PANORAMA LAB

AI-Enhanced Image Stitching and Edge Detection

By: Azza Eid

17 Oct 2024

Computer Vision Bootcamp Final Project with **GSG**

Overview

Discover the Panorama Lab, a project that brings the excitement of image processing to life! This interactive platform lets you adjust image parameters and instantly see the magic of real-time transformations.

The interface, crafted with care using PyQt and Qt Designer, provides a smooth and engaging user experience. Behind the scenes, the powerful OpenCV library drives complex image processing tasks, while YOLO (You Only Look Once) handles human detection. Whether it's stitching multiple images or exploring various edge detection techniques, users are invited to experiment in an interactive and intuitive environment.

Through Panorama Lab, I hope to share the excitement I find in exploring the dynamic world of image processing, bringing its magic to life for everyone!

Core Technologies

1. Frameworks and Libraries:
   - **PyQt5**: User interface development
   - **OpenCV**: Image processing and stitching
   - **YOLO**: AI-based detection
2. Development Tools:
   - **Qt Designer**: UI layout design
   - **Git**: Version control
   - **Virtual Environment**: Project isolation

Features and Methodology

1. **User Interface Features:**

    - The main layout consists of a side menu and a stacked widget, enabling easy navigation between the three pages: stitching, edge detection, and human detection. This design ensures a responsive and user-friendly experience.
    - On the first page, users can select multiple images, which are then displayed in the designated section of the interface.

2. **Image Processing Capabilities:**

    - **Image stitching**: This feature takes the images selected by the user to create a single panoramic image, utilizing feature matching techniques with OpenCV.
    - **Multiple edge detection methods**

        1. **Canny Edge Detection**
        This feature allows users to visualize the edges in their panoramic image using the Canny edge detection algorithm.
        The user interface includes a slider for selecting threshold values, with the resulting edge detection displayed directly on the image.

        2. **Difference of Gaussians (DoG) edge detection**
        This feature allows users to enhance the edges in their panoramic image using the Difference of Gaussians (DoG) edge detection method, followed by a morphological operation.
        The user interface provides a slider for adjusting the kernel size of the morphological operation, a combo box for selecting the type of operation (Opening or Closing), and another combo box for choosing the shape of the kernel (e.g., rectangular, elliptical, or cross). Users can see the results of selecting different parameters in real-time, enabling an interactive and responsive image processing experience.

    - **AI-based human detection**
    This feature utilizes an AI-based object detection model, specifically YOLOv11n, to identify human figures within the stitched image. The model processes the image in real-time, filtering and displaying only those detections with a confidence level above 50%, limiting detection to class 0, which corresponds to humans. YOLO (You Only Look Once) works by dividing the image into a grid and predicting bounding boxes and class probabilities for each grid cell, allowing for efficient and accurate real-time detection.
    Additionally, a slider in the user interface enables users to adjust the confidence threshold dynamically, updating the image and displaying the count of detected humans accordingly. This interactive approach enhances the user experience by providing immediate feedback based on their selected confidence level.

3. **Real-Time Preview**
   In the image processing sections, both edge detection using Canny and DoG, as well as the human detection feature, offer a real-time preview capability. Users can utilize their computer's camera to see live results, making this experience not only interactive but also highly enjoyable. This feature represents one of the most engaging aspects of image processing and computer vision, allowing users to witness real-time transformations.

System Architecture

The project follows the **Model-View-Controller (MVC)** design pattern to ensure a clear separation of concerns between data processing, user interface, and application logic. The architecture is organized as follows:

- **Models**: This component handles the core logic for image processing and common functionalities. It includes:
    - `ImageProcessor`: handles the stitching process to create panoramic images and includes various operations that can be applied to these images, such as Canny edge detection, Difference of Gaussian (DoG), and human detection, all specifically tailored for panoramic images. It utilizes cv2 for image processing and contains functions to retrieve images and convert them into a format that can be returned to the interface and displayed in QLabel widgets.
    - `commonFunctionality:` Contains shared functions such as YOLO for object detection and Difference of Gaussian (DoG) for edge detection. These functions can be accessed by both the `ImageProcessor` and live_`previews` models.
    - `live_previews:` Includes functions that interact with the user's camera for real-time previews, enhancing the user experience.
    - `imageType:` An enum used to prevent typing errors by defining image types such as panoramic images and Canny edge-detected images.
- **Views**: The user interface is managed within this component, found in the views folder. It contains:
    - The main application UI, which is a Python file generated from a .ui file designed in Qt Designer.
    - Dialog windows for specific tasks and interactions.
- **Controller:** The controller serves as the intermediary between the user interface and the image processing logic. It captures user interactions and links them to the appropriate operations in the models. Additionally, it handles tasks such as displaying images in QLabel at appropriate sizes, managing actions for navigating between pages, and loading images into the interface.

```
├── models/
│   ├── __init__.py
│   ├── imageProcessor.py         # Processes panoramic images, returns responses
│   ├── commonFunctionality.py    # Shared functions like YOLO, DoG
│   ├── live_previews.py          # Connects to the user's camera
│   ├── imageType.py              # Enum for image types (e.g., panoramic, Canny)
│
├── views/
│   ├── __init__.py
│   ├── MainWindow.py             # Main application interface
│   ├── resources_rc.py           # Binary data for assets used UI (e.g., Images)
│   ├── dialogs.py                # Dialog windows for tasks and interactions
│
├── controller.py                 # Links UI events to image operations in models
├── main.py                       # Run the App
```

Data Flow

User Input Flow:

```
User->>View: Interacts with UI
View->>Controller: Sends user action
Controller->>Model: Requests data update
Model-->>Controller: Returns processed data
Controller-->>View: Updates UI
```

User Interface Design

The user interface consists of a single main window with three pages and side menu for enabling easy navigation between them:

Page I: Stitching Page

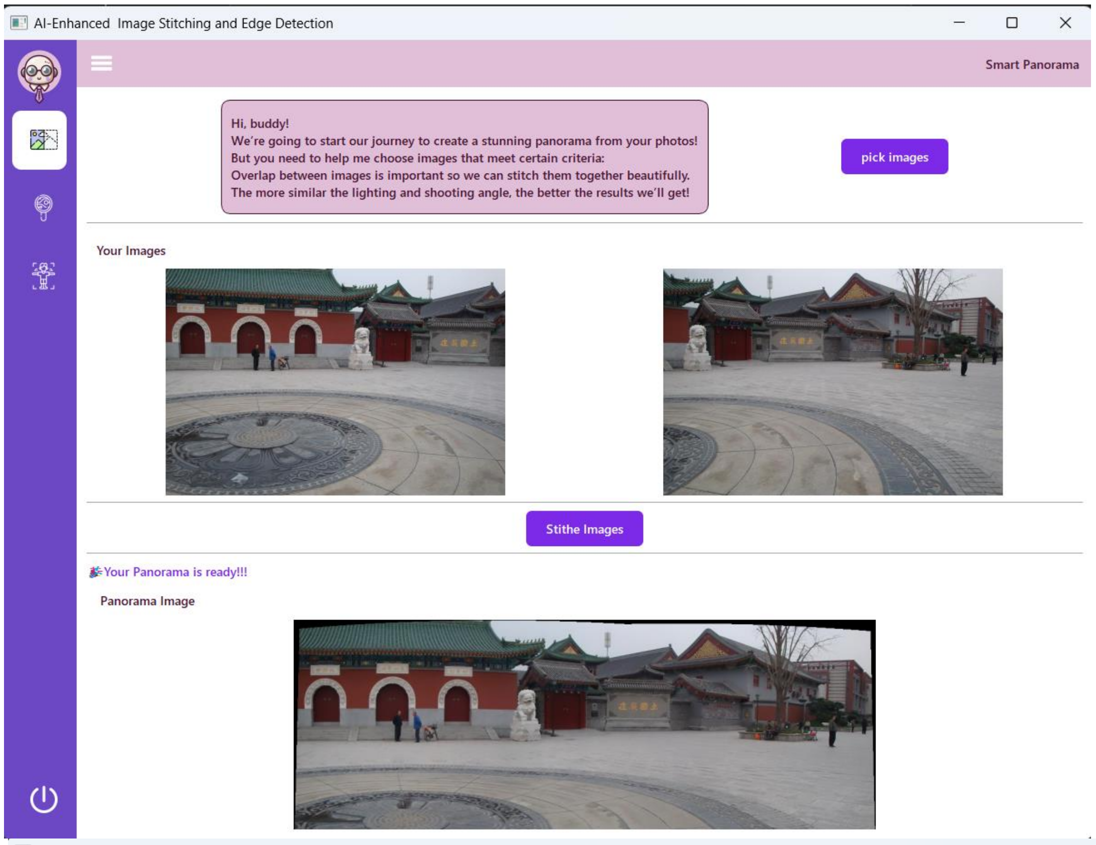
- **Image Selection Area**: Allows users to choose images for stitching.
- **Selected Images Preview**: Displays the images selected by the user.
- **Panorama Preview**: Shows the resulting panoramic image.
- **Dynamic Labels**: Provides error messages or feedback for the user.

Page II: Edge Detection Page

- **QTabWidget with Three Tabs**:
    1. **Canny Edge Detection**:
        - Result of Canny preview.
        - **Info Section**: Label to display used thresholds.
        - **Processing Controls**: Slider to change threshold and a button for live preview.
    2. **Difference of Gaussians (DoG) Edge Detection**:
        - Result of DoG preview.
        - **Processing Controls**:
            - Two QComboBox for selecting the type of operation (Opening or Closing) and the shape of the kernel.
            - Slider to adjust the kernel size for the morphological operation.
            - QPushButton for live preview.
    3. **Show Results of All Previous Processes**:
        - Displays results from the earlier processes.

Page III: Human Detection Page

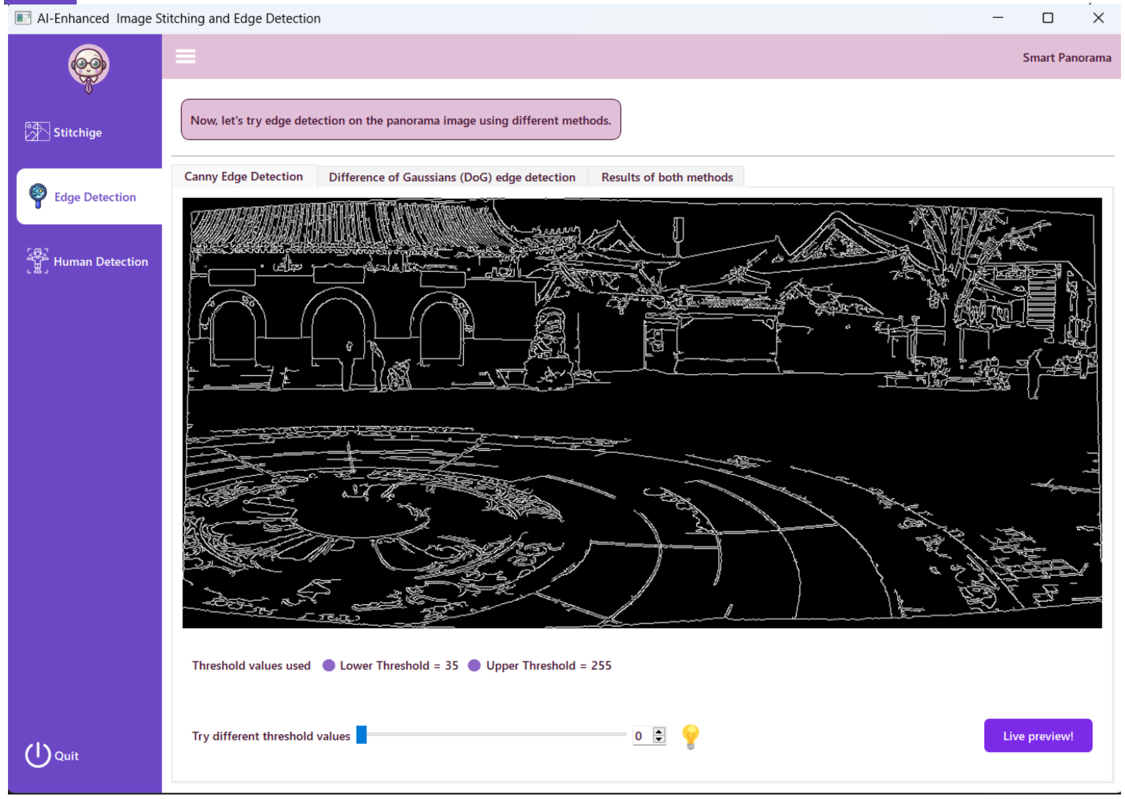
- **Label**: Shows the count of detected people.
- **Result of Human Detection Preview**: Displays the outcome of the human detection process.
- **Processing Controls**: Slider to change confidence level and a QPushButton for live preview.


AI-Enhanced Image Stitching and Edge Detection

Smart Panorama

Hi, buddy!
We're going to start our journey to create a stunning panorama from your photos!
But you need to help me choose images that meet certain criteria:
Overlap between images is important so we can stitch them together beautifully.
The more similar the lighting and shooting angle, the better the results we'll get!

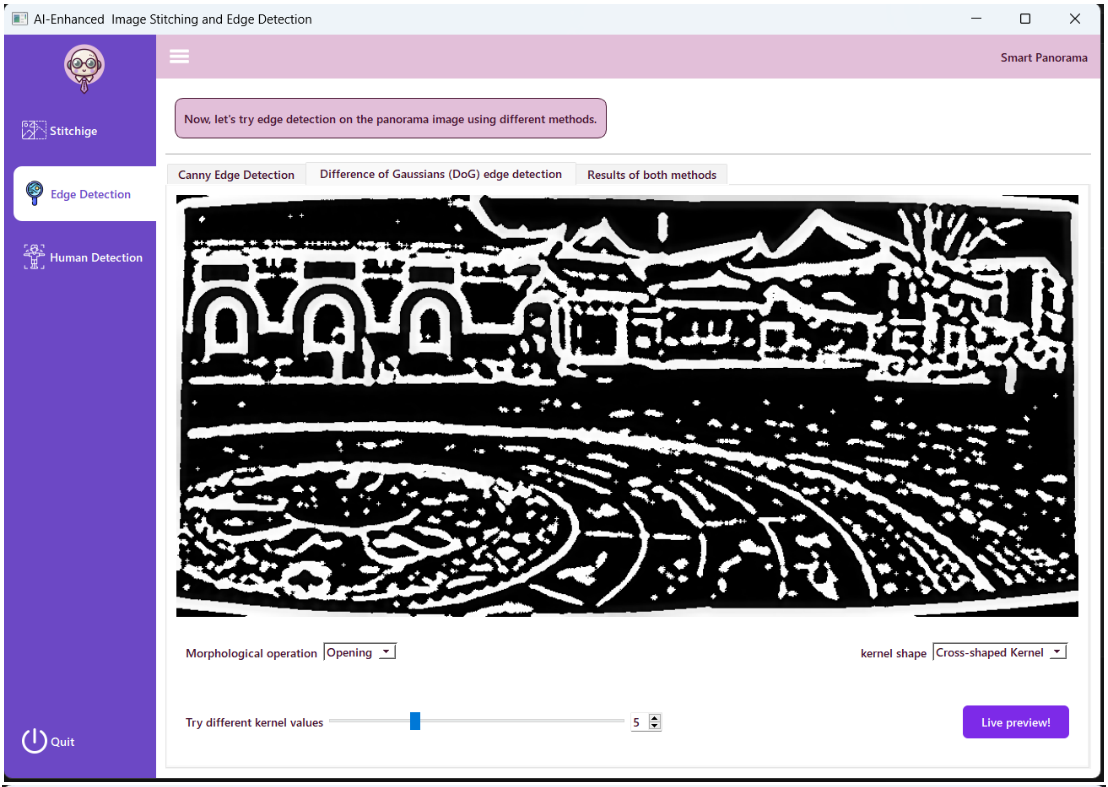
🎉Your Panorama is ready!!!


AI-Enhanced Image Stitching and Edge Detection

Now, let's try edge detection on the panorama image using different methods.

Threshold values used ● Lower Threshold = 35 ● Upper Threshold = 255

Try different threshold values 0



AI-Enhanced Image Stitching and Edge Detection

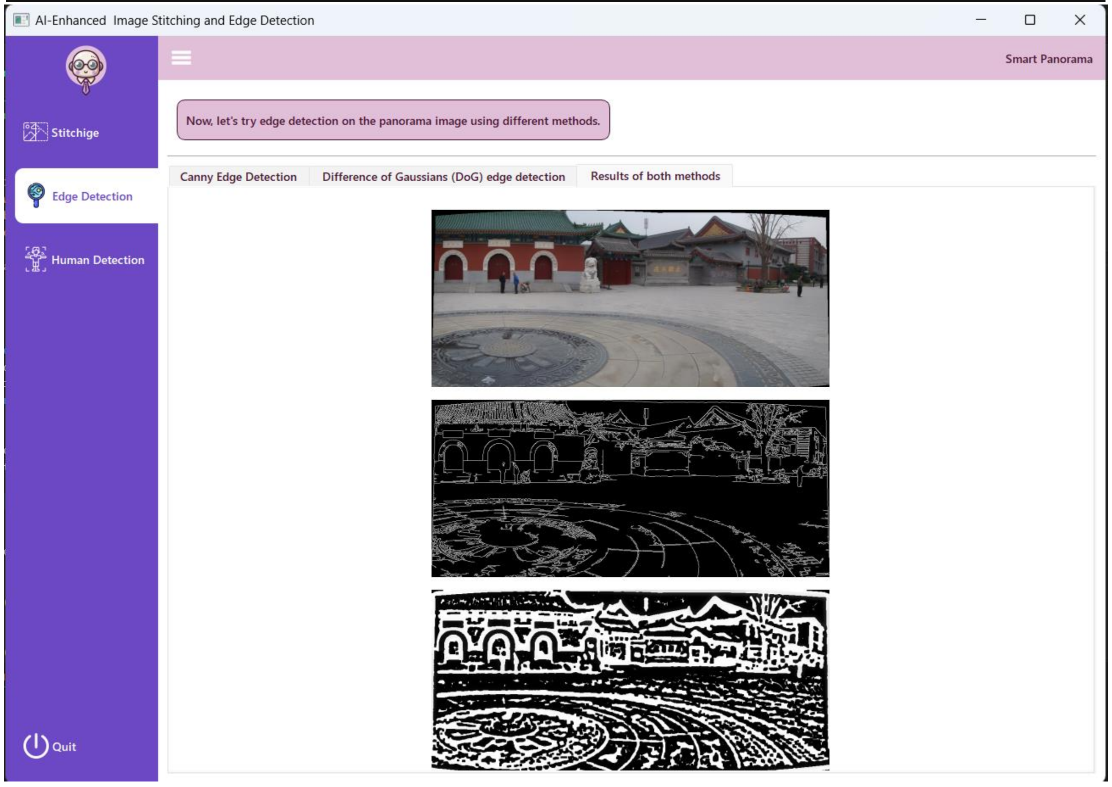
Smart Panorama

Now, let's try edge detection on the panorama image using different methods.

Morphological operation: Opening

kernel shape: Cross-shaped Kernel

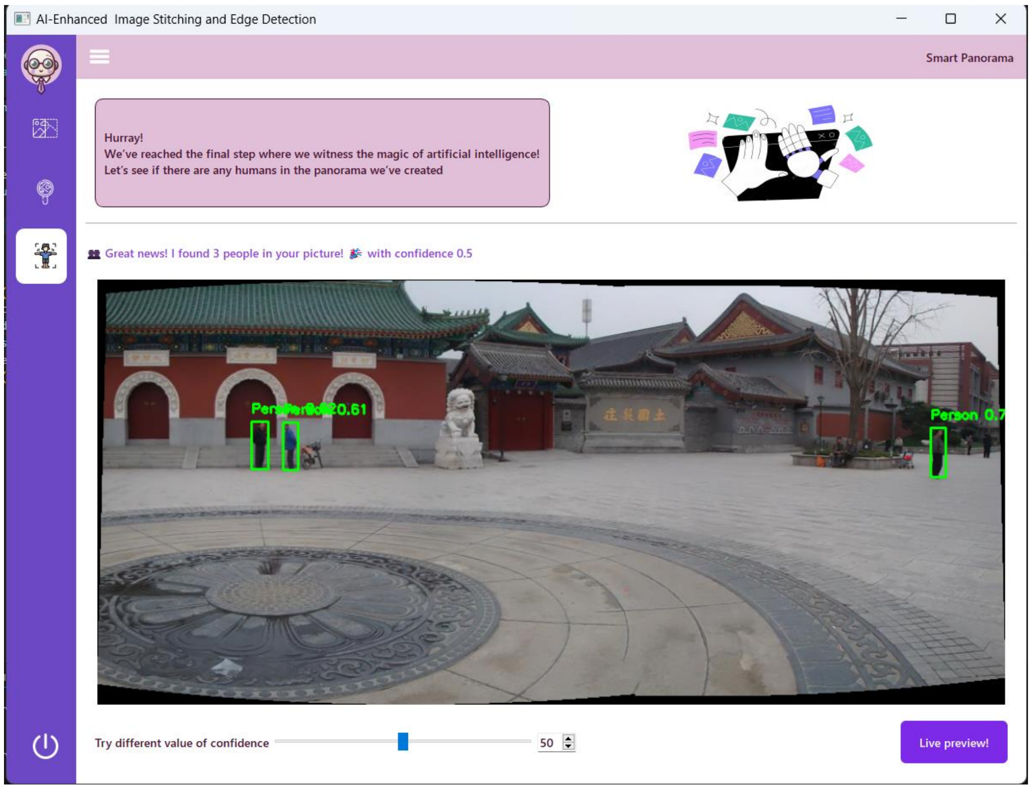
Try different kernel values: 5



AI-Enhanced Image Stitching and Edge Detection

Smart Panorama

Now, let's try edge detection on the panorama image using different methods.


AI-Enhanced Image Stitching and Edge Detection
Smart Panorama
Person 0.7
Try different value of confidence
50
Live preview!

Hurray!
We've reached the final step where we witness the magic of artificial intelligence!
Let's see if there are any humans in the panorama we've created

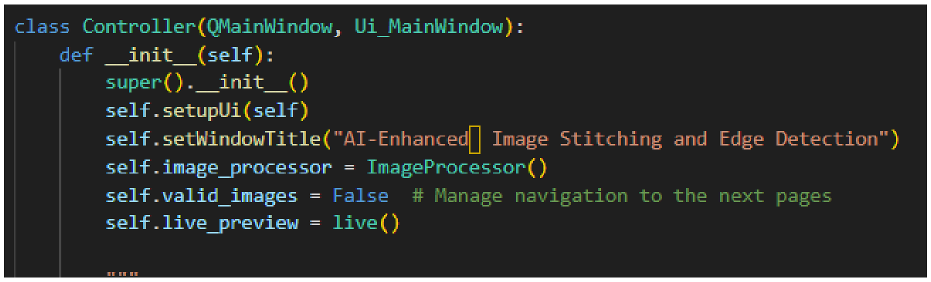
Great news! I found 3 people in your picture! with confidence 0.5

Key Challenges and Solutions

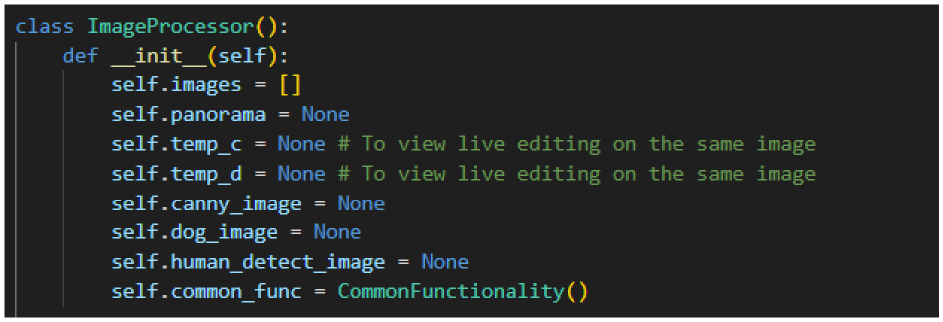
1. **Choice of PyQt for Desktop Application Development**: I chose PyQt for developing a desktop application due to its seamless integration with the designer, simplifying the development process. The library offers various tools for addressing various challenges, such as QImage for handling image formats, QFileDialog for file selection, and QPainter for custom drawing. Given the project's moderate size, a desktop application proves efficient without the complexities of a web application

2. **Selection of MVC Architecture**: After careful consideration of how to organize the interaction between the user interface and backend operations, I determined that the MVC architecture would streamline the process. This structure allows easy referencing of user interface actions and their corresponding functions, with `ImageProcessor` and `livePreview` included as properties within the controller class.

3. **Unified Model for Operations**: I created a single model, `ImageProcessor`, to handle all operations based on input images across different screens. This decision simplifies passing the same panoramic image between processing functions in the background. However, this approach led to code duplication between `ImageProcessor` and `livePreview`, prompting the introduction of a third model called `commonFunctionality`.

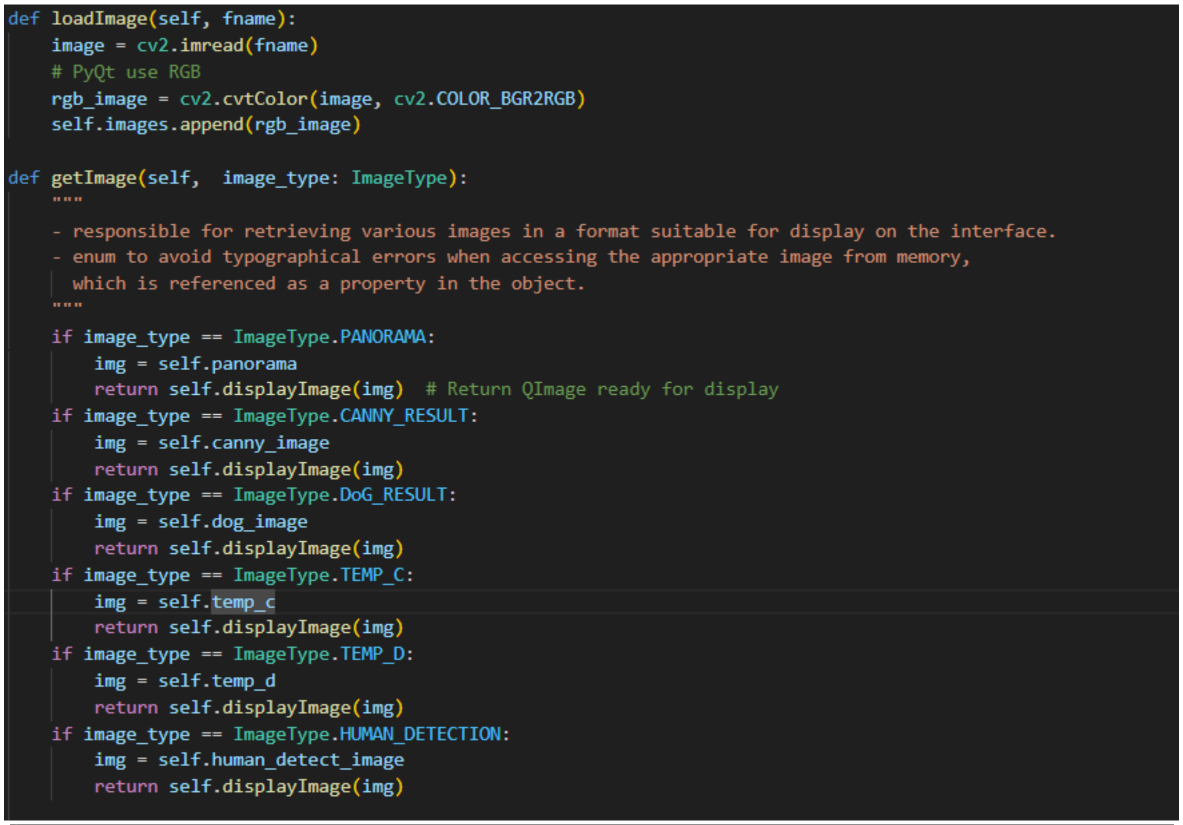


```
class Controller(QMainWindow, Ui_MainWindow):
    def __init__(self):
        super().__init__()
        self.setupUi(self)
        self.setWindowTitle("AI-Enhanced Image Stitching and Edge Detection")
        self.image_processor = ImageProcessor()
        self.valid_images = False  # Manage navigation to the next pages
        self.live_preview = live()

        """
```

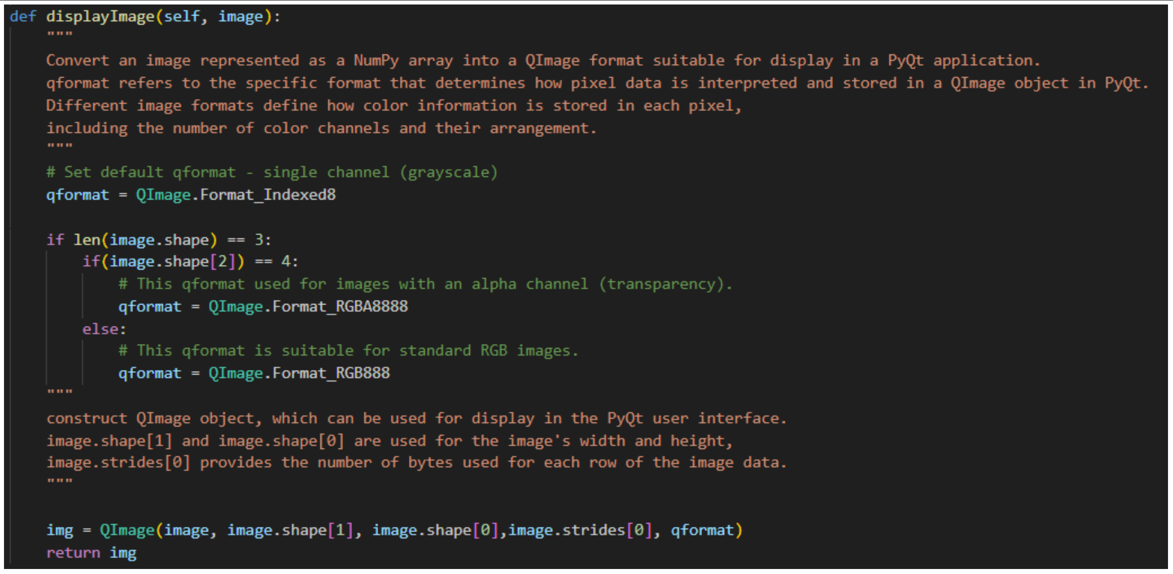


```
class ImageProcessor():
    def __init__(self):
        self.images = []
        self.panorama = None
        self.temp_c = None # To view live editing on the same image
        self.temp_d = None # To view live editing on the same image
        self.canny_image = None
        self.dog_image = None
        self.human_detect_image = None
        self.common_func = CommonFunctionality()
```

4. **Image Format Conversion**: Initially, I processed images in the background using formats produced by cv2, storing them in memory. To display these images in the interface and ensure compatibility with PyQt, I converted them to RGB format. I also developed a function to transform the images into a format acceptable for the user interface using QImage. To prevent typing errors when referencing stored images, I established an enum model categorizing the types of images in memory.

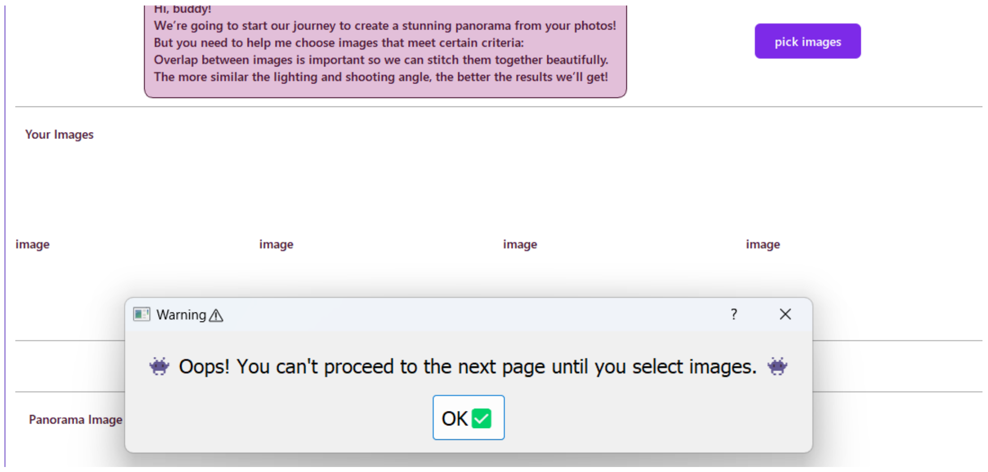


```python
def loadImage(self, fname):
    image = cv2.imread(fname)
    # PyQt use RGB
    rgb_image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    self.images.append(rgb_image)

def getImage(self,  image_type: ImageType):
    """
    - responsible for retrieving various images in a format suitable for display on the interface.
    - enum to avoid typographical errors when accessing the appropriate image from memory,
      which is referenced as a property in the object.
    """
    if image_type == ImageType.PANORAMA:
        img = self.panorama
        return self.displayImage(img)  # Return QImage ready for display
    if image_type == ImageType.CANNY_RESULT:
        img = self.canny_image
        return self.displayImage(img)
    if image_type == ImageType.DoG_RESULT:
        img = self.dog_image
        return self.displayImage(img)
    if image_type == ImageType.TEMP_C:
        img = self.temp_c
        return self.displayImage(img)
    if image_type == ImageType.TEMP_D:
        img = self.temp_d
        return self.displayImage(img)
    if image_type == ImageType.HUMAN_DETECTION:
        img = self.human_detect_image
        return self.displayImage(img)
```



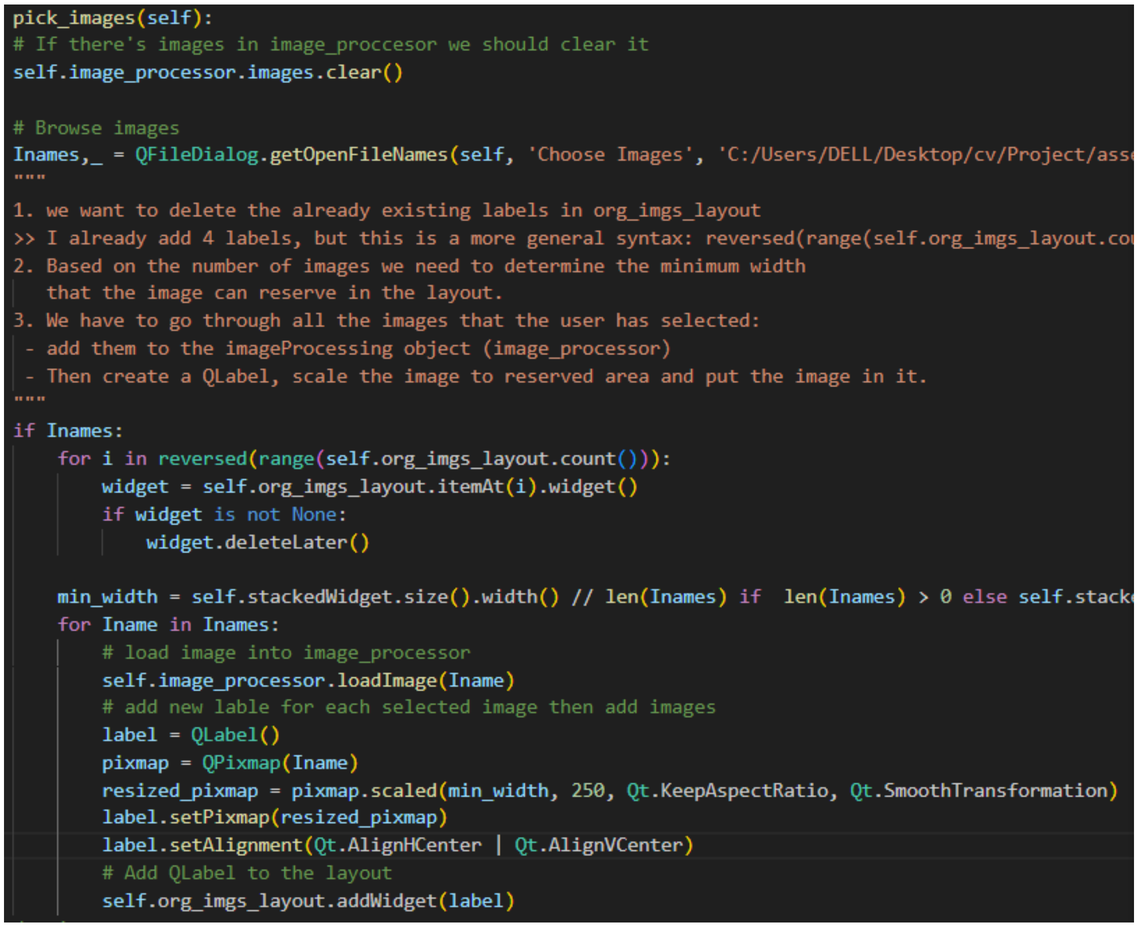
```python
def displayImage(self, image):
    """
    Convert an image represented as a NumPy array into a QImage format suitable for display in a PyQt application.
    qformat refers to the specific format that determines how pixel data is interpreted and stored in a QImage object in PyQt.
    Different image formats define how color information is stored in each pixel,
    including the number of color channels and their arrangement.
    """
    # Set default qformat - single channel (grayscale)
    qformat = QImage.Format_Indexed8

    if len(image.shape) == 3:
        if(image.shape[2]) == 4:
            # This qformat used for images with an alpha channel (transparency).
            qformat = QImage.Format_RGBA8888
        else:
            # This qformat is suitable for standard RGB images.
            qformat = QImage.Format_RGB888
    """
    construct QImage object, which can be used for display in the PyQt user interface.
    image.shape[1] and image.shape[0] are used for the image's width and height,
    image.strides[0] provides the number of bytes used for each row of the image data.
    """

    img = QImage(image, image.shape[1], image.shape[0],image.strides[0], qformat)
    return img
```

5. **User Experience Challenge**: A key challenge was to enhance the user experience by preventing users from navigating to the second and third pages without successfully completing the image stitching process and obtaining a panoramic image. Implementing this restriction ensures that users only proceed when the required operations are successfully executed, thereby maintaining the application's integrity and usability.


Hi, buddy!
We're going to start our journey to create a stunning panorama from your photos!
But you need to help me choose images that meet certain criteria:
Overlap between images is important so we can stitch them together beautifully.
The more similar the lighting and shooting angle, the better the results we'll get!

Oops! You can't proceed to the next page until you select images.

6. **Show selected images**: I utilized QFileDialog to allow users to select multiple images across various extensions. A notable challenge was managing the number of selected images, as the display mechanism required creating QPixmap objects to present the images in existing QLabel widgets. To overcome this, I designed a solution that involved clearing all pre-existing labels from the interface to free up space. Then dynamically generate new labels based on the number of images selected, ensuring a smooth and organized display of the user's choices.


```
pick_images(self):
# If there's images in image_proccesor we should clear it
self.image_processor.images.clear()

# Browse images
Inames,_ = QFileDialog.getOpenFileNames(self, 'Choose Images', 'C:/Users/DELL/Desktop/cv/Project/asse
"""
1. we want to delete the already existing labels in org_imgs_layout
>> I already add 4 labels, but this is a more general syntax: reversed(range(self.org_imgs_layout.co
2. Based on the number of images we need to determine the minimum width
   that the image can reserve in the layout.
3. We have to go through all the images that the user has selected:
 - add them to the imageProcessing object (image_processor)
 - Then create a QLabel, scale the image to reserved area and put the image in it.
"""
if Inames:
    for i in reversed(range(self.org_imgs_layout.count())):
        widget = self.org_imgs_layout.itemAt(i).widget()
        if widget is not None:
            widget.deleteLater()

    min_width = self.stackedWidget.size().width() // len(Inames) if  len(Inames) > 0 else self.stack
    for Iname in Inames:
        # load image into image_processor
        self.image_processor.loadImage(Iname)
        # add new lable for each selected image then add images
        label = QLabel()
        pixmap = QPixmap(Iname)
        resized_pixmap = pixmap.scaled(min_width, 250, Qt.KeepAspectRatio, Qt.SmoothTransformation)
        label.setPixmap(resized_pixmap)
        label.setAlignment(Qt.AlignHCenter | Qt.AlignVCenter)
        # Add QLabel to the layout
        self.org_imgs_layout.addWidget(label)
```

7. **Choice of YOLOv11n for Object Detection**: I opted for YOLOv11n as the object detection model due to its superior balance of speed and accuracy compared to the MobileNet SSD. While both models are designed for real-time applications, YOLOv11n offers faster inference times and improved accuracy in detecting objects in complex environments. This efficiency is particularly beneficial for applications requiring quick responses. Moreover, I tailored YOLOv11n specifically for human detection by focusing on class 0, which represents people. This customization enhances the model's ability to accurately identify humans while minimizing false positives for other object classes. I also studied the structure of the output returned by YOLO, which includes box coordinates, confidence levels, and class labels. Based on this structure, I developed an efficient method to filter out boxes corresponding to class 0 (humans) and used cv2 to draw these boxes on the detected frames. As a result, YOLOv11n proves to be the most suitable choice for the application's requirements, ensuring reliable and efficient human detection.
8. **Selecting Default Values for Edge Detection Algorithms**
   In the development of the image processing application, one of the key challenges was determining the appropriate default values for edge detection algorithms, specifically Canny and Difference of Gaussian (DoG). These algorithms are highly sensitive to parameter tuning, and selecting incorrect values can either overlook important edges or generate excessive noise, leading to poor-quality edge detection.

   Dynamic Threshold Selection for Canny Edge Detection
   Choosing fixed thresholds for Canny edge detection can result in inconsistent outcomes across different images. The challenge was to provide both an adaptive default and user-adjustable thresholds. To address this, I implemented dynamic thresholding based on the image's median pixel intensity.
   When no user input is provided, the algorithm automatically sets the lower threshold to 30% below the median and the upper threshold to three times the lower threshold. If the user adjusts the threshold via a slider, the algorithm recalculates accordingly, ensuring flexible yet effective edge detection across varying images.

   Formula:

   ```
   lower_threshold = max(median - median * 0.3, 0)
   upper_threshold = min(lower_threshold * 3, 255)
   ```

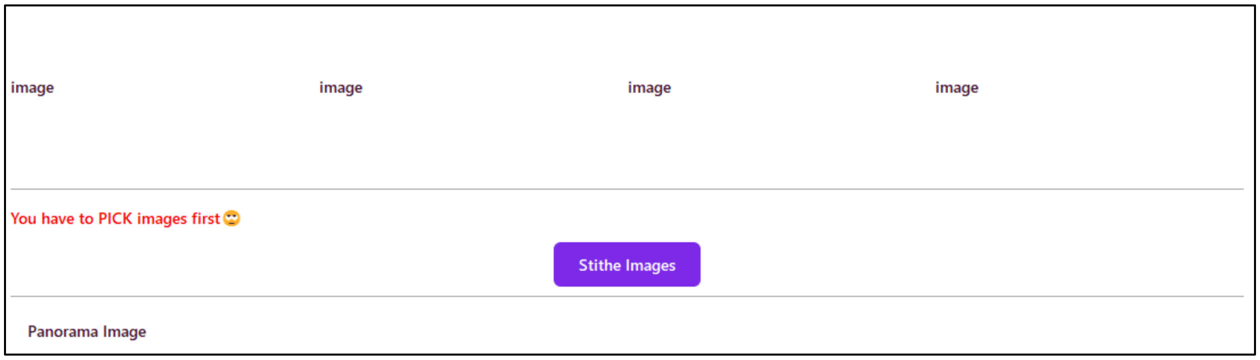
   User-Specified Thresholding
   If the user provides a lower threshold via the slider, the algorithm calculates the upper threshold as three times this value, in line with OpenCV's recommendations. This approach offers users flexibility while ensuring optimal edge detection.

Selecting Parameters for Difference of Gaussians (DoG) Edge Detection
In the apply_DoG function, I converted the input image to grayscale and then applied Gaussian blurring with two different kernel sizes. The first Gaussian kernel was set to (19, 19) with a standard deviation of 3, while the second was set to (31, 31) with a standard deviation of 5. This combination aimed to enhance edge detection while minimizing noise.
Another critical challenge was determining the parameters for the morphological operations that follow DoG processing. For this implementation, I selected a kernel size of 5 for the morphological operation and opted for the "closing" type. This choice was made to effectively close small gaps in the detected edges and create a more solid structure for further analysis.
By carefully selecting these parameters, I aimed to achieve a balance between sensitivity and specificity in edge detection.
To ensure that the kernel size for morphological operations is always an odd number, I configured the slider to start at 1 and increase in steps of 2. This guarantees that users can only select odd-sized kernels, which are crucial for effective morphological processing.

Error Handling and User Feedback

1- No Images for Stitching:

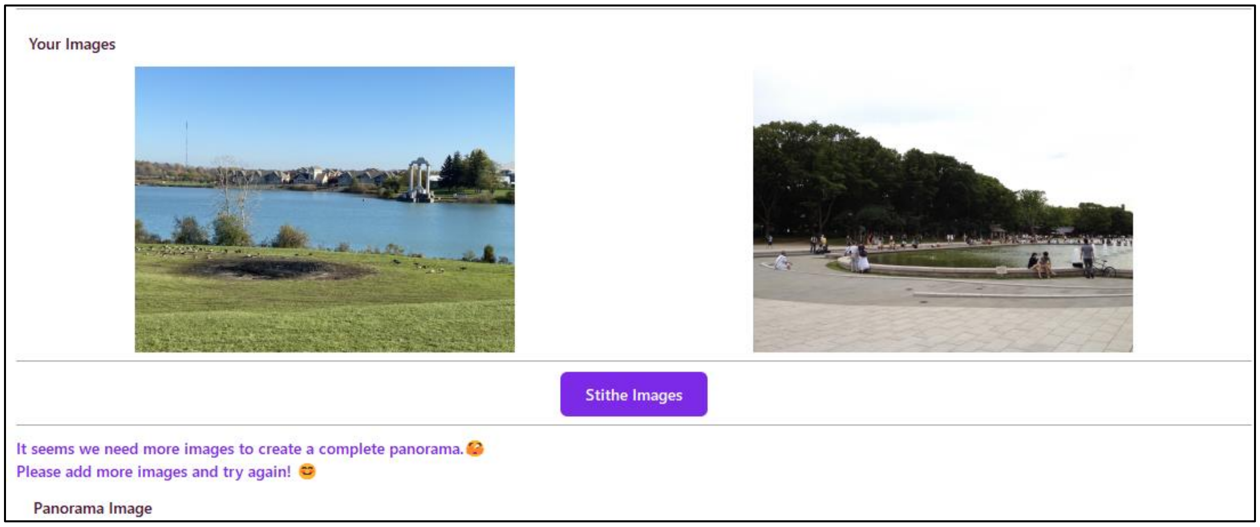
If the user attempts to execute the stitching operation without selecting any images, a message should be displayed indicating that the operation cannot be performed without images.



You have to PICK images first🙄

2- Stitching Operation Failed:

If the stitching process fails for any reason, the user should be informed that the operation was unsuccessful.



It seems we need more images to create a complete panorama.🥵
Please add more images and try again! 😊

Future Ideas and Expansions

1. **Feature Matching Visualization**:
   - Implement a button that allows users to visualize how features match between images before stitching. This could enhance user understanding of the stitching process and help in selecting better image pairs.
2. **Advanced Image Processing Techniques**:
   - Explore incorporating additional image processing techniques, such as histogram equalization, to enhance image quality before stitching or edge detection.

Addressing UI Delays in PyQt with Background Processing (In progress)

[problem] In the desktop application, a significant issue was encountered where the PyQt user interface (UI) would not update promptly during image processing tasks. Specifically, after stitching images to generate a panorama, the UI would delay displaying the panorama **until all subsequent tasks**—such as Canny edge detection, Difference of Gaussian (DoG) filtering, and human detection—were completed. This created a lag in the user experience, as the panorama image and other results were displayed simultaneously after all tasks had finished, making the application appear unresponsive during processing.

[Reason] This behavior occurs because PyQt operates by managing both the UI and processing tasks in the main thread. When heavy tasks are executed in this thread, they block the UI from updating until the processing is complete. While the initial attempt to resolve this issue involved separating the image processing functions and running them asynchronously using ThreadPoolExecutor, the UI still remained tied to the main thread. As a result, the panorama image could not be displayed immediately after stitching.

To address this, the proposed solution was to use QThread, which allows tasks to be processed in separate threads while keeping the main thread dedicated to UI updates. By moving image processing tasks into background threads with QThread, the UI remains responsive, allowing the panorama image to be displayed as soon as it is generated. Additionally, the results from other processing tasks (such as Canny and DoG) are sent back to the main thread using signals and slots, ensuring that the interface is updated progressively and safely without blocking the UI.

In conclusion, using QThread significantly improves the responsiveness of the application by allowing background tasks to run independently from the UI. This solution eliminates the delays caused by running all tasks in the main thread, providing a more fluid user experience.

Note: While this solution was proposed, it was not implemented in the current version of the application.

Project Scalability

The project is designed with scalability in mind, allowing for easy expansion and enhancement. By working with the UI file created in Qt Designer (in assets folder), new pages can be effortlessly added to the existing user interface. This modular approach enables developers to introduce additional functionalities without disrupting the existing structure.
Moreover, developers can easily install the necessary libraries for the environment by referencing the requirements.txt file. This file contains a list of all dependencies needed for the project, enabling a quick setup for any developer looking to contribute or extend the application.

To implement a new feature, the following steps can be taken:

1. **Modify the UI**: Simply open the UI file in Qt Designer, add the new page, and design its layout to accommodate the desired components and functionalities.
2. **Converting UI Design to Python Code**: To convert the UI design created in Qt Designer into a Python file that can be used in the project, the following command is executed in the terminal:
   ```
   pyuic5 -x assets/ui_design.ui -o Views/MainWindow.py
   ```
3. **Update the Controller**: In the controller file, new actions can be defined to manage interactions from the newly added page. This includes connecting buttons, sliders, and other UI elements to their corresponding functions.
4. **Extend the Models**: Finally, corresponding processing functions can be developed in the models. This allows the new page to perform specific operations, such as image processing or data analysis, while maintaining consistency with the existing functionalities.

This structured approach ensures that future developments can be integrated smoothly, enhancing the application's capabilities while preserving its overall integrity.

Acknowledgments

I would like to extend my thanks to **Dr. Izzeddin Teeti** for sharing his invaluable insights and experiences during the bootcamp. Additionally, I am grateful to co-Instructor **Rama Haseba** for her dedication and for always being there to address our questions. Your support has made a significant difference.