Introduction to Intelligent Systems
Assignment 2
Report

Aditya Prasad
2022036
BTech 2026
CSAI

Section A (Theoretical)

Q1)

Natural Language Processing (NLP) can be leveraged to enhance the performance of mobile search engines in several ways. Here are three techniques and strategies that can be employed to achieve this goal:

1. Sentiment analysis:
    - Mobile search engines can better identify the sentiment or emotion underlying user queries with the use of sentiment analysis. Search engines can learn more about a user's emotional state or intent by examining the sentiment of a query. Sentiment analysis, for instance, can recognise the negative sentiment in a user's search for "best smartphone for gaming, frustrated with lag," and deduce that the user is seeking for a solution to their gaming issues. As a result, the search engine can deliver more pertinent and sympathetic search results that take the user's particular problems into account.
    - Mobile search engines can improve the ranking and display of search results or recommendations by using sentiment analysis. User reviews, social media posts, and other types of user-generated content pertaining to goods, services, or subjects can all be subjected to sentiment analysis. Search engines can prioritise positive or highly rated results, filter out negative or low rated possibilities, and offer consumers more advantageous options by

recognising the mood conveyed in these sources. This raises the general standard of search outcomes and suggestions for mobile users.

- To monitor and control the sentiment surrounding brands, goods, or services, sentiment analysis can be used. Mobile search engines can examine reviews, comments, and social media posts from users to determine how people feel about particular companies. Search engines may give companies and enterprises timely insights into customer attitudes by monitoring sentiment trends. This enables them to proactively handle any negative sentiment or improve on positive sentiment. This aids in reputation management and empowers mobile users to decide wisely based on public perception of a specific business or item.
- Sentiment analysis can help mobile search engines provide users with more customised experiences. Search engines can better comprehend each user's preferences and feelings by examining the sentiment conveyed in user interactions like feedback, ratings, or inquiries. This enables the search engine to personalise and enhance the user experience by tailoring search results, recommendations, or advertising content to fit the user's sentiment. Search engines can adjust their tone or wording in responses to match the user's emotional state by taking the user's mood into account.

2. Topic modeling:
    - Mobile search engines that use topic modelling can better comprehend the primary themes or subjects found in documents like web pages, articles, or user-generated material. Search engines may recognise the underlying subjects and extract important keywords or phrases connected with each topic by analysing the text using topic modelling algorithms. This improves comprehension of the document's content and makes it possible for indexing, retrieval, and ranking of search results to be more precise.
    - Topic-based search and exploration features can be made available by mobile search engines by utilising topic modelling. Search engines can give users the ability to filter their search results based

on particular areas of interest by grouping content into several topics. This enables users to investigate data inside a certain domain or theme, resulting in a more targeted and effective search experience. Mobile users who desire quick access to pertinent information and have constrained screen area may find topic-based search to be especially helpful.

- Mobile search engines can produce more individualised content recommendations with the help of topic modelling. Search engines can recommend similar publications or resources that match the user's preferences by analysing the themes of interest for specific users based on their search history, browsing habits, or interactions. This improves the diversity and relevance of content recommendations, enabling users to learn new knowledge or delve further into interests within the constrained parameters of a mobile device.
- Mobile search engines can use topic modelling to quickly identify new trends or themes. Search engines can determine the most important or well-liked topics at any particular time by examining a vast number of news articles, social media posts, or other information sources. This makes it possible for search engines to update mobile users about the most recent advancements in their areas of interest by providing them with up-to-date news summaries or hot topics.

3. Natural Language Generation:
   Natural language summaries or snippets for search results can be produced using NLP. The search engine can offer brief and insightful summaries for each search result instead of a list of links. These summaries can be produced using methods like text summarization, sentiment analysis, and content extraction, allowing users to immediately understand the significance of each result and make decisions without having to click on every link.

By implementing these techniques and strategies, mobile search engines can enhance the user experience, improve the relevance of search results, and provide more personalized and contextually aware information to users.

Q2)

(i)

The robot can be categorized as a service robot made to maintain and clean the home's environment. We can also categorize it as a mobile robot based on its mobility.

(ii)

For this smart room-cleaning robot, a combination of passive and active sensors would be ideal. Passive sensors, such as camera sensors, receive and interpret existing information from the environment without actively emitting any signals. They are useful for perceiving the surroundings and identifying objects.

Active sensors, such as ultrasonic sensors and bumper sensors, emit signals (sound waves or physical contact) and measure their reflection or response. These sensors provide immediate feedback and can detect objects or obstacles that might not be easily visible to passive sensors.

By combining both passive and active sensors, the robot can gather comprehensive information about its surroundings, ensuring effective navigation, obstacle detection, and cleaning capabilities.

A few sensor examples are:

- Infrared (IR) Sensor:Objects can be found using infrared (IR) sensors, which work by monitoring the infrared radiation that objects emit. The robot can navigate securely by using these sensors to identify walls, furniture, and other obstructions.
- Bumper Sensor:Physical switches called bumper sensors are placed at the robot's front or sides. The switch is depressed when the robot collides with something, signaling a collision. Bumper sensors can give the robot quick feedback, assisting it in identifying and avoiding obstacles.

- Camera Sensor:A camera sensor can offer visual data about the environment, assisting the robot in recognising things in its path such as furniture and barriers. The robot can identify dust and debris on the floor by using it for object identification as well.
- Ultrasonic Sensor:When sound waves are emitted by an ultrasonic sensor, the amount of time it takes for the waves to return is measured. With the aid of this sensor, obstacles can be located and their distance from the robot determined. It is especially helpful for locating objects that are out of the camera sensor's field of view.

(iii)

The smart robot would need a variety of effectors to carry out the duties necessary for cleaning a room. The following effectors would be required:

- Vacuum Cleaner: For gathering dust and grime from the floors, a vacuum cleaner effector would be necessary. To collect debris into a collecting container or bag, it can use suction power. The effector should be made to clean various surfaces efficiently and should contain filters to stop dust from returning to the environment.

- Sweeping Brushes: Rotating brushes that sweep the floors and collect loose waste like crumbs, hair, or pet dander can be added to the robot. These brushes may be made to reach nooks and crannies for complete cleaning. They can either be installed as removable modules or fixed to the base of the robot.

- Wheel or Track Drive System: For the robot to move around the house effectively, a drive system is required. Mobile robots frequently use wheel- or track-based systems because they allow for movement and maneuverability over a variety of surfaces, including carpets, tiles, and hardwood floors. The drive system should be built with obstacle avoidance and fluid motion in mind.

- Dirt Collection System:A method for gathering and storing the gathered dust and debris should be built into the robot. It can feature

a dustbin that is detachable or self-emptying, making cleanup and maintenance simple. The amount of dirt that the trashcan can hold should be adequate to reduce the number of times it needs to be emptied.

These effectors are necessary to address different cleaning aspects and ensure efficient and thorough cleaning of the room. They provide the robot with the ability to move, collect dust and sweep floors covering a wide range of cleaning requirements in a typical home environment.

(iv)

Combining a knowledge base with reasoning strategies can help make the robot that cleans rooms intelligent. Here are some things to think about:

<u>Knowledge Base:</u>

- Environmental Knowledge: The robot should be familiar with the house's layout, including the arrangement of the rooms' furniture and any potential hazards. This knowledge can be pre-programmed or acquired through exploration and mapping.
- Cleaning Methods and Strategies: The robot should be knowledgeable about efficient cleaning methods for various cleaning jobs, such as sweeping, vacuuming, and mopping, as well as for different types of surfaces, such as carpet, tile, and hardwood. Either pre-programming or training data can be used to acquire this information.
- Knowledge of recognisable objects: The robot should have a database of furniture pieces including chairs, tables, and appliances that are frequently present in a space. Knowing this information makes it easier to spot barriers and spot places that need to be cleaned.
- Safety and operational standards: The robot needs to be knowledgeable about safety standards to prevent harm to the environment or to itself. These skills include being aware of boundaries, avoiding breakables, and knowing when to stop cleaning in particular circumstances.

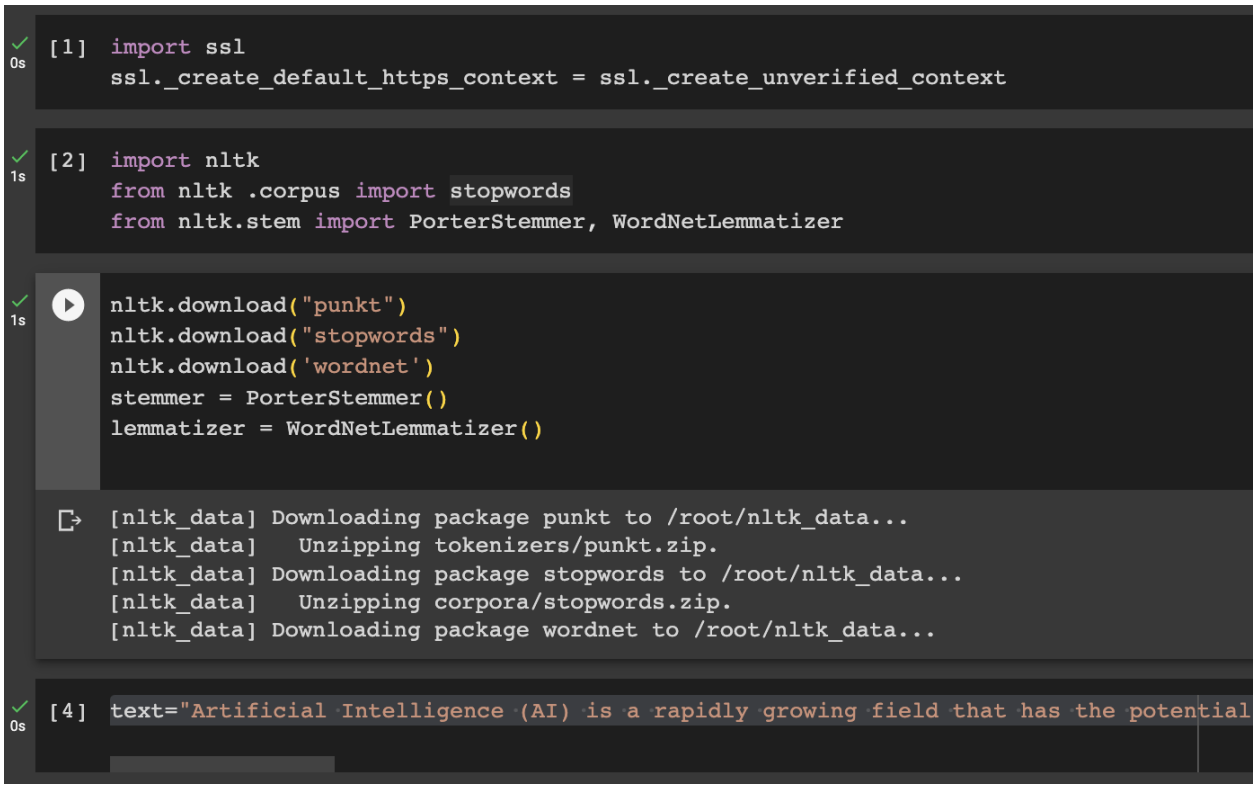
Reasoning Techniques:

- The robot must have the ability to reason in order to design the best routes for effective navigation within the house while avoiding obstacles. The optimum routes can be found using methods like A* search, Dijkstra's algorithm, or potential field methods.
- The robot must consider the degree of cleanliness in various locations and rank activities accordingly. It should determine which areas need to be cleaned right away and allocate the necessary resources to carry out the jobs efficiently.
- In order to analyse sensor feedback and gradually enhance its cleaning performance, the robot can apply reasoning processes. Based on customer preferences and comments, it can adjust its techniques, learn from mistakes, and improve its cleaning procedures.
- When deciding how to avoid obstacles, the robot should use the sensor data (from the camera and ultrasonic sensors, for example) to guide its reasoning. Real-time path planning and collision detection are involved in this.

The room-cleaning robot can make intelligent decisions, move around its environment quickly, recognise and deal with barriers, and use the right cleaning techniques by integrating its knowledge base with reasoning techniques. The knowledge base gives the required context and information, and reasoning enables the robot to successfully interpret and use that knowledge in practical settings.

Section B (Code Implementation)

Q1)

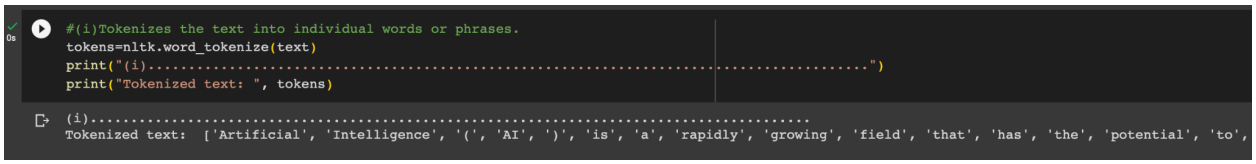


```
[1]  import ssl
     ssl._create_default_https_context = ssl._create_unverified_context

[2]  import nltk
     from nltk .corpus import stopwords
     from nltk.stem import PorterStemmer, WordNetLemmatizer

     nltk.download("punkt")
     nltk.download("stopwords")
     nltk.download('wordnet')
     stemmer = PorterStemmer()
     lemmatizer = WordNetLemmatizer()

     [nltk_data] Downloading package punkt to /root/nltk_data...
     [nltk_data]   Unzipping tokenizers/punkt.zip.
     [nltk_data] Downloading package stopwords to /root/nltk_data...
     [nltk_data]   Unzipping corpora/stopwords.zip.
     [nltk_data] Downloading package wordnet to /root/nltk_data...

[4]  text="Artificial Intelligence (AI) is a rapidly growing field that has the potential
```

i) The code performs tokenization of the given text using the `nltk.word_tokenize()` function. Tokenization is the process of splitting a text into individual words or phrases, known as tokens. In this case, the `text` variable contains a long passage of text about artificial intelligence (AI).

The `nltk.word_tokenize()` function takes the `text` as input and returns a list of tokens. Each token represents a word or a punctuation mark from the text. The tokens are generated by splitting the text at whitespace and punctuation boundaries.

The code then prints the tokenized text using the `print()` function. The output will be a list of tokens representing the words and punctuation marks found in the text. This step is useful for further text processing and analysis tasks that operate on individual words or phrases.

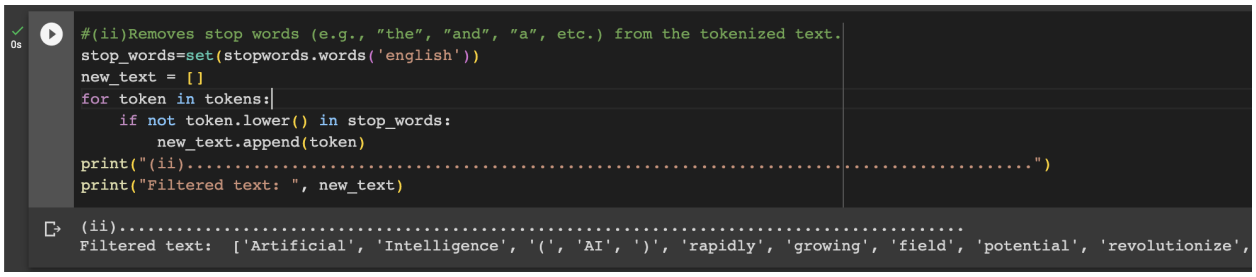


ii) The code removes stop words from the tokenized text using the NLTK stopwords corpus. Stop words are commonly occurring words in a language that do not carry significant meaning and are often removed in text analysis tasks to focus on the more important content words.

The code first imports the `stopwords` module from NLTK and retrieves a set of English stop words using `stopwords.words('english')`. The stop words are stored in the `stop_words` variable.

Then, a new empty list called `new_text` is created to store the filtered tokens without stop words. The code iterates over each token in the `tokens` list, and for each token, it checks if the lowercase version of the token exists in the `stop_words` set. If the token is not a stop word, it is appended to the `new_text` list.

Finally, the code prints the filtered text using the `print()` function. The output will be a list of tokens from the original text, excluding the stop words. This step helps to remove common and less informative words, allowing for a focus on the more meaningful content in subsequent analysis or processing.



iii) The code performs stemming and lemmatization on the tokenized text obtained from the previous step. Stemming and lemmatization are techniques used to reduce words to their base or root form.

In the code snippet, two separate lists are created: `stem` and `lemm`. These lists will store the stemmed and lemmatized versions of the tokens, respectively.

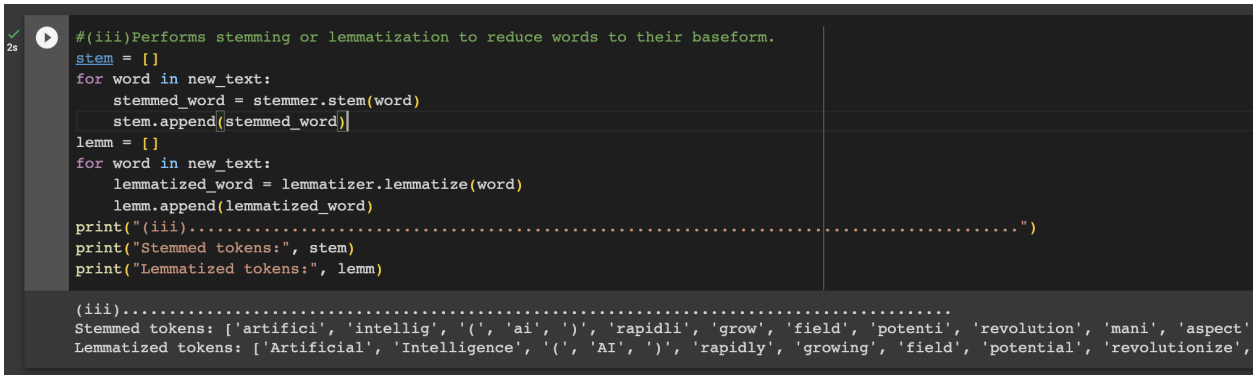
The code then iterates over each token in the `new_text` (assuming `new_text` is the variable containing the tokenized text). For each token, the code applies stemming using the `stemmer.stem()` function and lemmatization using the `lemmatizer.lemmatize()` function.

In each iteration, the stemmed or lemmatized word is appended to the corresponding list (`stem` or `lemm`).

Finally, the code prints the stemmed tokens and lemmatized tokens using the `print()` function. The output will be the lists `stem` and `lemm`, which contain the stemmed and lemmatized versions of the tokens, respectively.

Stemming aims to remove prefixes and suffixes from words, resulting in a base form that may not always be a valid word. Lemmatization, on the other hand, takes into account the context and morphological analysis of words to reduce them to their base form, which is a valid word.

Both stemming and lemmatization are useful in text analysis tasks, such as information retrieval, text classification, and natural language understanding, as they help to consolidate different word forms into a common base form.



```
#(iii)Performs stemming or lemmatization to reduce words to their baseform.
stem = []
for word in new_text:
    stemmed_word = stemmer.stem(word)
    stem.append(stemmed_word)
lemm = []
for word in new_text:
    lemmatized_word = lemmatizer.lemmatize(word)
    lemm.append(lemmatized_word)
print("(iii).....")
print("Stemmed tokens:", stem)
print("Lemmatized tokens:", lemm)
```

```
(iii).....
Stemmed tokens: ['artifici', 'intellig', '(', 'ai', ')', 'rapidli', 'grow', 'field', 'potenti', 'revolution', 'mani', 'aspect'
Lemmatized tokens: ['Artificial', 'Intelligence', '(', 'AI', ')', 'rapidly', 'growing', 'field', 'potential', 'revolutionize',
```

iv)

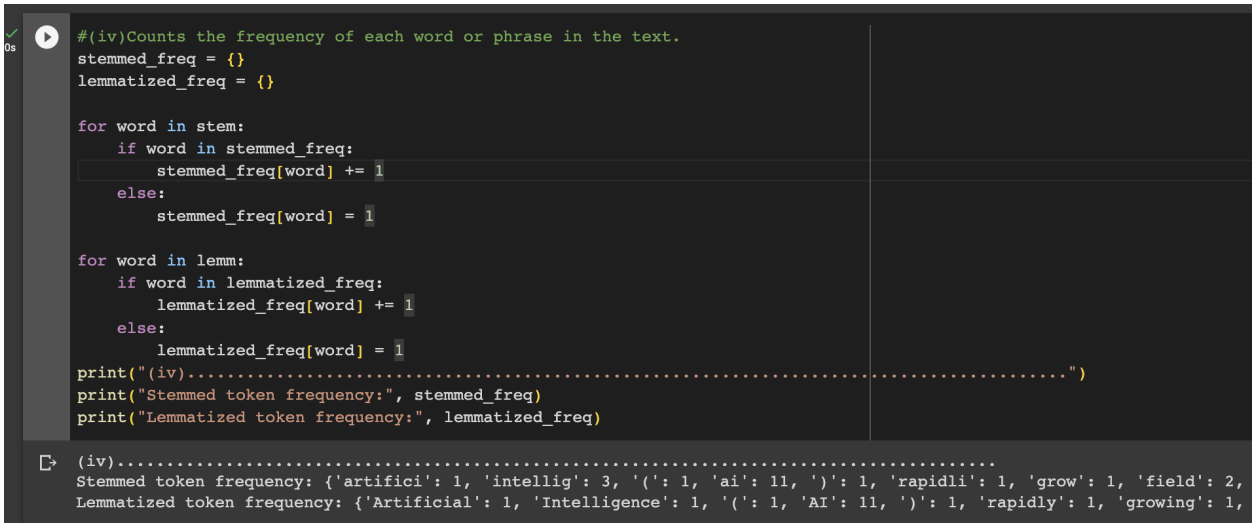
The code performs frequency analysis of the tokenized text using stemming and lemmatization techniques.

First, the code initializes two empty dictionaries `stemmed_freq` and `lemmatized_freq` to store the frequency of each stemmed and lemmatized word, respectively.

Then, it iterates through each stemmed word in the list `stem`, which was obtained in step (iii) using the PorterStemmer. For each stemmed word, the code checks if the word is already present in the `stemmed_freq` dictionary. If it is, then the frequency of that word is incremented by 1. If it is not, then the word is added to the dictionary with a frequency of 1.

Similarly, the code iterates through each lemmatized word in the list `lemm`, which was obtained in step (iii) using the WordNetLemmatizer. For each lemmatized word, the code checks if the word is already present in the `lemmatized_freq` dictionary. If it is, then the frequency of that word is incremented by 1. If it is not, then the word is added to the dictionary with a frequency of 1.

Finally, the code prints the frequency of each stemmed and lemmatized word using the `print()` function. The output will be two dictionaries containing the frequency of each word after stemming and lemmatization, respectively. This step is useful for analyzing the most common words in the text and understanding the distribution of word usage.



```
#(iv)Counts the frequency of each word or phrase in the text.
stemmed_freq = {}
lemmatized_freq = {}

for word in stem:
    if word in stemmed_freq:
        stemmed_freq[word] += 1
    else:
        stemmed_freq[word] = 1

for word in lemm:
    if word in lemmatized_freq:
        lemmatized_freq[word] += 1
    else:
        lemmatized_freq[word] = 1
print("(iv).....................................................................................")
print("Stemmed token frequency:", stemmed_freq)
print("Lemmatized token frequency:", lemmatized_freq)
```

```
(iv).....................................................................................
Stemmed token frequency: {'artifici': 1, 'intellig': 3, '(': 1, 'ai': 11, ')': 1, 'rapidli': 1, 'grow': 1, 'field': 2,
Lemmatized token frequency: {'Artificial': 1, 'Intelligence': 1, '(': 1, 'AI': 11, ')': 1, 'rapidly': 1, 'growing': 1,
```

v)The code snippet you provided calculates and outputs the most frequent stemmed and lemmatized tokens (words or phrases) in the given text.

First, the code assumes that you have previously calculated the frequency counts of stemmed and lemmatized tokens using `stemmed_freq` and `lemmatized_freq` dictionaries.

The code then sorts the stemmed and lemmatized frequency dictionaries in descending order based on the frequency count using the `sorted()` function. The `key=lambda x: x[1]` argument specifies that the sorting should be based on the second element (frequency count) of each key-value pair in the dictionaries.

Next, the code selects the top `n` items from the sorted dictionaries using the `[:n]` slicing notation.

Finally, the code prints the most frequent stemmed and lemmatized tokens along with their corresponding frequency counts. It iterates over the selected items from the sorted dictionaries using a loop and prints each token and its frequency using the `print()` function.

The output will display the most frequent stemmed tokens and lemmatized tokens separately, with the specified number of tokens (`n`) and their corresponding frequency counts. This information can help identify the most commonly occurring words or phrases in the text.

```python
#(v)Outputs the most frequent words or phrases in the text, along with their frequency count.
n = 10
most_common_stemmed = sorted(stemmed_freq.items(), key=lambda x: x[1], reverse=True)[:n]
most_common_lemmatized = sorted(lemmatized_freq.items(), key=lambda x: x[1], reverse=True)[:n]
print("(v).....................................................................................")
print("Most frequent stemmed tokens (top {}):".format(n))
for token, freq in most_common_stemmed:
    print("{}: {}".format(token, freq))
print("Most frequent lemmatized tokens (top {}):".format(n))
for token, freq in most_common_lemmatized:
    print("{}: {}".format(token, freq))
```

```
(v).....................................................................................
Most frequent stemmed tokens (top 10):
,: 17
.: 13
ai: 11
concern: 5
potenti: 4
develop: 4
machin: 4
human: 4
use: 4
also: 4
Most frequent lemmatized tokens (top 10):
,: 17
.: 13
AI: 11
concern: 5
potential: 4
human: 4
used: 4
also: 4
machine: 3
improve: 3
```

Q2)

```
[ ] path ='/content/drive/MyDrive/MiMM-SBILab'
```

```
[ ] import os
    import random
    from PIL import Image,ImageOps
    import matplotlib.pyplot as plt
```

```
[ ] image_files = [os.path.join(path, file) for file in os.listdir(path) if file.endswith('.bmp')]
```

```
[ ] random_images = random.sample(image_files, 3)
```

i)

```
#(i)
# Define the desired size
width, height = 300, 150

# Loop through each random image and resize it
for image_filename in random_images:
    # Open the image
    img = Image.open(os.path.join(path, image_filename))

    # Resize the image
    img_resized = img.resize((width, height))

     # Display the original and resized images side by side
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].imshow(img)
    axes[0].set_title('Original Image')
    axes[1].imshow(img_resized)
    axes[1].set_title('Resized Image')
    plt.show()
```

The code is written in Python and it aims to resize a set of random images to a desired width and height.

In the first line, the desired size is defined by assigning values to the variables `width` and `height`.

The code then loops through each random image file in a given directory (`path`).

Inside the loop, the image is opened using the `Image.open()` function from the PIL (Python Imaging Library) module.

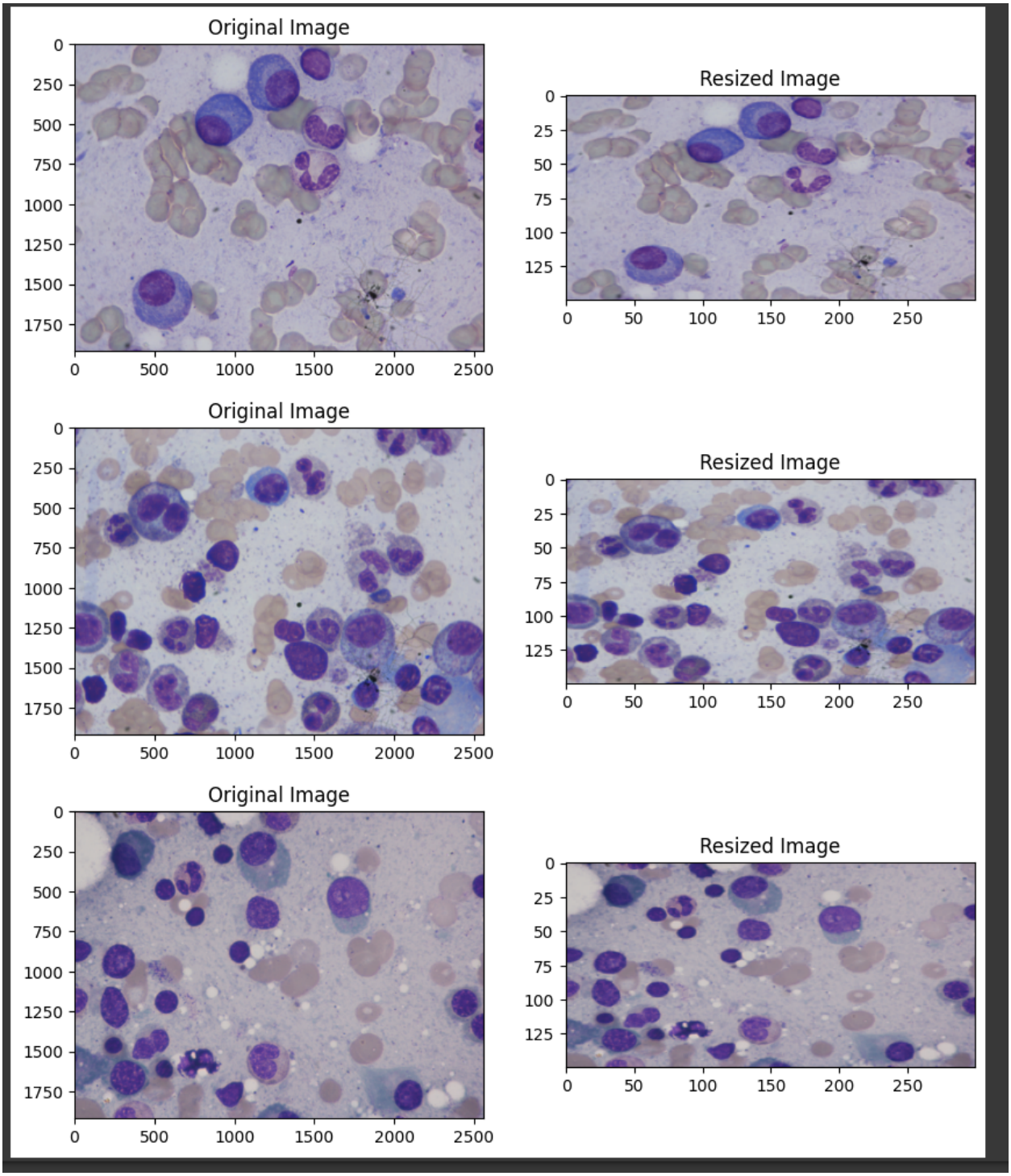
The `resize()` method is called on the image object to resize it to the desired width and height. The resized image is stored in the variable `img_resized`.

Next, the original and resized images are displayed side by side using the `subplots()` function from the `matplotlib.pyplot` module. The figure size is set to 10 inches by 5 inches.

Two subplots are created, one for the original image and one for the resized image. The `imshow()` function is used to display the images on the respective subplots.

The `set_title()` function is called to set titles for the subplots indicating whether it's the original or resized image.

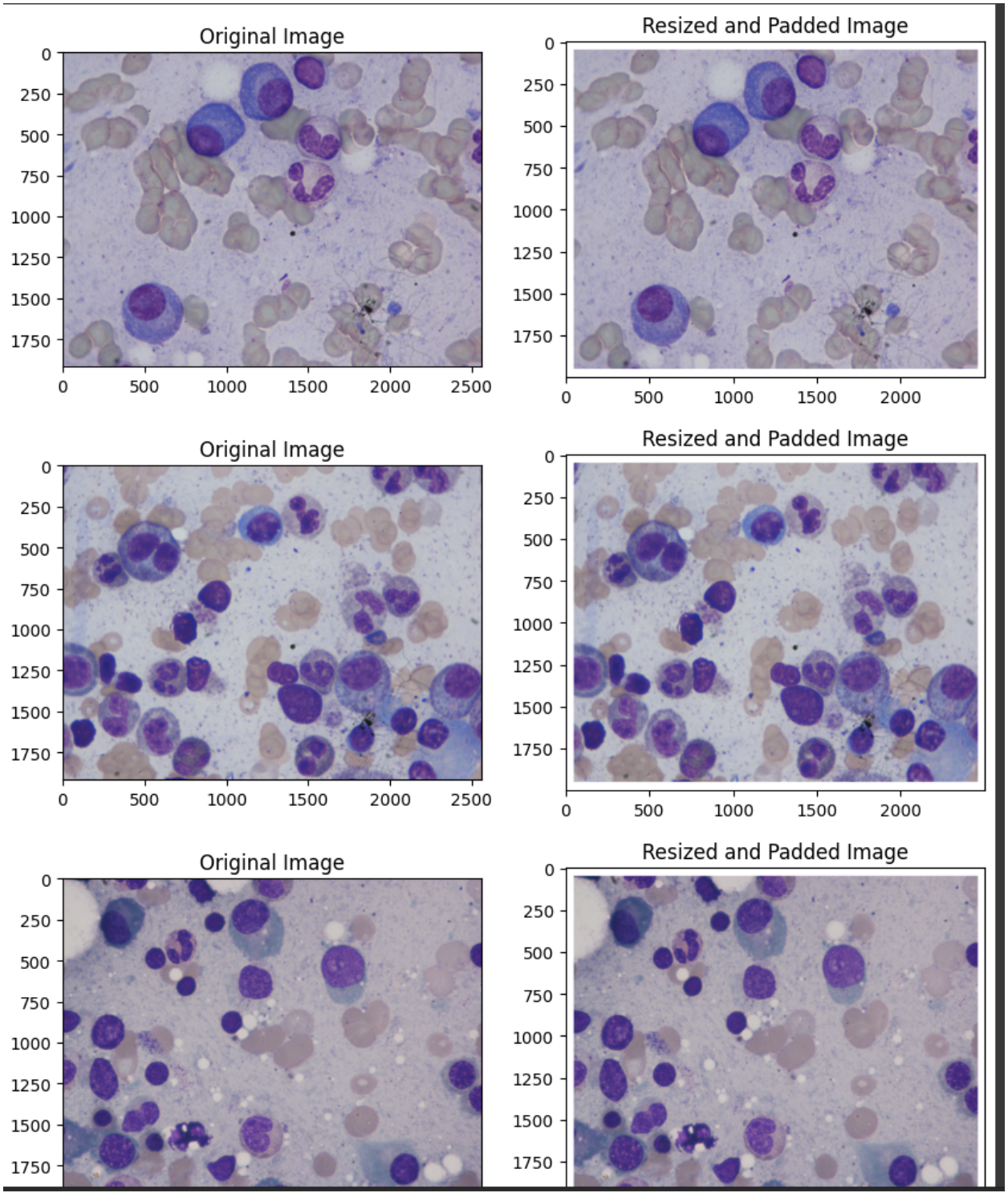
Finally, the `show()` function is called to display the figure with the images. This process is repeated for each random image in the loop.


Original Image
Resized Image
Original Image
Resized Image
Original Image
Resized Image

ii)

```
#(ii)
# Define the desired size and padding
width, height = 2500, 2000

for image_filename in random_images:
    # Open the image
    img = Image.open(os.path.join(path, image_filename))



   # Calculate the new size
    img_ratio = img.size[0] / img.size[1]
    new_width = img.size[0] + 100
    new_height = img.size[1] + 100

    # Create a new padded image
    pad_img = Image.new("RGB", (new_width, new_height), (300, 300, 300))

    # Calculate the paste coordinates
    paste_x = 50
    paste_y = 50

    # Paste the original image onto the padded image
    pad_img.paste(img, (paste_x, paste_y))


    # Resize and pad the image
    img_resized = ImageOps.pad(pad_img.resize((width, height)), (width, height), method=Image.NEAREST, color='white', centering=(0.5, 0.5))


    # Display the original and resized images side by side
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].imshow(img)
    axes[0].set_title('Original Image')
    axes[1].imshow(img_resized)
    axes[1].set_title('Resized and Padded Image')
    plt.show()
```

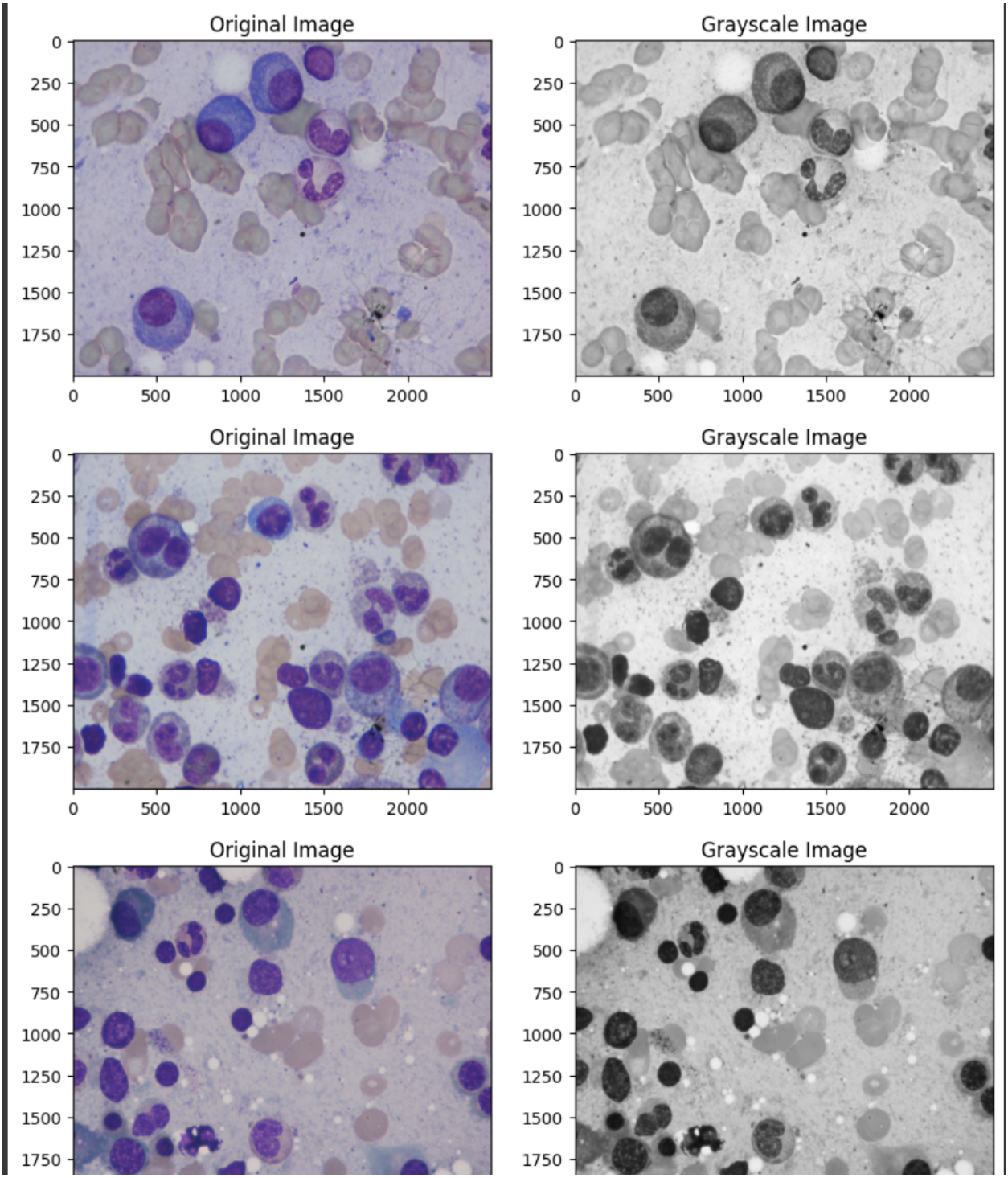
The code takes a set of random image files from a directory and resizes and pads them to a desired size while maintaining their aspect ratio. It first defines the desired width and height, and then loops through the random image files, opening each image and calculating the new dimensions for padding. It then creates a new padded image using the calculated dimensions and pastes the original image onto it. The padded image is then resized to the desired size and further padded if necessary to fit the dimensions perfectly. Finally, the original and resized images are displayed side by side using matplotlib. The resulting images are resized and padded while preserving their aspect ratio and centered within the desired size.


Original Image
Resized and Padded Image
Original Image
Resized and Padded Image
Original Image
Resized and Padded Image

iii)

```python
#(iii)
for image_filename in random_images:
    # Open the image
    img = Image.open(os.path.join(path, image_filename))

    # Resize the image
    img_resized = img.resize((width, height))

    # Convert to grayscale
    img_gray = img_resized.convert('L')

    # Save the grayscale image with a new filename
    #img_gray.save(os.path.join(path, f'{image_filename.split(".")[0]}_gray.jpg'))

    # Display the original and grayscale images side by side
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].imshow(img_resized)
    axes[0].set_title('Original Image')
    axes[1].imshow(img_gray, cmap='gray')
    axes[1].set_title('Grayscale Image')
    plt.show()
```

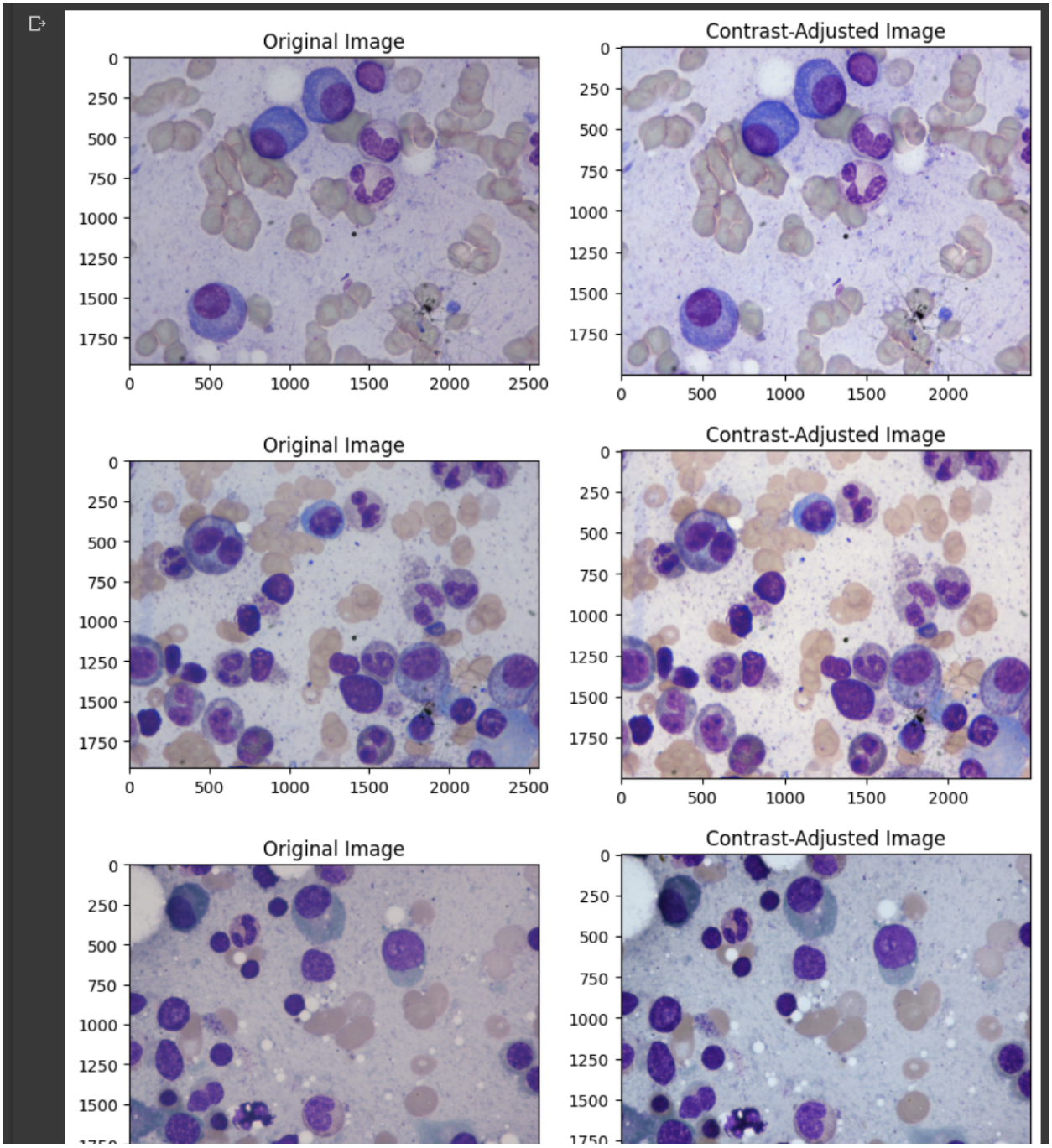
The code above is a Python script that performs several image processing tasks on a set of random images. It first loops through a list of image filenames, opens each image using the Python Imaging Library (PIL), and resizes it to a specified width and height. Then, it converts the resized image to grayscale using the 'L' mode in PIL and saves it with a new filename. Finally, it displays the original and grayscale images side by side using the Matplotlib library. The resulting output shows a comparison of the original and grayscale versions of each image.


Original Image
Grayscale Image
Original Image
Grayscale Image
Original Image
Grayscale Image
0
250
500
750
1000
1250
1500
1750
0
500
1000
1500
2000

iv)

```
#(iv)
# Loop through each random image and resize it
for image_filename in random_images:
    # Open the image
    img = Image.open(os.path.join(path, image_filename))

    # Resize the image
    img_resized = img.resize((width, height))

    # Change the contrast of the image
    img_contrast = ImageOps.autocontrast(img_resized)

    # Save the resized and contrast-adjusted image with a new filename
    #new_filename = f'{image_filename.split(".")[0]}_resized_contrast.jpg'
    #img_contrast.save(os.path.join(path, new_filename))

    # Display the original, resized, and contrast-adjusted images side by side
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].imshow(img)
    axes[0].set_title('Original Image')
    axes[1].imshow(img_contrast)
    axes[1].set_title('Contrast-Adjusted Image')
    plt.show()
```

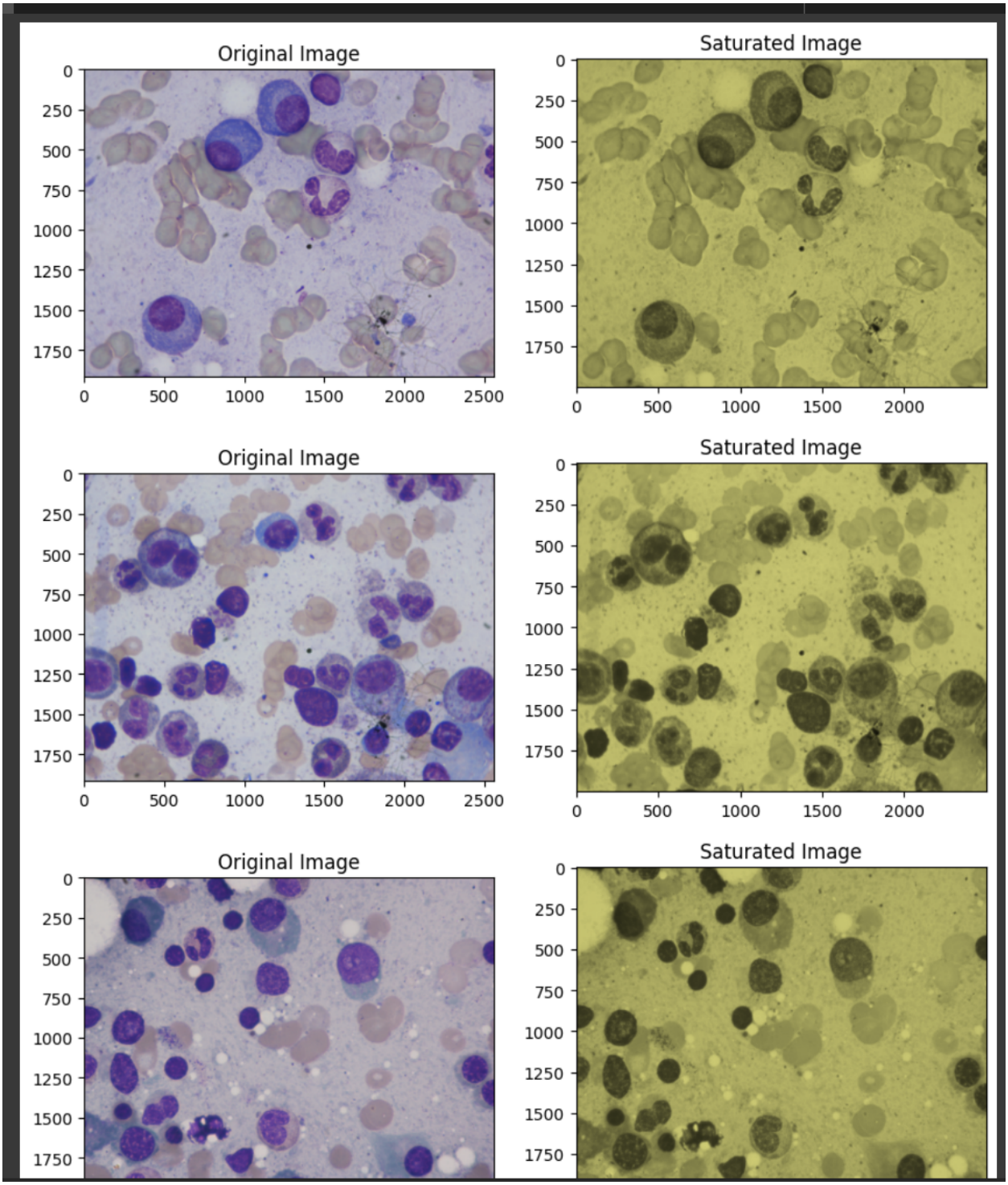
This code snippet loops through a list of random image filenames and performs several operations on each image. First, it opens the image using the PIL library. Then, it resizes the image to the specified width and height. After resizing, it adjusts the contrast of the image using the `ImageOps.autocontrast` function. Finally, it displays the original image and the contrast-adjusted image side by side using matplotlib. The code is commented out for saving the resized and contrast-adjusted image with a new filename, but that functionality can be uncommented if desired. Overall, the code processes each random image by resizing it and enhancing its contrast, and then visually compares the original and adjusted images.


Original Image
Contrast-Adjusted Image
Original Image
Contrast-Adjusted Image
Original Image
Contrast-Adjusted Image

v)

```python
#(v)
# Define the desired size for resizing
width, height = 2500, 2000

# Define the desired saturation level
saturation_level = 0.5  # 0.0 will make the image grayscale, and 1.0 will keep the original saturation level

# Loop through each random image and resize it
for image_filename in random_images:
    # Open the image
    img = Image.open(os.path.join(path, image_filename))

    # Resize the image
    img_resized = img.resize((width, height))

    # Change the saturation of the image
    img_saturated = ImageOps.colorize(ImageOps.grayscale(img_resized), (0, 0, 0), (255, 255, int(255 * saturation_level)))

    # Save the resized and saturated image with a new filename
    new_filename = f'{image_filename.split(".")[0]}_resized_saturated.jpg'
    img_saturated.save(os.path.join(path, new_filename))

    # Display the original, resized, and saturated images side by side
    fig, axes = plt.subplots(1, 2, figsize=(10, 5))
    axes[0].imshow(img)
    axes[0].set_title('Original Image')
    axes[1].imshow(img_saturated)
    axes[1].set_title('Saturated Image')
    plt.show()
```

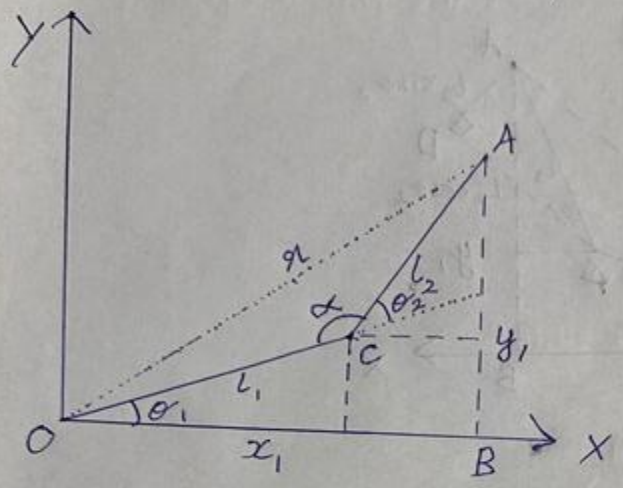
This code is used to resize and adjust the saturation level of a series of random images. It starts by defining the desired size and saturation level. Then, for each image in the random_images list, it opens the image, resizes it to the desired width and height, and changes its saturation level. The saturation adjustment is achieved by converting the resized image to grayscale using the `ImageOps.grayscale()` function, and then colorizing it with the desired saturation level using the `ImageOps.colorize()` function. The resized and saturated image is saved with a new filename. Finally, the code displays the original and saturated images side by side using matplotlib.


Original Image
Saturated Image
Original Image
Saturated Image
Original Image
Saturated Image

Q3)

Derivation:

Derivation of the geomatric solution for the inverse kinematics of a 2 DOF manipulator.



Here

$$r^2 = x_1^2 + y_1^2$$

Applying cosine rule on ΔOCA

$$\Rightarrow r^2 = l_1^2 + l_2^2 - 2 l_1 l_2 \cos\alpha$$

$$\cos\alpha = \frac{l_1^2 + l_2^2 - r^2}{2 l_1 l_2}$$

ATF (figure)

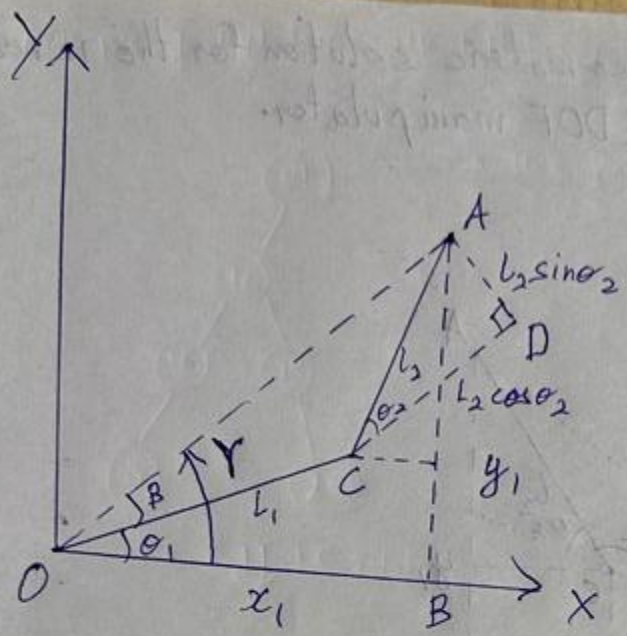
$$\theta_2 = \pi - \alpha$$

$$\alpha = \pi - \alpha$$

$$\Rightarrow \cos\alpha = -\cos\theta_2$$

$$\Rightarrow \cos\theta_2 = \frac{r^2 - l_1^2 - l_2^2}{2 l_1 l_2}$$

$$\boxed{\theta_2 = \cos^{-1}\left(\frac{x_1^2 + y_1^2 - l_1^2 - l_2^2}{2 l_1 l_2}\right)}$$



Here

$$\tan\beta = \frac{DA}{OD}$$

$$\beta = \tan^{-1}\left(\frac{l_2 \sin\theta_2}{l_1 + l_2\cos\theta_2}\right)$$

$$\tan\gamma = \frac{AB}{OB}$$

$$\gamma = \tan^{-1}\left(\frac{y_1}{x_1}\right)$$

$$\theta_1 = \gamma - \beta$$

$$\boxed{\theta_1 = \tan^{-1}\left(\frac{y_1}{x_1}\right) - \tan^{-1}\left(\frac{l_2 \sin\theta_2}{l_1 + l_2\cos\theta_2}\right)}$$

Python Code:

```python
import math

def inverse_kinematics(x, y, L1, L2):
    # Validate if the end-effector position is reachable
    distance = math.sqrt(x**2 + y**2)
    if distance > L1 + L2:
        raise ValueError("The end-effector position is not reachable with the given link lengths.")

    # Step 2: Find θ2
    cos_theta2 = (x**2 + y**2 - L1**2 - L2**2) / (2 * L1 * L2)
    theta2 = math.acos(cos_theta2)

    # Step 1: Find θ1
    theta1 = math.atan2(y, x) - math.atan2(L2 * math.sin(theta2), L1 + L2 * cos_theta2)

    return theta1, theta2

# Input from the user
x = float(input("Enter the x-coordinate of the end-effector: "))
y = float(input("Enter the y-coordinate of the end-effector: "))
L1 = float(input("Enter the length of link 1: "))
L2 = float(input("Enter the length of link 2: "))

try:
    theta1, theta2 = inverse_kinematics(x, y, L1, L2)
    print("Joint angles:")
    print("θ1 =", round(theta1, 6))
    print("θ2 =", round(theta2, 6))
except ValueError as e:
    print("Error:", str(e))
```

```
Enter the x-coordinate of the end-effector: 2.5
Enter the y-coordinate of the end-effector: 1.8
Enter the length of link 1: 3.0
Enter the length of link 2: 2.5
Joint angles:
θ1 = -0.223006
θ2 = 1.964921
```

This code implements an inverse kinematics calculation for a two-link robotic arm. Inverse kinematics is the process of determining the joint angles required to reach a specific end-effector position.

The `inverse_kinematics` function takes the Cartesian coordinates (x, y) of the end-effector, as well as the lengths of the two arm links (L1 and L2) as input. It first validates if the end-effector position is reachable by checking if the distance is within the sum of the link lengths. If the position is not reachable, it raises a `ValueError` with an appropriate error message.

If the position is reachable, the function proceeds to calculate the joint angles. It uses trigonometric formulas to determine the angles θ1 and θ2.

Finally, the main part of the code prompts the user to enter the end-effector coordinates and link lengths. It calls the `inverse_kinematics` function with the provided inputs and prints the resulting joint angles. If an error occurs during the calculation, it catches the `ValueError` and prints the error message instead.

Overall, this code allows users to input the desired end-effector position and link lengths, and it calculates the corresponding joint angles necessary to achieve that position with a two-link robotic arm.