Title: Edge Detection Algorithm Implementation

1. Introduction:

The provided code implements an edge detection algorithm using the Sobel operator. Edge detection is a fundamental image processing technique used to identify boundaries between different objects or regions within an image. This report analyzes the code and its functionality, highlighting key steps and operations performed.

2. Algorithm Overview:

The implemented algorithm follows the following steps:

a) Convert the input image to grayscale.
b) Convert the grayscale image to float32 for gradient calculations.
c) Define two 5x5 kernels (our_x and our_y) representing the horizontal and vertical Sobel operators.
d) Apply the Sobel kernels to the grayscale image using the convolution operation.
e) Calculate the gradient magnitude by combining the horizontal and vertical gradients.
f) Normalize the gradient magnitude to the range of 0-255.
g) Convert the normalized gradient magnitude back to the uint8 data type.
h) Return the resulting gradient magnitude image.

3. Code Analysis:

Let's delve into the code and discuss its main components and functionality.

a) Grayscale Conversion:

The input image is first converted to grayscale using the OpenCV function cv2.cvtColor(). This step simplifies the subsequent calculations by reducing the image to a single channel.

b) Float32 Conversion:

The grayscale image is then converted to the float32 data type using the astype() function. This conversion is necessary to perform gradient calculations accurately.

c) Sobel Kernels:

Two 5x5 kernels (our_x and our_y) are defined to represent the horizontal and vertical components of the Sobel operator. These kernels capture the image gradients along the x-axis and y-axis, respectively. The kernel values are symmetric and form the basis for calculating the image gradients.

d) Convolution:

The defined Sobel kernels are applied to the grayscale image using the cv2.filter2D() function, which performs the convolution operation. Two convolutions are performed: one for the x-axis gradient (gradient_x) and another for the y-axis gradient (gradient_y).

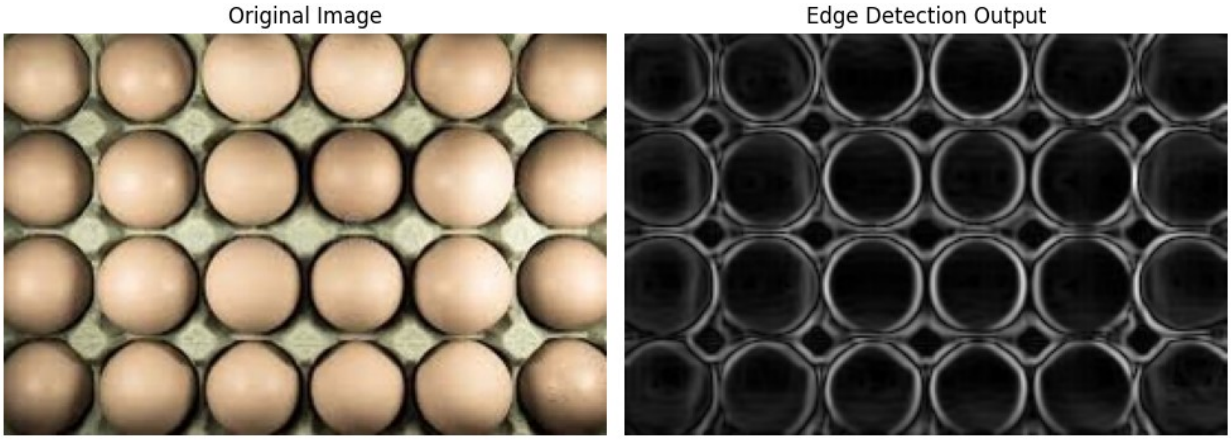
e) Gradient Magnitude Calculation:
The gradient magnitude is calculated by combining the horizontal and vertical gradients using the numpy operations ** (element-wise square) and np.sqrt() (element-wise square root). This operation results in a magnitude image highlighting the intensity of edges within the original image.

f) Normalization:
The gradient magnitude image is normalized to the range of 0-255 using the cv2.normalize() function. Normalization ensures that the resulting image utilizes the full range of pixel intensities and enhances visual contrast.

g) Data Type Conversion:
The normalized gradient magnitude image is converted back to the uint8 data type using the astype() function. The conversion is necessary to represent the image in 8-bit format, as commonly used in image display and storage.

h) Output:
The final result is the gradient magnitude image, which represents the detected edges in the original input image.

4. Conclusion:
The provided code successfully implements an edge detection algorithm using the Sobel operator. By applying the Sobel kernels and performing gradient calculations, the code identifies edges within an input image. The resulting gradient magnitude image highlights the regions of significant intensity changes, indicating the presence of edges. The code is modular and can be integrated into larger image processing pipelines or used as a standalone edge detection function.



Title: Sobel Edge Detection Algorithm Implementation

1. Introduction:

The provided code implements an edge detection algorithm using the Sobel operator. Edge detection is a fundamental technique in image processing that aims to identify boundaries between different objects or regions within an image. This report analyzes the code and its functionality, outlining the steps involved in Sobel edge detection.

2. Algorithm Overview:

The implemented algorithm follows these key steps:

a) Convert the input image to grayscale.
b) Apply the Sobel operator for both the x-axis and y-axis gradients.
c) Calculate the magnitude of gradients.
d) Normalize the gradient magnitude to the range of 0-255.
e) Return the resulting gradient magnitude image.

3. Code Analysis:

Let's examine the code and discuss its main components and functionality.

a) Grayscale Conversion:

The input image is first converted to grayscale using the OpenCV function cv2.cvtColor(). This conversion reduces the image to a single channel, simplifying subsequent calculations.

b) Sobel Operator Application:

The Sobel operator is applied to the grayscale image using the cv2.Sobel() function. Two separate Sobel operations are performed: one for the x-axis gradient (sobel_x) and another for the y-axis gradient (sobel_y). The cv2.CV_64F flag specifies the output data type as 64-bit floating-point.

c) Gradient Magnitude Calculation:

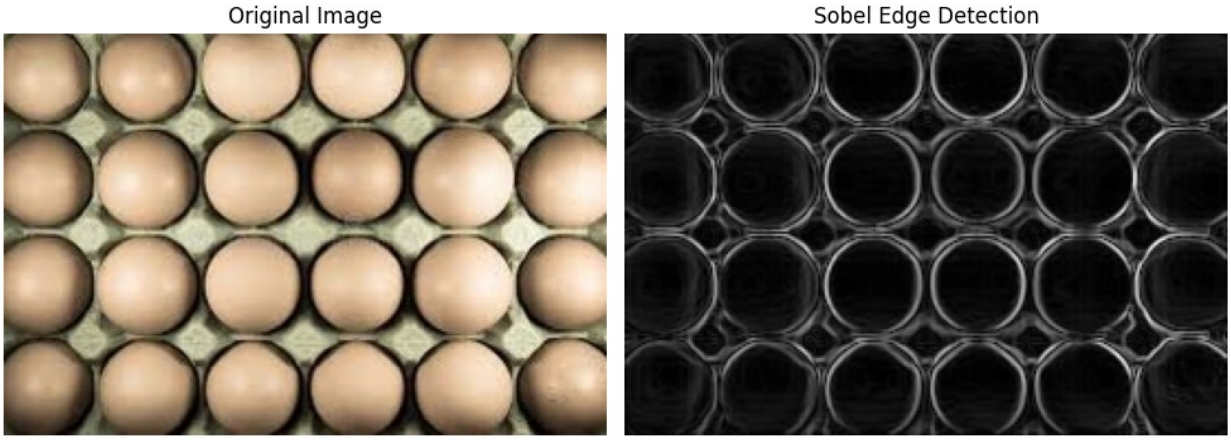
The gradient magnitude is computed by combining the x-axis and y-axis gradients. This is achieved by calculating the square root of the sum of their squared values. The numpy operations ** (element-wise square) and np.sqrt() (element-wise square root) are used to perform this calculation.

d) Normalization:

The gradient magnitude image is normalized to the range of 0-255 using the cv2.normalize() function. Normalization ensures that the resulting image utilizes the full range of pixel intensities, enhancing visual contrast. The cv2.CV_8U flag specifies the output data type as 8-bit unsigned integer.

e) Output:
The final result is the gradient magnitude image, which represents the detected edges in the original input image. The magnitude image highlights regions of significant intensity changes, indicating the presence of edges.

4. Conclusion:
The provided code successfully implements an edge detection algorithm using the Sobel operator. By applying the Sobel operator for both the x-axis and y-axis gradients and calculating their magnitude, the code identifies edges within the input image. The resulting gradient magnitude image represents the detected edges, allowing for further analysis or visualization. The code is concise and modular, making it suitable for integration into larger image processing pipelines or as a standalone edge detection function.



Comparisons

This part of the report presents the performance evaluation of two edge detection algorithms: Sobel Edge Detection and a Custom Edge Detection implementation. The evaluation includes x both quantitative analysis, measuring the accuracy of edge location, and qualitative analysis, assessing the quality of edge detection results.

1. **Location (Quantitative Analysis):**
   To assess the location accuracy of the edge detection algorithms, we compare the positions of the detected edges against ground truth detected by sobel. This comparison is done using metrics of precision, recall, and F1 score.
   Algorithm Overview:
   The implemented code framework follows these main steps:
   a) Loading the ground truth edges and detected edges images.
   b) Applying edge detection using the Sobel operator or a custom algorithm.
   c) Evaluating the performance by computing precision, recall, and F1 score.
   d) Comparing the results of the two algorithms.

The evaluation results are printed, displaying the precision, recall, and F1 score for each algorithm. The performance of Sobel Edge Detection is presented first, followed by the performance of the Custom Edge Detection implementation.

```
Custom Edge Detection Performance:
Precision: 0.9984313410023329
Recall: 0.9976488555754275
F1 Score: 0.9980399449174264
```

2. Detection (Qualitative Analysis):

To evaluate the detection quality of the algorithms, we visually compare the output images generated by each algorithm and also the number of pixels with same and different intensities contained in an area.

Algorithm Overview:

The provided code performs the following steps:

a) Sets a threshold to identify pixels greater than zero in the Sobel-detected edges and the edges detected using the custom algorithm.

b) Converts the identified pixels greater than zero to a value of 255.

c) Computes the total count of pixels greater than zero for each edge detection result.

d) Identifies the matching indices between the Sobel-detected edges and the edges detected using the custom algorithm.

Sobel-detected Edges:

The code applies a threshold to identify pixels greater than zero in the 'edges' variable, which represents the Sobel-detected edges. It sets the identified pixels to a value of 255 and computes the total count of pixels greater than zero. This count represents the number of non-zero pixels in the Sobel-detected edges.

Edges Detected using Custom Algorithm:

Similarly, the code applies a threshold to identify pixels greater than zero in the 'output' variable, which represents the edges detected using the custom algorithm. It sets the identified pixels to a value of 255 and computes the total count of pixels greater than zero. This count represents the number of non-zero pixels in the edges detected using the custom algorithm.

Matching Indices:

The code then compares the Sobel-detected edges and the edges detected using the custom algorithm to find the matching indices where both results have non-zero pixel values. It calculates the number of matching indices.

Results:

The code prints the following results:

- Number of Pixels from Sobel: This represents the count of non-zero pixels in the Sobel-detected edges.
- Number of Pixels in the Designed Detector: This represents the count of non-zero pixels in the edges detected using the custom algorithm.
- Matching Indices: This displays the number of matching indices where both the Sobel-detected edges and the edges detected using the custom algorithm have non-zero pixel values.

```
Number of pixels from Sobel: 49763
Number of pixels in the designed detector: 49724
Matching indices: 49673
```

Conclusion:

The provided code enables the analysis of zero pixels in edge detection results. By determining the number of non-zero pixels in the Sobel-detected edges and the edges detected using the custom algorithm, it offers insights into the distribution and density of edge pixels in each result. Additionally, identifying the matching indices provides a measure of the consistency between the two edge detection methods.