.3),  # Add dropout
    layers.Dense(4)  # Output layer
])
model.compile(optimizer=Adam(learning_rate=0.001, loss='mse',
    metrics=['mae'])
# Train the model
# epochs: Number of times that goes through the dataset.
# batch_size = number of images processed before updating weights.
# Validation split = % of data used for validation, not training.
model.fit(X_train, y_train, epochs=10, batch_size=32,
    validation_split=0.2)

model.save("mandelbrot_CNN_model.keras")
```

Listing 5: Train the model.

Three different layers are defined with activation, pooling and dropout in each of them. In the activation the number of filters is set (32 or 64) where each filter will focus on a different feature with sixes 3*3 (pixels). With pooling the spatial dimensions are reduced by 2. The dropout in each layer specifies the number of neurons that will randomly be disabled. This introduces a factor of randomness that avoids making the network too reliant on some features. Lastly, the dimensions are 'flattened' to a single dimension with 64 neurons.

The model is then stored, "*.keras* file", and used later in the .py to run the program.

Zoom in and application of the model.

Once the model has been defined and trained with our dataset, the last step is to define a function that generates a zoom in animation that advances progressively to the coordinates predicted by the model. This animated imaged is saved in a file "*zoom_animation.gif*" in the same directory.

```
    def zoom_in(snippet, model, full_width=800, full_height=600,
    zoom_steps=30, gif_filename="zoom_animation.gif"):
    # Predict bounding box
    snippet_resized = np.expand_dims(np.array(snippet.resize((480,
    480))) / 255.0, axis=(0, -1))
    x_min, x_max, y_min, y_max = model.predict(snippet_resized)[0]
    frames = []
    # Generate zoom-in animation frames
    for step in range(zoom_steps):
        interp_x_min=x_min + (-2 - x_min)*(1 - step / zoom_steps)
        interp_x_max=x_max + (0.8 - x_max)*(1 - step / zoom_steps)
        interp_y_min=y_min + (-1.4 - y_min)*(1 - step / zoom_steps)
        interp_y_max=y_max + (1.4 - y_max)*(1 - step / zoom_steps)
        mandel_zoom = mandelbrot(full_height, full_width, 100,
    interp_x_min, interp_x_max, interp_y_min, interp_y_max)
        # Plot the Mandelbrot set for the current step
```

```python
        fig, ax = plt.subplots(figsize=(12, 8))
        ax.imshow(mandel_zoom, cmap='magma', extent=[interp_x_min, interp_x_max, interp_y_min, interp_y_max])
        ax.set_title(f'Zoom Step {step + 1}/{zoom_steps}')
        ax.set_xlabel("Re")
        ax.set_ylabel("Im")
        plt.colorbar(ax.imshow(mandel_zoom, cmap='magma'), label='Iteration count')
        fig.canvas.draw()  # Draw the plot
        image_buf = np.frombuffer(fig.canvas.tostring_rgb(), dtype=np.uint8)
        image_buf = image_buf.reshape(fig.canvas.get_width_height()[::-1] + (3,))  # Convert to RGB shape
        frames.append(image_buf)  # Append the frame
        plt.close(fig)
    # Save all frames as a GIF
    imageio.mimsave(gif_filename, frames, duration=0.1)  # Time steps
    print(f"GIF saved as {gif_filename}")
```

Listing 6: Zoom animation.

The function has as input both the image of the fragment to search for, and the model. The other input parameters are already set for the dimensions and characteristics of the GIF. Then, the fragment, *snippet*, is resized for the characteristics upon which our model has been trained and the coordinates are predicted.

Once the model has been used to predict the parameters, a loop generates the Mandelbrot set for each frame of the steps of the zoom-in over the predicted area and overlaps them to create the frames. Finally the image is saved in GIF file with a standard name defined in the arguments of the function.

mandelbrot_CNN_tf.py

This last file encompasses some of the functions already described, needed to apply the model and obtain the output.

This code has around 100 lines of code only, so I will briefly go over the libraries that have been used (external libraries needed) and the summarized process it follows.

As described previously, to run the code for a given Mandelbrot fragment simply run it in the terminal alongside the image and, in case it is needed, install the libraries tensorflow, imageio, argparse and glob:

```
 > python  ./mandelbrot_CNN_tensorflow.py  {file_name}.png
```

```
pip install tensorflow
pip install imageio
pip install argparse
pip install glob
```

Mandelbrot_CNN_tf.py file is a very simple file. It simply requires the zoom function, that in turn requires of the mandelbrot function and the description of the model, in order to obtain the result and generate an output. The mandelbrot function is defined, same as described previously, as its needed for the zoom function, and the model is simply loaded from the directory,

```python
    from tensorflow.keras.models import load_model # type: ignore
model = load_model("mandelbrot_CNN_model_updated.keras")
```

Listing 7: Load the model trained previously.

Additionally, a function "*take_input*" is define to take whichever image is given by the user and automatically calls zoom_in function. This take_input is defined in main, so it is called from the terminal.

Results and analysis.

As mentioned previously when introducing the model, initially I created a model without the use of dropouts. Then, with the input of Kristijonas, the same model was devised but with a higher number of epochs, 30 instead of 10, and with dropouts at each layer. The results between these two models are presented and discussed. This later model will be referred to as '*updated*'.

First of all, the metrics that are obtained for each of the models. Here we can see the comparison between model Loss and Mae, as well as coordinate accuracy, for each of the models.

As can be seen, the training metrics are considerably higher in the updated model than the original. On the other hand, the coordinate accuracy values are better in the updated, with more uncertainty in the original model. Both models present oddities. The original model does not have an ordained decrease of the training metrics over the epochs as well as overfitting, while for the updated model has incredibily high loss and mae values for the first epoch (612 and 7 aprox.), although it quickly decreseases to more acceptable values from the second epoch. Nonetheless, even though the coordinate accuracy looks better in the updated model, its worse training metrics arouse questions.

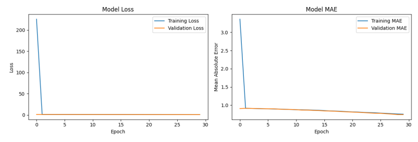

Figure 6: Updated model.

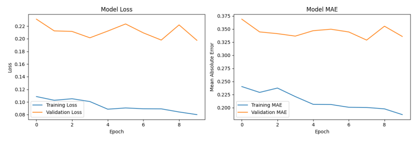

Figure 7: Model.

	Updated model	Model
Training Mae	0.7521	0.1867
Validation Mae	0.7393	0.3359
Training Loss	0.7624	0.0800
Validation Loss	0.7286	0.1974

Table 1: Final training and validation metrics.

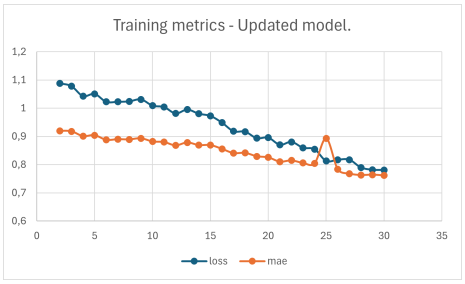

Figure 8: Training metrics excluding the first, odd, epoch.

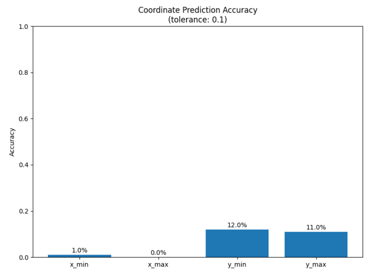

Figure 9: Updated model.

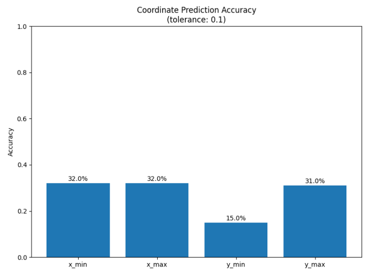

Figure 10: Model.

An interesting result we seem to obtain, is that low (64*64) models give better results. The image size is determined not by the quality of the training dataset, but by the image size defined in load_data function. For this, let's compare the outcome of a random snippet of the Mandelbrot set, and see which is the outcome from 3 different models: the original one and one from Kristijonas, at 64*64, and the updated model with 480*480. We think this might be because the low number of layers, only 3, not reducing enough the dimensions of the images with activation and pooling, doesn't make it efficient to identify small patterns of fragment images.

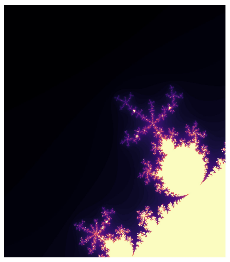

Figure 11: Random .png file to run the code.

4 Procedures: Feature Matching

This section includes the description of another approach used for this problem - feautre matching. The core idea of this method is to identify locations within the Mandelbrot set by comparing visual features between an input image and a database which includes reference images. This approach uses computer vision techniques, specifically the Scale-Invariant Feature Transform (SIFT), to detect and match distinctive points between images.

4.1 Technical Details

4.1.1 SIFT (Scale-Invariant Feature Transform)

SIFT is used to detect features in Mandelbrot set images, more details on it can be found in the Used Tools section 4.2. For brief introduction the SIFT algorithm works in these main four steps:

1. Finding distinctive points at different scales (keypoints)
2. Refining their locations (keypoint localization)
3. Calculating their orientations
4. Creating unique descriptors for each point

4.1.2 Feature Matching Process

The feature matching happens in this order of steps:

1. Take SIFT features from both input and reference image
2. For each feature in the input image, find the two nearest neighbors in the reference image
3. Apply Lowe's ratio test to filter out only good matches:

   $$\frac{distance_{closest}}{distance_{second\_closest}} < threshold$$

   where $threshold = 0.7$ in our current implementation

4.1.3 Database Creation

As mentioned above, this approach needs a database of reference images to use. The creation of said database can be compared and maybe loosely called as this approach's version of "learning". The created database consists of:

- Mandelbrot set images at various coordinates and zoom levels
- Matching metadata (center coordinates and zoom level)

- Created SIFT features for every image

```python
def add_reference_image(self, image, coordinates):
    """Add an image and its coordinates to the reference database
    """
    # Convert image to grayscale if needed
    if len(image.shape) == 3:
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    else:
        gray = image

    # Extract features
    keypoints, descriptors = self.detector.detectAndCompute(gray,
    None)

    # Store features and coordinates if descriptors and keypoints
    exist for them
    if descriptors is not None and len(keypoints) > 0:
            self.features_db.append({
                'keypoints': keypoints,
                'descriptors': descriptors
            })
            self.coordinates_db.append(coordinates)
```

Listing 8: Database Creation

4.1.4 Location Recognition

For an input image In, we:

1. Extract SIFT features $F_{In} = \{f_1, f_2, ..., f_n\}$
2. For each reference image R_i:
   - Find matches M_i between F_{In} and F_{R_i}
3. After applying the filter test we select the reference image with the most good matches

4.1.5 Error Calculation

Error calculation used is very trivial. Position error is calculated using geometric distance:

$$E_{pos} = \sqrt{(x_{pred} - x_{actual})^2 + (y_{pred} - y_{actual})^2}$$

Zoom error is calculated seperately as absolute difference:

$$E_{zoom} = |zoom_{pred} - zoom_{actual}|$$

4.2 Used Tools

4.2.1 OpenCV (cv2)

OpenCV is our main computer vision library. This approach utilizes two main components for feature extraction and feature matching.

1. SIFT (Scale-Invariant Feature Transform) SIFT operates in two main steps:

1. **Keypoint Detection**:
   - Identifies distinctive points in the image
   - Each keypoint has:
     - Location (x, y)
     - Scale (size of the region)
     - Orientation (dominant gradient direction)
2. **Descriptor Computation**:
   - For each keypoint, computes a 128-dimensional descriptor
   - Descriptor captures gradient information around the keypoint
   - These descriptors are what get matched between images

The combination of keypoint location and its descriptor makes SIFT invariant to these changes:

- Scale changes
- Image rotation
- Brightness changes
- Viewpoint/perspective changes

```python
class MandelbrotFeatureMatcher:
    def __init__(self, use_sift=True):
        # Initialize SIFT detector
        self.detector = cv2.SIFT_create()

        # Initialize FLANN matcher
        FLANN_INDEX_KDTREE = 1
        index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
        search_params = dict(checks=50)
        self.matcher = cv2.FlannBasedMatcher(index_params,
    search_params)
```

Listing 9: SIFT Initialization.

2. FLANN-Based Matcher We use FLANN (Fast Library for Approximate Nearest Neighbors) for feature matching:

- **Algorithm**: FLANN with kd-tree index
  - Parameters used:
    - `FLANN_INDEX_KDTREE = 1`
    - `trees = 5`

        * `checks = 50`
    - Builds a k-dimensional tree for fast nearest neighbor search

- **Matching Process**:
    1. For each input feature, find k=2 nearest neighbors
    2. Apply Lowe's ratio test:

$$ratio = \frac{distance_{best\_match}}{distance_{second\_best}} < 0.7$$

    3. If ratio < threshold - it's a good match

It is essentially an efficient search algorithm for nearest neighbor search problems, it:

1. Creates multiple kd-trees to organize points for our SIFT descriptors
2. For each query point:
    - Searches these trees in parallel
    - Checks a specified number of leaf nodes (in our case, 50)
    - Returns the closest matches found

```python
def find_location(self, query_image):
    # Convert to grayscale
    gray = cv2.cvtColor(query_image, cv2.COLOR_BGR2GRAY)

    # Detect keypoints and compute descriptors
    keypoints, descriptors = self.detector.detectAndCompute(gray,
    None)

    # Find matches with database
    best_matches = []
    for db_features in self.features_db:
        matches = self.matcher.knnMatch(
            descriptors,
            db_features['descriptors'],
            k=2
        )

        # Apply ratio test
        good_matches = []
        for m, n in matches:
            if m.distance < 0.7 * n.distance:
                good_matches.append(m)

        best_matches.append(len(good_matches))
```

Listing 10: Feature Matching process

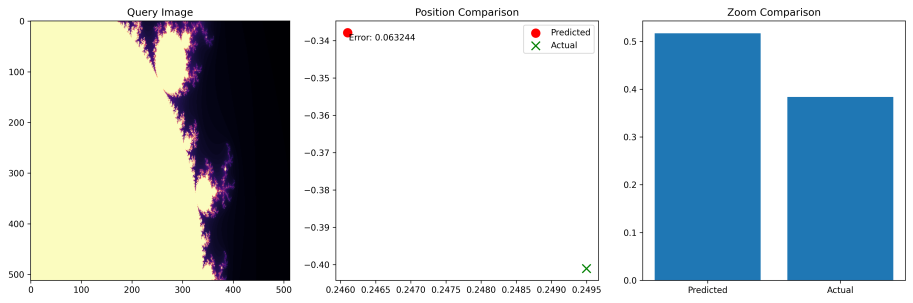

Figure 12: Example query image result from feature matching testing. It shows the original input image on the left, in the center - coordinate comparison between prediction and the actual center coordinate of inputm, on the right - a comparison between predicted and actual image zoom magnitude levels.

4.3 Results

This section includes result graphs from feature matching testing.

Test Set Results:

- Total test images: 200
- Successful matches: 186
- Failed matches: 14
- Success rate: 93.0%
- Average position error: 0.494662
- Average zoom error: 0.226827
- Average matches found: 87.7

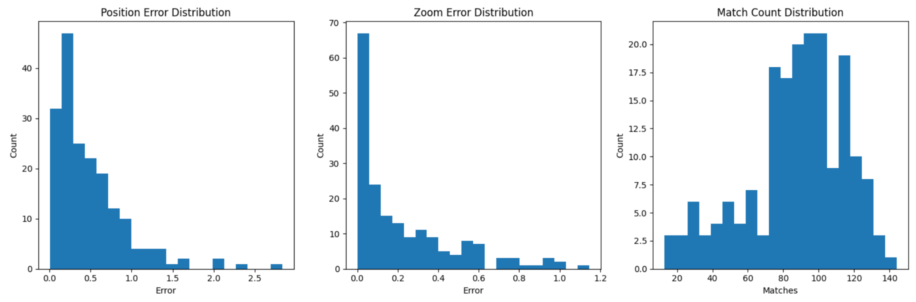

Figure 13: Distribution analysis of feature matching performance over 200 test images. On the left we have position error distribution showing histogram of different errors in coordinate prediction, in the center - zoom error distribution showing the accuracy of zoom level predictions, and on the right match count distribution displaying the number of successful SIFT feature matches found per image.