This code is for scanning an image of a document and processing it to obtain a clean version of the document with minimal noise. Let's go through it step by step:

Step -1 Getting Bird's Eye View of the Image

```
def scan(img):
    # Resize image to workable size
    dim_limit = 1080
    max_dim = max(img.shape)
    if max_dim > dim_limit:
        resize_scale = dim_limit / max_dim
        img = cv2.resize(img, None, fx=resize_scale, fy=resize_scale)
    # Create a copy of resized original image for later use
    orig_img = img.copy()
    # Repeated Closing operation to remove text from the document.
    kernel = np.ones((5, 5), np.uint8)
    img = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel, iterations=3)
    # GrabCut
    mask = np.zeros(img.shape[:2], np.uint8)
    bgdModel = np.zeros((1, 65), np.float64)
    fgdModel = np.zeros((1, 65), np.float64)
    rect = (20, 20, img.shape[1] - 20, img.shape[0] - 20)
    cv2.grabCut(img, mask, rect, bgdModel, fgdModel, 5, cv2.GC_INIT_WITH_RECT)
    mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
    img = img * mask2[:, :, np.newaxis]

    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    gray = cv2.GaussianBlur(gray, (11, 11), 0)
    # Edge Detection.
    canny = cv2.Canny(gray, 0, 200)
    canny = cv2.dilate(canny, cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,
5)))

    # Finding contours for the detected edges.
    contour, hierarchy = cv2.findContours(canny, cv2.RETR_LIST,
cv2.CHAIN_APPROX_NONE)
    # Keeping only the largest detected contour.
    page = sorted(contour, key=cv2.contourArea, reverse=True)[:5]

    # Detecting Edges through Contour approximation.
    # Loop over the contours.
    if len(page) == 0:
        return orig_img
    for c in page:
        # Approximate the contour.
        epsilon = 0.02 * cv2.arcLength(c, True)
        corners = cv2.approxPolyDP(c, epsilon, True)
        # If our approximated contour has four points.
        if len(corners) == 4:
            break
```

```
    # Sorting the corners and converting them to desired shape.
    corners = sorted(np.concatenate(corners).tolist())
    # For 4 corner points being detected.
    corners = order_points(corners)

    destination_corners = find_dest(corners)

    h, w = orig_img.shape[:2]
    # Getting the homography.
    M = cv2.getPerspectiveTransform(np.float32(corners),
np.float32(destination_corners))
    # Perspective transform using homography.
    final = cv2.warpPerspective(orig_img, M, (destination_corners[2][0],
destination_corners[2][1]),
                                flags=cv2.INTER_LINEAR)
    return finalThis is a function definition. It takes an image as input and
returns a processed version of it.
```

```
dim_limit = 1080
max_dim = max(img.shape)
if max_dim > dim_limit:
    resize_scale = dim_limit / max_dim
    img = cv2.resize(img, None, fx=resize_scale, fy=resize_scale)
```

The function first resizes the input image if its size is larger than a certain threshold (dim_limit). The maximum dimension of the input image is calculated and if it is greater than the threshold, the image is resized to fit within the limit while maintaining its aspect ratio. This is done to ensure that the processing is faster and more efficient.

```
orig_img = img.copy()
```

This creates a copy of the resized image for later use.

```
kernel = np.ones((5, 5), np.uint8)
img = cv2.morphologyEx(img, cv2.MORPH_CLOSE, kernel, iterations=3)
```

The function then applies a morphological operation called "Closing" to the input image. Closing is a dilation followed by an erosion operation and is commonly used to remove noise and fill gaps in an image. In this case, it is used to remove any text from the document. A kernel of size 5x5 is used for this operation and it is applied three times.

```
mask = np.zeros(img.shape[:2], np.uint8)
bgdModel = np.zeros((1, 65), np.float64)
fgdModel = np.zeros((1, 65), np.float64)
rect = (20, 20, img.shape[1] - 20, img.shape[0] - 20)
cv2.grabCut(img, mask, rect, bgdModel, fgdModel, 5, cv2.GC_INIT_WITH_RECT)
mask2 = np.where((mask == 2) | (mask == 0), 0, 1).astype('uint8')
img = img * mask2[:, :, np.newaxis]
```

The function then uses the GrabCut algorithm to separate the foreground (document) from the background. GrabCut is an iterative segmentation algorithm that uses a user-defined bounding box to separate the foreground from the background. The algorithm updates the segmentation iteratively based on the user input. In this case, a rectangular bounding box is defined using the dimensions of the input image with some margin. The GrabCut algorithm then segments the input image and returns a mask. The mask is then thresholded to create a binary mask, which is used to remove the background from the input image.

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
gray = cv2.GaussianBlur(gray, (11, 11), 0)
```

The function converts the input image to grayscale and applies a Gaussian blur to it. Gaussian blur is a common filter used to smooth an image and reduce noise. In this case, it is used to prepare the image for edge detection.

```
canny = cv2.Canny(gray, 0, 200)
canny = cv2.dilate(canny, cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5)))
```

The function then applies the Canny edge detection algorithm to the blurred image to detect edges. Canny is an edge detection algorithm that uses a multi-stage process to detect a wide range of edges in an image. In this case, it is used to detect the edges

```
destination_corners = find_dest(corners)
```

```
def order_points(pts):
    '''Rearrange coordinates to order:
      top-left, top-right, bottom-right, bottom-left'''
    rect = np.zeros((4, 2), dtype='float32')
    pts = np.array(pts)
    s = pts.sum(axis=1)
    # Top-left point will have the smallest sum.
    rect[0] = pts[np.argmin(s)]
    # Bottom-right point will have the largest sum.
    rect[2] = pts[np.argmax(s)]

    diff = np.diff(pts, axis=1)
    # Top-right point will have the smallest difference.
    rect[1] = pts[np.argmin(diff)]
    # Bottom-left will have the largest difference.
    rect[3] = pts[np.argmax(diff)]
    # return the ordered coordinates
    return rect.astype('int').tolist()
```

The `order_points()` function takes in the detected corners of the page as an argument and reorders them to have the top-left, top-right, bottom-right, and bottom-left points in a consistent order. This is important for the later steps of the image processing pipeline to work correctly.

The function first creates a 4x2 numpy array of float32 data type called `rect` to store the reordered points. It then converts the input `pts` argument to a numpy array and computes the sum of each row of coordinates using the `sum()` method and the `axis=1` argument. The `argmin()` and `argmax()` methods are then used to find the indices of the points that have the smallest and largest sums, respectively. The top-left point is assigned to `rect[0]` and the bottom-right point is assigned to `rect[2]`.

The function then computes the difference between the x and y coordinates of the input `pts` array using the `diff()` method and the `axis=1` argument. The `argmin()` and `argmax()` methods are again used to find the indices of the points that have the smallest and largest differences, respectively. The top-right point is assigned to `rect[1]` and the bottom-left point is assigned to `rect[3]`.

Finally, the function returns the reordered `rect` array as a list of integers using the `astype()` method and the `'int'` argument.

```
def find_dest(pts):

    (tl, tr, br, bl) = pts
    # Finding the maximum width.
    widthA = np.sqrt(((br[0] - bl[0]) ** 2) + ((br[1] - bl[1]) ** 2))
    widthB = np.sqrt(((tr[0] - tl[0]) ** 2) + ((tr[1] - tl[1]) ** 2))
    maxWidth = max(int(widthA), int(widthB))

    # Finding the maximum height.
    heightA = np.sqrt(((tr[0] - br[0]) ** 2) + ((tr[1] - br[1]) ** 2))
    heightB = np.sqrt(((tl[0] - bl[0]) ** 2) + ((tl[1] - bl[1]) ** 2))
    maxHeight = max(int(heightA), int(heightB))
    # Final destination co-ordinates.
    destination_corners = [[0, 0], [maxWidth, 0], [maxWidth, maxHeight], [0,
maxHeight]]

    return order_points(destination_corners)

 destination_corners = find_dest(corners)
```

The `find_dest(pts)` function calculates the final destination points of the transformed image.

The function takes in the four corner points of the detected document as `pts`. These points are assumed to be in an unordered format. The function then rearranges these points using the `order_points(pts)` function defined earlier to get the points in the order of the top-left, top-right, bottom-right, and bottom-left.

Once the points are ordered, the function calculates the maximum width and height of the document using the Euclidean distance formula between the corner points.

`widthA` calculates the width of the document using the distance between the bottom-right corner `br` and the bottom-left corner `bl`. Similarly, `widthB` calculates the width of the document using the distance between the top-right corner `tr` and the top-left corner `tl`. The `maxWidth` variable is then set to the maximum of these two values.

`heightA` calculates the height of the document using the distance between the top-right corner `tr` and the bottom-right corner `br`. Similarly, `heightB` calculates the height of the document using the distance between the top-left corner `tl` and the bottom-left corner `bl`. The `maxHeight` variable is then set to the maximum of these two values.

Finally, the `destination_corners` list is created using the `maxWidth` and `maxHeight` values calculated earlier. These points are in the order of top-left, top-right, bottom-right, and bottom-left. The `order_points(destination_corners)` function is then called to order the points correctly and the ordered coordinates of the final destination points are returned.

```
 h, w = orig_img.shape[:2]
 # Getting the homography.
 M = cv2.getPerspectiveTransform(np.float32(corners),
np.float32(destination_corners))
 # Perspective transform using homography.
 final = cv2.warpPerspective(orig_img, M, (destination_corners[2][0],
destination_corners[2][1]),
                                flags=cv2.INTER_LINEAR)
```

The code above performs the final transformation of the original image using the perspective transformation matrix obtained from the homography.

First, the height and width of the original image are extracted using the `shape` attribute of the NumPy array `orig_img`. Then, the `getPerspectiveTransform()` function of OpenCV is used to get the homography matrix `M`. This function takes in the four corner points of the source image (`corners`) and their corresponding four corner points in the destination image (`destination_corners`), and returns the homography matrix `M`.

Finally, the `warpPerspective()` function of OpenCV is used to perform the actual perspective transformation. It takes in the original image `orig_img`, the homography matrix `M`, and the output size of the destination image (the width and height of the bottom-right corner point of the destination image) as input. The `flags` parameter is set to `cv2.INTER_LINEAR`, which specifies the interpolation method used to perform the transformation.

The output of this function is the final transformed image `final`, which has been cropped and warped to appear as though it was taken from a bird's-eye view perspective.

Step-2 Tuning the Image

```python
def increase_brightness(img, value=30):
    hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
    h, s, v = cv2.split(hsv)
    lim = 255 - value
    v[v > lim] = 255
    v[v <= lim] += value
    final_hsv = cv2.merge((h, s, v))
    img = cv2.cvtColor(final_hsv, cv2.COLOR_HSV2BGR)
    return img


def final_image(rotated):
# Create our shapening kernel, it must equal to one eventually
  kernel_sharpening = np.array([[0,-1,0],
                                [-1, 5,-1],
                                [0,-1,0]])
  # applying the sharpening kernel to the input image & displaying it.
  sharpened = cv2.filter2D(rotated, -1, kernel_sharpening)
  sharpened=increase_brightness(sharpened,35)
  return sharpened
```

`increase_brightness(img, value=30)`:

- This function takes an image `img` and an optional `value` parameter to increase the brightness.
- `hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)`: The image is first converted from the BGR color space to the HSV color space.
- `h, s, v = cv2.split(hsv)`: The `split()` function is used to separate the hue, saturation, and value channels of the image.
- `lim = 255 - value`: `lim` is a threshold value for the brightness.
- `v[v > lim] = 255`: If the pixel value in the value channel is greater than `lim`, set it to the maximum value of 255.
- `v[v <= lim] += value`: If the pixel value in the value channel is less than or equal to `lim`, add the `value` parameter to it.
- `final_hsv = cv2.merge((h, s, v))`: The `merge()` function is used to merge the modified hue, saturation, and value channels back into a single image in the HSV color space.
- `img = cv2.cvtColor(final_hsv, cv2.COLOR_HSV2BGR)`: Finally, the image is converted back to the BGR color space.

`final_image(rotated)`:

- This function takes a rotated image as input.
- `kernel_sharpening = np.array([[0,-1,0], [-1, 5,-1], [0,-1,0]])`: A sharpening kernel is created as a 3x3 numpy array with the center element having the highest weight (5).
- `sharpened = cv2.filter2D(rotated, -1, kernel_sharpening)`: The sharpening kernel is applied to the input image using the `filter2D()` function in OpenCV.
- `sharpened=increase_brightness(sharpened,35)`: The `increase_brightness()` function is called to increase the brightness of the image.
- `return sharpened`: The final sharpened and brightened image is returned.

```
cleaned_image = final_image(scan(img))
cv2.immshow("Image",cleaned_image)
cv2.waitKey()
```

Finally show the Image.

```
gray = cv2.cvtColor(cleaned_image, cv2.COLOR_BGR2GRAY)
```

This line of code converts an input image `cleaned_image` from the BGR (blue-green-red) color space to grayscale using the `cv2.cvtColor()` function. The resulting grayscale image is stored in the `gray` variable.

```
thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
[1]
```

This line of code applies a thresholding operation to the grayscale image `gray` using the `cv2.threshold()` function. The thresholding operation is used to segment the image into foreground and background regions based on a threshold value. The `cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU` flags used in the `cv2.threshold()` function indicate that the threshold value should be automatically calculated using Otsu's method, and that the output image should be a binary image with foreground pixels set to 255 and background pixels set to 0, but with the binary values inverted.

The thresholded image is stored in the `thresh` variable.

```
inverted_image = cv2.bitwise_not(thresh)
```

This line of code inverts the binary image `thresh` using the `cv2.bitwise_not()` function. This is done because in the previous step, the binary image was inverted, and we want to restore the original orientation of the image. The resulting inverted binary image is stored in the `inverted_image` variable.

Overall, these three lines of code can be used to perform image segmentation, which is a process of dividing an image into multiple segments or regions, with the goal of simplifying or changing the representation of an image into something more meaningful and easier to analyze.

```
gray = inverted_image.copy()
```

This line of code creates a copy of the inverted binary image `inverted_image` and stores it in a new variable called `gray`. This is done because the subsequent operations modify the image, and we want to keep the original image intact.

```
thresh = cv2.adaptiveThreshold(gray, 255, cv2.ADAPTIVE_THRESH_MEAN_C,
cv2.THRESH_BINARY_INV, 21, 5)
```

This line of code applies adaptive thresholding to the grayscale image `gray` using the `cv2.adaptiveThreshold()` function. Adaptive thresholding is a technique for thresholding images where the threshold value is calculated for each pixel based on a small region around it, instead of using a single global threshold value for the entire image.

The `255` value is the maximum pixel value that can be assigned, `cv2.ADAPTIVE_THRESH_MEAN_C` is the adaptive thresholding method, `cv2.THRESH_BINARY_INV` is the thresholding type, and `21` and `5` are the block size and constant value used for calculating the threshold.

The thresholded image is stored in the `thresh` variable.

```
kernel = np.ones((3, 3), np.uint8)
opened = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel)
```

This line of code performs morphological opening on the thresholded image `thresh` using the `cv2.morphologyEx()` function. Morphological opening is a technique for removing noise and small gaps in an image by eroding and then dilating the image. The `np.ones((3, 3), np.uint8)` code creates a 3x3 square kernel that is used for the morphological opening operation.

The resulting opened image is stored in the `opened` variable.

```
contours, hierarchy = cv2.findContours(opened, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
```

This line of code finds contours in the opened image `opened` using the `cv2.findContours()` function. Contours are curves that connect continuous points along the boundary of an object in an image.

The `cv2.RETR_EXTERNAL` flag retrieves only the external contours, and the `cv2.CHAIN_APPROX_SIMPLE` flag compresses horizontal, vertical, and diagonal segments and leaves only their end points. The resulting contours and hierarchy are stored in the `contours` and `hierarchy` variables, respectively.

Overall, these lines of code are used to detect and extract the contours of bubbles in an image. This can be useful in various applications such as optical mark recognition (OMR) systems for grading multiple choice tests, and image-based survey or form processing.

```
# Loop over the contours
for contour in contours:
    # Calculate the ratio of the contour area to its bounding box area
    area = cv2.contourArea(contour)
    x, y, w, h = cv2.boundingRect(contour)
    box_area = w * h
    ratio = area / box_area

    # If the ratio is close to 1, the contour is likely a circular bubble
    if ratio > 0.5:
        # Create a mask for the bubble
        mask = np.zeros_like(opened)
        cv2.drawContours(mask, [contour], 0, 255, -1)

        # Apply the mask to the thresholded image to isolate the bubble
        masked_img = cv2.bitwise_and(opened, mask)

        # Calculate the mean pixel value inside the masked bubble
        mean_val = cv2.mean(gray, mask=masked_img)[0]

        # Mark the bubble with a green rectangle if it's dark, otherwise mark
with a red rectangle
        if mean_val < 150:
            cv2.rectangle(cleaned_image, (x, y), (x + w, y + h), (0, 255, 0),
2)
```

```
area = cv2.contourArea(contour)
x, y, w, h = cv2.boundingRect(contour)
box_area = w * h
ratio = area / box_area
```

These lines calculate the ratio of the contour area to its bounding box area. The `cv2.contourArea()` function calculates the area of the contour, and the `cv2.boundingRect()` function calculates the bounding box around the contour, represented by the `x`, `y`, `w`, and `h` variables. The `box_area` variable is calculated by multiplying the width and height of the bounding box. The `ratio` variable is then calculated as the ratio of the contour area to the bounding box area.

```
if ratio > 0.5:
    mask = np.zeros_like(opened)
    cv2.drawContours(mask, [contour], 0, 255, -1)
```

This line checks if the `ratio` value is greater than 0.5, which is used as a threshold to determine if the contour is likely a circular bubble.These lines create a mask for the bubble using the `np.zeros_like()` function to create a black image with the same shape as the `opened` image, and then use the `cv2.drawContours()` function to draw the contour onto the mask. The `0` index in `[contour]` refers to the index of the contour to draw, and the `-1` value for the thickness fills the contour with white color.

```
masked_img = cv2.bitwise_and(opened, mask)
```

This line applies the mask to the thresholded image `opened` using the `cv2.bitwise_and()` function to isolate the bubble.

```
mean_val = cv2.mean(gray, mask=masked_img)[0]
```

This line calculates the mean pixel value inside the masked bubble using the `cv2.mean()` function. The `gray` image is used as the source image, and the `mask` is applied to limit the calculation to only the pixels inside the bubble. The `[0]` index is used to extract the mean value from the resulting tuple.

```
if mean_val < 150:
    cv2.rectangle(cleaned_image, (x, y), (x + w, y + h), (0, 255, 0), 2)
```

These lines mark the bubble with a green rectangle if the mean pixel value inside the bubble is less than 150, indicating a dark bubble. If the mean value is greater than or equal to 150, indicating a light bubble, a red rectangle is used instead. The `cv2.rectangle()` function is used to draw the rectangle onto the `cleaned_image` variable using the `(x, y)` and `(x + w, y + h)` coordinates of the bounding box, and the `(0, 255, 0)` or `(0, 0, 255)` color tuple depending on the mean value. The `2` value sets the thickness of the rectangle to 2 pixels.