TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN GIỮA KÌ MÔN NHẬP MÔN XỬ LÝ ẢNH SỐ

XỬ LÝ ẢNH SỐ

Người thực hiện: **NGUYỄN QUỐC HƯNG – 52100551**
Lớp **: 21050281**
Khoá **: 25**

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2024

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA CÔNG NGHỆ THÔNG TIN

ĐỒ ÁN GIỮA KÌ MÔN NHẬP MÔN XỬ LÝ ẢNH SỐ

XỬ LÝ ẢNH SỐ

Người hướng dẫn: **THẦY TRẦN LƯƠNG QUỐC ĐẠI**

Người thực hiện: **NGUYỄN QUỐC HƯNG – 52100551**

Lớp **: 21050281**

Khoá **: 25**

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2024

LỜI CẢM ƠN

Lời đầu tiên, chúng em xin chân thành gửi lời cảm ơn tới Trường Đại học Tôn Đức Thắng nói chung, khoa Công Nghệ Thông Tin nói riêng đã đưa môn Xử lý ảnh số vào trong chương trình giảng dạy. Đặc biệt, chúng em chân thành cảm ơn và lòng biết ơn sâu sắc tới thầy Trần Lương Quốc Đại đã mang đến cho chúng em những kiến thức bổ ích, không chỉ về chuyên môn mà còn là những kĩ năng mềm thiết yếu. Cùng những tinh hoa kiến thức đó, chúng em đã cải thiện được bản thân cả trong học tập lẫn đời sống.

Trong suốt thời gian làm bài báo cáo giữa kỳ môn Xử lý ảnh số, chúng em đã từng gặp nhiều vấn đề về lý thuyết trên lớp. Nhưng nhờ có sự giảng dạy tận tình của thầy mà mọi khó khăn trong việc học của chúng em đã được giải đáp. Tuy nhiên, vì kiến thức chuyên môn còn hạn chế và còn nhiều thiếu sót, chúng em rất mong nhận được sự góp ý, chỉ bảo thêm từ thầy để hoàn thiện những kiến thức còn thiếu của mình, để chúng em có thể dùng làm hành trang thực hiện tiếp các bài báo cáo tiếp theo và sau này, cũng như là trong học tập hoặc làm việc trong tương lai.

Lời cuối, một lần nửa xin gửi đến thầy và khoa lời cảm ơn chân thành và tốt đẹp nhất!

TP. Hồ Chí Minh, ngày 7 tháng 11 năm 2024

Tác giả

(Ký tên và ghi rõ họ tên)

Hưng

Nguyễn Quốc Hưng

ĐỒ ÁN ĐƯỢC HOÀN THÀNH TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Tôi xin cam đoan đây là sản phẩm đồ án của riêng tôi / chúng tôi và được sự hướng dẫn của Thầy Trần Lương Quốc Đại;. Các nội dung nghiên cứu, kết quả trong đề tài này là trung thực và chưa công bố dưới bất kỳ hình thức nào trước đây. Những số liệu trong các bảng biểu phục vụ cho việc phân tích, nhận xét, đánh giá được chính tác giả thu thập từ các nguồn khác nhau có ghi rõ trong phần tài liệu tham khảo.

Ngoài ra, trong đồ án còn sử dụng một số nhận xét, đánh giá cũng như số liệu của các tác giả khác, cơ quan tổ chức khác đều có trích dẫn và chú thích nguồn gốc.

Nếu phát hiện có bất kỳ sự gian lận nào tôi xin hoàn toàn chịu trách nhiệm về nội dung đồ án của mình. Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

TP. Hồ Chí Minh, ngày 7 tháng 11 năm 2024

Tác giả

(ký tên và ghi rõ họ tên)

Hung

Nguyễn Quốc Hưng

PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN

Phần xác nhận của GV hướng dẫn

_____

_____

_____

_____

_____

_____

_____

Tp. Hồ Chí Minh, ngày tháng năm
(kí và ghi họ tên)

Phần đánh giá của GV chấm bài

_____

_____

_____

_____

_____

_____

_____

Tp. Hồ Chí Minh, ngày tháng năm
(kí và ghi họ tên)

TÓM TẮT

Bài báo cáo này sẽ có 2 nội dung chính. Một là phân tích các bước và thuật toán thực hiện chương trình. Hai là tổng hợp kết quả đạt được sau quá trình nghiên cứu.

Đối với nội dung thứ nhất, các nội dung sẽ được đề cập đến là: Nghiên cứu cơ sở lý thuyết cơ bản được sử dụng và trình bày/mô tả các bước thuật toán, giải thích cách thư viện OpenCV được sử dụng để thực hiện yêu cầu.

Phần nội dung thứ hai sẽ cho thấy kết quả đạt được sau khi trình bày và áp dụng thư viện OpenCV để giải quyết.

MỤC LỤC

DANH MỤC KÍ HIỆU VÀ CHỮ VIẾT TẮT

CHƯƠNG 1. PHƯƠNG PHÁP THỰC HIỆN

1. Kiến thức cơ bản được sử dụng

1.1. Giới thiệu thư viện OpenCV

Project **OpenCV** (Open Source Computer Vision Library) được bắt đầu từ Intel năm 1999 bởi Gary Bradsky. OpenCV viết tắt cho Open Source Computer Vision Library. OpenCV là thư viện nguồn mở hàng đầu cho Computer Vision và Machine Learning, và hiện có thêm tính năng tăng tốc GPU cho các hoạt động theo real-time.

OpenCV được phát hành theo giấy phép BSD (*), do đó nó miễn phí cho cả học tập và sử dụng với mục đích thương mại. Nó có trên các giao diện C++, C, Python và Java và hỗ trợ Windows, Linux, Mac OS, iOS và Android. OpenCV được thiết kế để hỗ trợ hiệu quả về tính toán và chuyên dùng cho các ứng dụng real-time (thời gian thực).

OpenCV có một cộng đồng người dùng khá hùng hậu hoạt động trên khắp thế giới bởi nhu cầu cần đến nó ngày càng tăng theo xu hướng chạy đua về sử dụng computer vision của các công ty công nghệ. OpenCV hiện được ứng dụng rộng rãi toàn cầu, với cộng đồng hơn 47.000 người, với nhiều mục đích và tính năng khác nhau từ interactive art, đến khai thác mỏ, khai thác web map hoặc qua robotic cao cấp.

OpenCV (Open Source Computer Vision Library) được sử dụng để xử lý ảnh và video trong các ứng dụng máy tính. Thư viện này chủ yếu được sử dụng trong các lĩnh vực như nhận diện hình ảnh, thị giác máy tính, phân tích video và xử lý dữ liệu hình ảnh. OpenCV cung cấp một loạt các công cụ giúp dễ dàng triển khai các thuật toán phức tạp về hình ảnh, video và các tác vụ thị giác máy tính khác.

OpenCV được sử dụng cho đa dạng nhiều mục đích và ứng dụng khác nhau bao gồm:

- Hình ảnh street view
- Kiểm tra và giám sát tự động
- Robot và xe hơi tự lái
- Phân tích hình ảnh y học
- Tìm kiếm và phục hồi hình ảnh/video
- Phim – cấu trúc 3D từ chuyển động
- Nghệ thuật sắp đặt tương tác

Theo tính năng và ứng dụng của OpenCV, có thể chia thư viện này thánh các nhóm tính năng:

- Xử lý và hiển thị Hình ảnh/ Video/ I/O (core, imgproc, highgui)
- Phát hiện các vật thể (objdetect, features2d, nonfree)
- Geometry-based monocular hoặc stereo computer vision (calib3d, stitching, videostab)
- Computational photography (photo, video, superres)
- Machine learning & clustering (ml, flann)
- CUDA acceleration (gpu)

Để sử dụng OpenCV đối với python ta cần cài đặt "pip install opencv-python". Khi muốn sử dụng thư viện này ta chỉ cần gọi "import cv2".

1.2. Đọc, ghi dữ liệu Video và Ảnh

1.2.1. Đọc hình ảnh

Phương thức **imread** được sử dụng để đọc ảnh bằng OpenCV trong python. Phương thức này có 2 tham số:

- **filename**: Đường dẫn đến tệp ảnh.
- **flags**: Cách thức đọc ảnh. Tham số này có thể là một trong ba giá trị:
  - **IMREAD_COLOR** (mặc định): Đọc ảnh dưới dạng màu (3 kênh: BGR).

- **IMREAD_GRAYSCALE**: Đọc ảnh dưới dạng ảnh đen trắng (1 kênh: Gray).
- **IMREAD_UNCHANGED**: Đọc ảnh cùng với alpha channel nếu có (màu và độ trong suốt).

Nếu tệp không tồn tại hoặc có lỗi trong quá trình đọc ảnh, cv2.imread() sẽ trả về None:

```
# Đọc ảnh với chế độ mặc định (màu)
img = cv2.imread('image.jpg')

# Đọc ảnh dưới dạng ảnh đen trắng
img_gray = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

if img is None:
    print("Error: Image not found")
    exit()
```

Sau khi đọc ảnh, ta có thể hiển thị ảnh bằng hàm **imshow**(). Phương thức này mở một cửa sổ hiển thị ảnh:

```
# Hiển thị ảnh
cv2.imshow('Image', img)

# Đợi nhấn phím bất kỳ để đóng cửa sổ
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Trong đó **imshow**(window_name, image) là một phương thức giúp mở cửa sổ hiển thị hình ảnh:

- **window_name**: Tên cửa sổ hiển thị ảnh.
- **image**: Ảnh cần hiển thị (dưới dạng numpy array).

waitKey(0): Đợi sự kiện nhấn phím, 0 có nghĩa là đợi vô hạn.

destroyAllWindows(): Đóng tất cả cửa sổ hiển thị ảnh khi nhấn phím bất kỳ.

Các định dạng ảnh mà OpenCV có thể đọc thông qua imread():

- **JPEG** (.jpg, .jpeg): Một trong những định dạng phổ biến nhất cho ảnh, thường dùng cho ảnh chụp và ảnh có độ nén cao.
- **PNG** (.png): Hỗ trợ ảnh có nền trong suốt (alpha channel) và không nén mất dữ liệu (lossless).
- **BMP** (.bmp): Định dạng ảnh bitmap, không nén và thường có dung lượng tệp lớn.
- **TIFF** (.tiff, .tif): Định dạng ảnh chất lượng cao, hỗ trợ nhiều kênh màu và có thể sử dụng nén mất dữ liệu hoặc không nén.
- **WebP** (.webp): Một định dạng nén ảnh với chất lượng cao và dung lượng tệp thấp, được Google phát triển.
- **PPM** (.ppm): Định dạng ảnh không nén, ít phổ biến hơn so với các định dạng khác.
- **PGM** (.pgm): Định dạng ảnh đen trắng, không nén.
- **PBM** (.pbm): Định dạng ảnh bitmap đen trắng.

1.2.2. Ghi ảnh

Ta có thể ghi hình ảnh vào tệp bằng cách sử dụng phương thức **imwrite**() của thư viện OpenCV. Phương thức này cho phép lưu ảnh dưới các định dạng phổ biến như JPEG, PNG, BMP, TIFF, và nhiều định dạng khác:

```
cv2.imwrite(filename, img)
```

Trong đó:

- **filename**: Đường dẫn tệp nơi hình ảnh được lưu, hoặc có thể là tên tệp (nếu lưu vào thư mục hiện tại) hoặc đường dẫn đầy đủ đến thư mục và tệp.
- **img**: Là anh mà ta muốn lưu. Đây là đối tượng ảnh mà ta đã đọc hoặc xử lý trong OpenCV.

```
# Lưu ảnh đã xử lý
success = cv2.imwrite('output_image_blurred.jpg', blurred_img)

if success:
    print("Ảnh đã được lưu thành công sau khi áp dụng bộ lọc.")
else:
    print("Có lỗi khi lưu ảnh.")
```

1.2.3. Đọc video

Để đọc video ta sử dụng **VideoCapture** là một lớp của thư viện OpenCV cho phép đọc video từ các nguồn như tệp video trên ổ đĩa hoặc từ camera. Lớp này có thể đọc được các video có phần mở rộng như:

- **MP4** (.mp4) - Sử dụng codec H.264 hoặc H.265, là định dạng phổ biến cho video chất lượng cao với kích thước tệp nhỏ.
- **AVI** (.avi) - Một định dạng video thường được sử dụng trên hệ điều hành Windows.
- **MOV** (.mov) - Định dạng của Apple, thường sử dụng trong hệ sinh thái macOS và iOS.
- **MKV** (.mkv) - Định dạng chứa đa phương tiện với khả năng hỗ trợ nhiều định dạng âm thanh, video và phụ đề.
- **FLV** (.flv) - Định dạng thường được sử dụng cho các video phát trực tuyến.

Khi khởi tạo đối tượng này, ta cần cung cấp:

- Đường dẫn của tệp video (nếu đọc video từ một tệp).
- **Số nguyên (0, 1, 2, …)** nếu đọc từ camera; thường 0 là camera mặc định của hệ thống.
- Cú pháp thực hiện:

```
# Đọc từ tệp video
cap = cv2.VideoCapture('video.mp4')

# Hoặc mở camera mặc định
cap = cv2.VideoCapture(0)
```

- Sau khi khởi tạo đối tượng VideoCapture, ta cần kiểm tra video đã được mở thành công chưa. Để làm việc này ta cần sử dụng phương thức isOpened của thư viện OpenCV. Phương thức này tả về True nếu video đã được mở thành công, ngược lại trả về False, cú pháp thực hiện:

```
if not cap.isOpened():
    print("Cannot open video")
    exit()
```

Khi đã mở video thành công, ta có thể đọc và xử lý từng khung hình của video bằng vòng lặp. Trong đó, với mỗi khung hình trong mỗi vòng lặp, để đọc được một khung hình ta sử dụng phương thức read. Phương thức này trả về hai giá trị là ret và frame:

  - **ret**: Một giá trị Boolean, True nếu khung hình đọc thành công và False nếu không (thường khi video kết thúc).
  - **frame**: Khung hình hiện tại dưới dạng một ảnh (dưới dạng numpy array).

```
while True:
    # Đọc khung hình tiếp theo
    ret, frame = cap.read()

    # Nếu không đọc được khung hình (cuối video), kết thúc vòng lặp
    if not ret:
        print("End of video stream")
        break

    # Hiển thị khung hình
    cv2.imshow('Video Frame', frame)

    # Nhấn phím 'q' để thoát khỏi vòng lặp
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
```

Trong đó:

- **ret, frame = cap.read()**: Đọc một khung hình từ video. Nếu ret là False, có nghĩa là không thể đọc thêm khung hình (kết thúc video hoặc gặp lỗi).

  cv2.imshow(): Hiển thị khung hình trong một cửa sổ. Tên cửa sổ là "Video Frame" và nó sẽ hiển thị nội dung của frame.
- **cv2.waitKey(1) & 0xFF == ord('q')**: Kiểm tra xem người dùng có nhấn phím 'q' không, cv2.waitKey(1) sẽ chờ 1ms giữa các khung hình và nếu phím 'q' được nhấn, vòng lặp sẽ dừng.

Khi đã hoàn tất việc đọc và hiển thị video, hoặc khi người dùng thoát chương trình, ta cần phải giải phóng tài nguyên đã sử dụng khi hiển thị video và đóng các cửa sổ hiển thị. Điều này rất quan trọng vì nếu không giải phóng tài nguyên và đóng cửa sổ, chương trình có thể tiêu tốn bộ nhớ không cần thiết và gây ra các lỗi khi chạy lại.:

- **cap.release()**: Giải phóng đối tượng VideoCapture, đóng kết nối với video hoặc camera.
- **cv2.destroyAllWindows()**: Đóng tất cả các cửa sổ OpenCV đã mở trong quá trình chạy chương trình.

1.2.4. Ghi video

Để ghi video trong OpenCV ta sử dụng lớp **VideoWriter**. Lớp này cho phép bạn ghi các khung hình (frames) vào tệp video với các định dạng phổ biến như .avi, .mp4, .mov, v.v.

Cú pháp cơ bản của VideoWriter:

```
cv2.VideoWriter(filename, fourcc, fps, frameSize)
```

- **filename**: Đường dẫn đến tệp video. Đây có thể là tên tệp (nếu muốn lưu vào thư mục hiện tại) hoặc đường dẫn đầy đủ.

- **fourcc**: Mã định dạng video (codec). Đây là một mã gồm 4 ký tự để chỉ định codec sẽ được sử dụng để nén video. Ví dụ: cv2.VideoWriter_fourcc(*'XVID') cho codec XVID.
- **fps**: Số khung hình mỗi giây (frames per second) của video. Đây là thông số quan trọng vì nó ảnh hưởng đến tốc độ video.
- **frameSize**: Kích thước của từng khung hình, thể hiện dưới dạng tuple (width, height). Đảm bảo kích thước này khớp với kích thước của khung hình bạn muốn ghi.

Khi ta sử dụng VideoWriter, ta cần chỉ định codec (compressor-decompressor) để mã hóa video. Nhằm đảm bảo tệp video có kích thước nhỏ, chất lượng tốt và dễ dàng chia sẻ, phát lại. Nếu không sử dụng codec, video sẽ không được nén và có thể gây ra nhiều vấn đề về kích thước và khả năng tương thích:

- **XVID** là một codec nén video theo chuẩn MPEG-4, phổ biến trong các video chia sẻ trên internet. Nó cung cấp nén hiệu quả với chất lượng tốt. Phù hợp cho việc chia sẻ, lưu trữ video vừa phải.
- **DIVX** là một codec nén video mạnh mẽ, cũng dựa trên chuẩn MPEG-4. DIVX được phát triển để nén video có độ phân giải cao mà vẫn giữ được chất lượng. Phù hợp cho video chất lượng cao.
- **MJPG** là codec nén video theo dạng JPEG, nén mỗi khung hình của video như một ảnh JPEG riêng biệt. Đây là codec đơn giản và ít hiệu quả nhất trong các codec hiện có. Phù hợp cho video đơn giản, yêu cầu nén nhanh.
- **MP4V** là codec video nén theo chuẩn MPEG-4. Đây là một codec phổ biến và hỗ trợ tốt cho nhiều loại video. Phù hợp cho video ổn định, dễ tương thích.

- **H.264** (hoặc AVC - Advanced Video Coding) là codec video được sử dụng rộng rãi nhất hiện nay, đặc biệt cho các video chất lượng cao và nén hiệu quả. Phù hợp cho video chất lượng cao, phát trực tuyến.

Các bước để ghi một video:

- Khởi tạo VideoWriter với các tham số phù hợp.
- Đọc hoặc tạo các khung hình video từ camera, tệp video, hoặc tạo các khung hình bằng cách sử dụng các thao tác xử lý hình ảnh.
- Ghi khung hình vào video bằng phương thức write() của VideoWriter để ghi mỗi khung hình vào tệp video.
- Giải phóng tài nguyên sau khi hoàn thành việc ghi video, bằng release().

```
# Lấy kích thước của khung hình (width, height)
frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

# Tạo đối tượng VideoWriter để ghi video
# Sử dụng codec XVID, 20 fps và kích thước khung hình được lấy từ webcam
fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter('output_video.avi', fourcc, 20.0, (frame_width,
frame_height))

while True:
    # Đọc khung hình từ webcam
    ret, frame = cap.read()

    if not ret:
        print("Không thể đọc khung hình.")
        break

    # Ghi khung hình vào video
    out.write(frame)

# Giải phóng tài nguyên
cap.release()
out.release()
cv2.destroyAllWindows()
```

Trong đó:

- get(cv2.CAP_PROP_FRAME_WIDTH) và get(cv2.CAP_PROP_FRAME_HEIGHT) để lấy chiều rộng và chiều cao của khung hình mà webcam/video cung cấp.
- VideoWriter('output_video.avi', fourcc, 20.0, (frame_width, frame_height)) tạo đối tượng VideoWriter để ghi video với codec XVID, 20 khung hình mỗi giây (fps) và kích thước khung hình phù hợp với webcam/video.
- Trong vòng lặp, khung hình được đọc từ webcam bằng cap.read() và sau đó ghi vào tệp video bằng out.write(frame).
- cap.release() và out.release() để giải phóng camera và đối tượng ghi video sau khi hoàn thành. cv2.destroyAllWindows() đóng tất cả cửa sổ OpenCV.

1.3. Chuyển đổi không gian màu

Trong OpenCV có nhiều không gian màu khác nhau được sử dụng, thông thường sẽ sử dụng các không gian màu:

- **RGB** (Red, Green, Blue): Là không gian màu phổ biến nhất, đại diện cho các ảnh kỹ thuật số với ba kênh màu đỏ, xanh lá, và xanh dương. Mỗi kênh có giá trị từ 0 đến 255, tạo thành một tổ hợp màu.
- **Grayscale** (Ảnh xám): Là ảnh có một kênh duy nhất, giá trị từ 0 đến 255, biểu thị độ sáng của ảnh. Các giá trị thấp gần 0 là màu đen, cao gần 255 là màu trắng.
- **HSV** (Hue, Saturation, Value): HSV là không gian màu dựa trên sắc độ, độ bão hòa, và giá trị sáng. Đây là không gian màu thuận lợi để thao tác trên sắc độ, hữu ích cho việc phát hiện và phân loại màu:
  - **Hue (H)**: xác định màu sắc (0-180 trong OpenCV)

- **Saturation (S)**: xác định độ bão hòa màu (mức độ nhạt hay đậm của màu, từ 0-255)
- **Value (V)**: xác định độ sáng của màu (từ 0-255)

Thư viện OpenCV sử dụng hàm **cvtColor**() để chuyển đổi giữa các không gian màu, cú pháp:

```
import cv2

image = cv2.imread('path_to_image.jpg')  # Đọc ảnh màu
converted_image = cv2.cvtColor(image, code)  # Chuyển đổi không gian màu
```

Trong đó:

- **image** là ảnh đầu vào.
- **code** là mã chuyển đổi màu từ không gian này sang không gian khác.

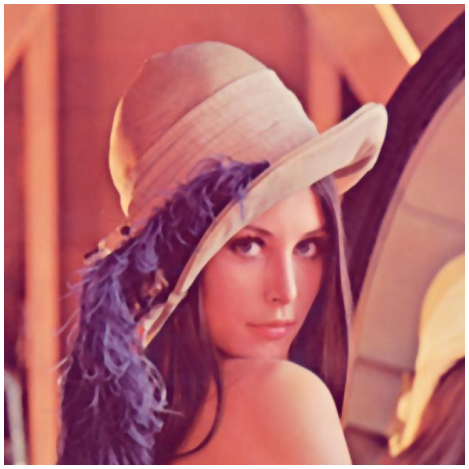
Các mã (code) chuyển đổi thường gặp bao gồm:

- **COLOR_BGR2GRAY**: Chuyển từ BGR (OpenCV mặc định đọc ảnh theo thứ tự BGR) sang Grayscale.
- **COLOR_BGR2HSV**: Chuyển từ BGR sang HSV.
- **COLOR_HSV2BGR**: Chuyển từ HSV về BGR.

Để chuyển ảnh từ không gian màu BGR sang Grayscale, dùng mã COLOR_BGR2GRAY:

```
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow('Grayscale Image', gray_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Kết quả:



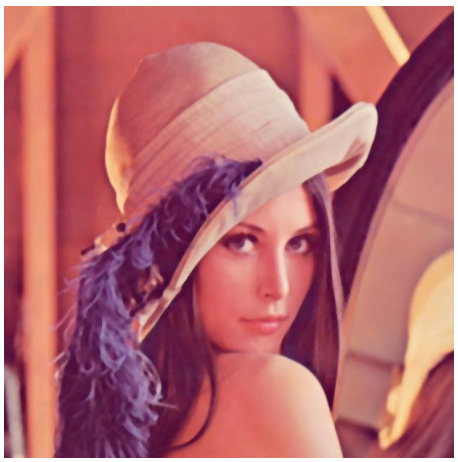
Hình 1: Ảnh gốc

Hình 2: Ảnh xám

Để chuyển đổi sang không gian HSV, dùng mã COLOR_BGR2HSV:

```
hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
cv2.imshow('HSV Image', hsv_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Kết quả:



Hình 1: Ảnh gốc

Hình 2: Ảnh HSV

Với ảnh BGR, ta có thể tách riêng các kênh B, G, R:

```
image = cv2.imread('path_to_image.png')
# Tách các kênh màu B, G, R
B, G, R = cv2.split(image)
# Tạo ảnh với từng kênh màu riêng biệt (cho các kênh không cần thiết bằng 0)
zeros = np.zeros_like(B)  # Ảnh đen để thay thế cho các kênh không sử dụng
# Ảnh kênh xanh dương (B)
blue_channel = cv2.merge([B, zeros, zeros])
# Ảnh kênh xanh lá (G)
green_channel = cv2.merge([zeros, G, zeros])
# Ảnh kênh đỏ (R)
red_channel = cv2.merge([zeros, zeros, R])
# Hiển thị ảnh gốc và các kênh màu với màu sắc đúng
cv2.imshow('Original Image', image)
cv2.imshow('Blue Channel', blue_channel)
cv2.imshow('Green Channel', green_channel)
cv2.imshow('Red Channel', red_channel)
# Đợi cho đến khi có phím được nhấn rồi đóng tất cả cửa sổ
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Trong đó:

```
B, G, R = cv2.split(image)
```

- Hàm cv2.split() tách ảnh image thành ba kênh màu riêng biệt: B (kênh xanh dương), G (kênh xanh lá), và R (kênh đỏ). Mỗi kênh này là một mảng 2D lưu trữ cường độ màu tại mỗi pixel.

```
zeros = np.zeros_like(B)
```

- zeros là một mảng 2D cùng kích thước với kênh B, nhưng tất cả giá trị đều bằng 0. Mảng này dùng để thay thế các kênh không cần thiết khi tạo ảnh chỉ chứa một kênh màu (đỏ, xanh lá, hoặc xanh dương).

```
blue_channel = cv2.merge([B, zeros, zeros])
```

- cv2.merge() kết hợp ba kênh lại để tạo một ảnh mới, trong đó kênh xanh dương (B) giữ nguyên giá trị, còn các kênh khác (zeros cho xanh lá và đỏ) đều có giá trị 0.

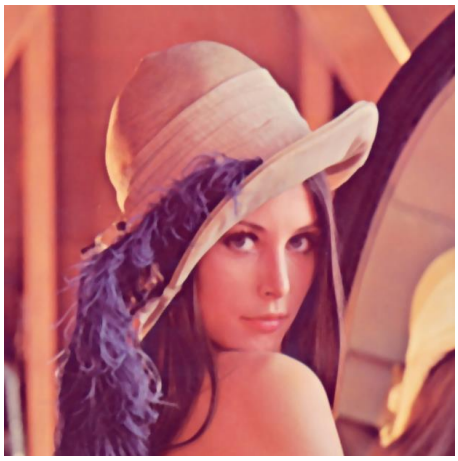
```
green_channel = cv2.merge([zeros, G, zeros])
```

- Chỉ kênh xanh lá (G) giữ nguyên, còn kênh xanh dương và đỏ được thay thế bằng zeros.

```
red_channel = cv2.merge([zeros, zeros, R])
```

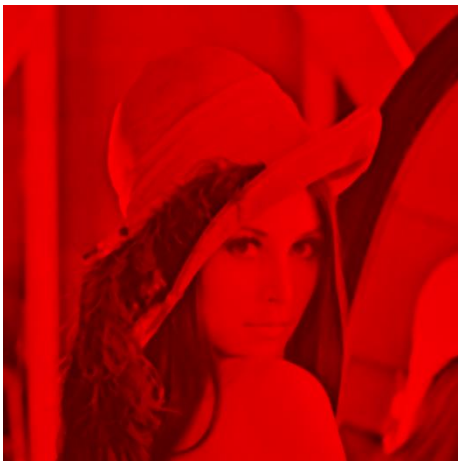
- Chỉ kênh đỏ (R) giữ nguyên, còn kênh xanh dương và xanh lá được thay thế bằng zeros.

Kết quả:

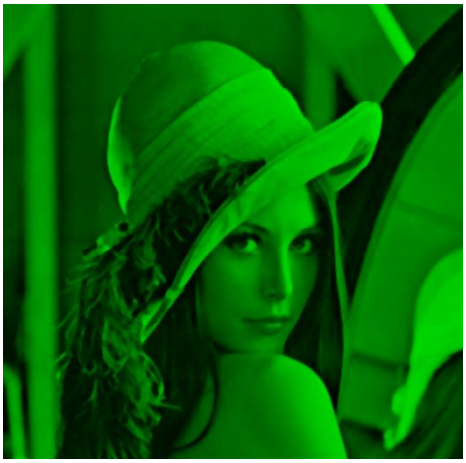


Hình 1: Ảnh gốc

Hình 2: Kênh B



Hình 3: Kênh R



Hình 4: Kênh G

1.4. Tạo mặt nạ màu

"Tạo mặt nạ màu" (color masking) là quá trình tạo một mặt nạ (mask) để chỉ giữ lại những phần của hình ảnh có màu sắc nằm trong một khoảng màu nhất định, đồng thời lọc bỏ các phần không nằm trong khoảng màu đó. Đây là

một kỹ thuật phổ biến trong xử lý hình ảnh để tách đối tượng hoặc vùng màu cụ thể trong hình ảnh.

Một số bước để tạo mặt nạ màu cho hình ảnh:

- Màu sắc trong ảnh thường được dễ dàng xác định trong không gian màu HSV (Hue-Saturation-Value) hơn so với không gian màu RGB. Vậy nên ta sẽ chuyển đổi hình ảnh sang không gian màu HSV.
- Để tạo mặt nạ ta cần xác định các ngưỡng màu cần lọc, cách xác định là ta sẽ tạo các khoảng giá trị HSV cho màu mà ta muốn lọc. Trong trường hợp ta muốn lọc lấy các vùng có màu xanh lá trong ảnh ta có thể xác định một khoảng màu bằng cách thiết lập giá trị lower_bound và under_bound:

```
# Đặt khoảng màu để lọc (ví dụ: màu xanh lá cây)
lower_bound = (36, 50, 70)  # giá trị HSV dưới cùng
upper_bound = (89, 255, 255) # giá trị HSV trên cùng
```

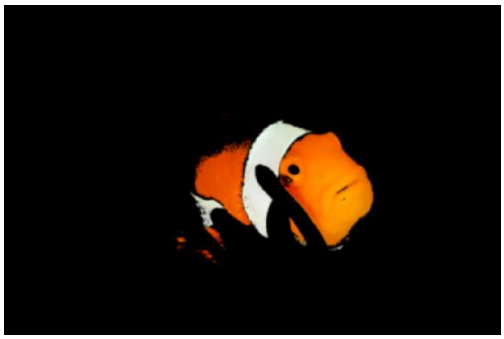
- Ta sử dụng **inRange**() của OpenCV để tạo mặt nạ, khi đó hình ảnh sẽ chỉ giữ lại các pixel trong khoảng màu đã xác định:

```
# Tạo mặt nạ màu
mask = cv2.inRange(hsv_image, lower_bound, upper_bound)
```

Kết quả là một hình ảnh nhị phân (đen trắng), trong đó các pixel thuộc phạm vi màu xác định sẽ có giá trị 255 (trắng), và các pixel ngoài phạm vi sẽ có giá trị 0 (đen).

- Sau khi tạo mặt nạ, ta có thể áp dụng mặt nạ này lên hình ảnh gốc để giữ lại chỉ những phần có màu xác định:

```
# Áp dụng mặt nạ lên ảnh gốc để chỉ giữ lại các vùng màu
masked image = cv2.bitwise and(image, image, mask=mask)
```



Ảnh gốc *Ảnh sau khi tạo mặt nạ màu*

Mặt nạ màu thường được sử dụng trong các ứng dụng như:

- Tách nền ảnh hoặc đối tượng
- Theo dõi đối tượng màu (ví dụ: bóng, phương tiện giao thông)
- Phát hiện biển báo giao thông hoặc các vật thể dựa vào màu sắc

1.5. Phân ngưỡng

Phân ngưỡng là một kỹ thuật quan trọng trong xử lý ảnh số, giúp phân đoạn ảnh dựa trên giá trị cường độ sáng của từng pixel. Mục tiêu của phân ngưỡng là tách đối tượng khỏi nền trong ảnh, dựa vào sự khác biệt về màu sắc hoặc độ sáng.

Phân ngưỡng cơ bản thường là quá trình biến đổi ảnh xám thành ảnh nhị phân. Dựa vào giá trị ngưỡng TTT, mỗi pixel của ảnh sẽ được so sánh với ngưỡng này để xác định là nền hay đối tượng. Quá trình này có thể biểu diễn như sau:

$$f(x,y) = \begin{cases} 255 & \text{nếu } I(x,y) > T \\ 0 & \text{nếu } I(x,y) \leq T \end{cases}$$

Trong đó:

- f(x,y) là giá trị của ảnh sau khi phân ngưỡng tại vị trí (x,y).
- I(x,y) là giá trị cường độ sáng của ảnh gốc tại vị trí (x,y).

- T là giá trị ngưỡng (có thể được thiết lập cố định hoặc tự động tìm kiếm).

Một số phương pháp phân ngưỡng cơ bản:

- Phân ngưỡng tĩnh (Global Thresholding), là phương pháp cơ bản nhất, dùng một ngưỡng duy nhất cho toàn bộ ảnh. Phương pháp này thích hợp khi ảnh có độ sáng đồng đều. Tuy nhiên, trong trường hợp ảnh có độ sáng thay đổi (do ánh sáng hoặc đổ bóng), phương pháp này có thể không hiệu quả. Các bước cơ bản của phân ngưỡng tĩnh:
  - **Bước 1 chọn ngưỡng T**: Một giá trị cố định để phân biệt giữa các điểm ảnh sáng và tối.
  - **Bước 2 so sánh mỗi pixel**: Nếu giá trị pixel I(x, y) lớn hơn hoặc bằng T, pixel đó được gán thành 1 (đối tượng); nếu nhỏ hơn T, gán thành 0 (nền).
- Phân ngưỡng động (Adaptive Thresholding), phương pháp này xác định ngưỡng riêng cho từng vùng nhỏ của ảnh thay vì áp dụng một ngưỡng cho toàn bộ ảnh, từ đó phân tách đối tượng và nền hiệu quả hơn trong ảnh có độ sáng thay đổi.

  Các phương pháp phổ biến trong phân ngưỡng động gồm:
  - **Adaptive Mean Thresholding**: Ngưỡng tại mỗi pixel được tính là trung bình của các pixel lân cận trừ đi một giá trị cố định C.
  - **Adaptive Gaussian Thresholding**: Ngưỡng tại mỗi pixel được tính dựa trên tổng trọng số của các pixel lân cận, với trọng số là hàm Gaussian, rồi trừ đi một giá trị C.
- Phân ngưỡng Otsu, là một thuật toán tìm ngưỡng tự động. Phương pháp này tìm giá trị ngưỡng sao cho khoảng cách giữa trung bình của hai nhóm điểm ảnh (nền và đối tượng) là lớn nhất, từ đó giúp phân

biệt rõ ràng hai vùng sáng và tối. Các bước cơ bản của phương pháp Otsu:

- Xây dựng histogram của ảnh.
- Tính toán ngưỡng tối ưu T sao cho phương sai giữa các nhóm điểm ảnh là lớn nhất, điều này sẽ giúp tối ưu hóa độ phân biệt giữa nền và đối tượng.
- Áp dụng ngưỡng T vào ảnh để phân loại các điểm ảnh thành hai nhóm.

- Một số lý thuyết cơ bản đằng sau các kỹ thuật phân ngưỡng gồm:
  - **Phân tích histogram**: Xây dựng histogram của ảnh để xác định mức cường độ sáng phổ biến, từ đó chọn ra giá trị ngưỡng để phân tách các nhóm điểm ảnh.
  - **Phương sai giữa các lớp**: Được sử dụng trong phương pháp Otsu để chọn giá trị ngưỡng tối ưu. Phương sai lớn nhất giữa các lớp (background và foreground) thường dẫn đến việc phân tách tốt nhất giữa đối tượng và nền.
  - **Trung bình cục bộ**: Được sử dụng trong phân ngưỡng động, dựa trên cường độ sáng của các vùng xung quanh điểm ảnh để xác định ngưỡng cục bộ.

Trong OpenCV **threshold**() là hàm chính để thực hiện các loại phân ngưỡng cơ bản:

```
retval, dst = cv2.threshold(src, thresh, maxval, type)
```

Trong đó:

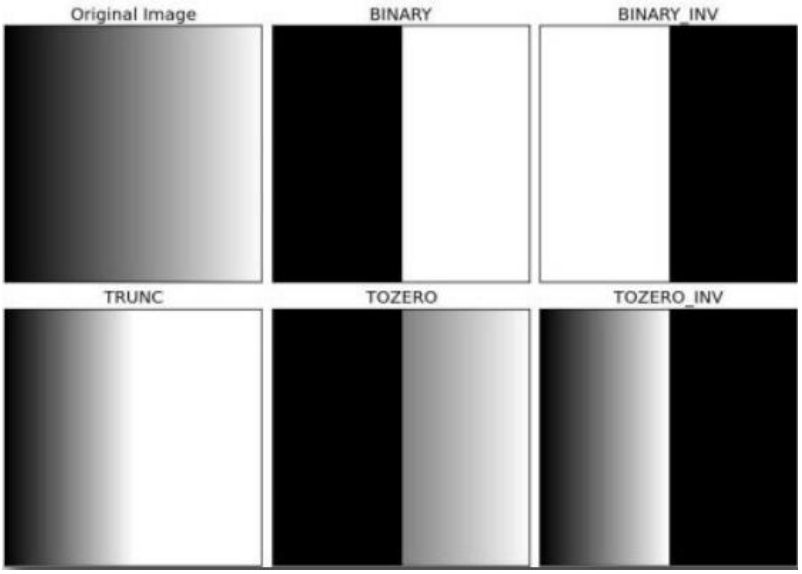
- src: Ảnh đầu vào, phải là ảnh đơn kênh (ảnh xám).
- thresh: Giá trị ngưỡng (nếu sử dụng phân ngưỡng cố định).
- maxval: Giá trị tối đa để gán cho các pixel khi điều kiện phân ngưỡng thỏa mãn.

- type: Loại phân ngưỡng cần áp dụng (chi tiết các loại sẽ được đề cập dưới đây).
- retval: Giá trị ngưỡng được sử dụng (đặc biệt hữu ích trong phương pháp Otsu).
- dst: Ảnh đầu ra sau khi phân ngưỡng.

Các loại phân ngưỡng chính (type):

- THRESH_BINARY: Gán pixel thành maxval nếu pixel lớn hơn thresh, ngược lại gán về 0.
- THRESH_BINARY_INV: Ngược lại với THRESH_BINARY - gán về 0 nếu pixel lớn hơn thresh, gán về maxval nếu không.
- THRESH_TRUNC: Các pixel lớn hơn thresh sẽ được gán bằng thresh, các pixel khác giữ nguyên giá trị.
- THRESH_TOZERO: Pixel lớn hơn thresh giữ nguyên giá trị, các pixel khác gán về 0.
- THRESH_TOZERO_INV: Ngược lại với THRESH_TOZERO - pixel lớn hơn thresh gán về 0, các pixel khác giữ nguyên giá trị.

Đây là hình minh họa trực quan với ngưỡng bằng 127:



Phương pháp phân ngưỡng ở trên không phù hợp cho nhiều trường hợp, như là ánh sáng không đồng đều trên ảnh. Trong trường hợp đó chúng ta dùng hàm **adaptiveThreshold**(). Phương thức này tính giá trị trung bình của các n điểm xung quanh pixel đó rồi trừ cho C chứ không lấy ngưỡng cố định (n thường là số lẻ, còn C là số nguyên bất kỳ). Cú pháp:

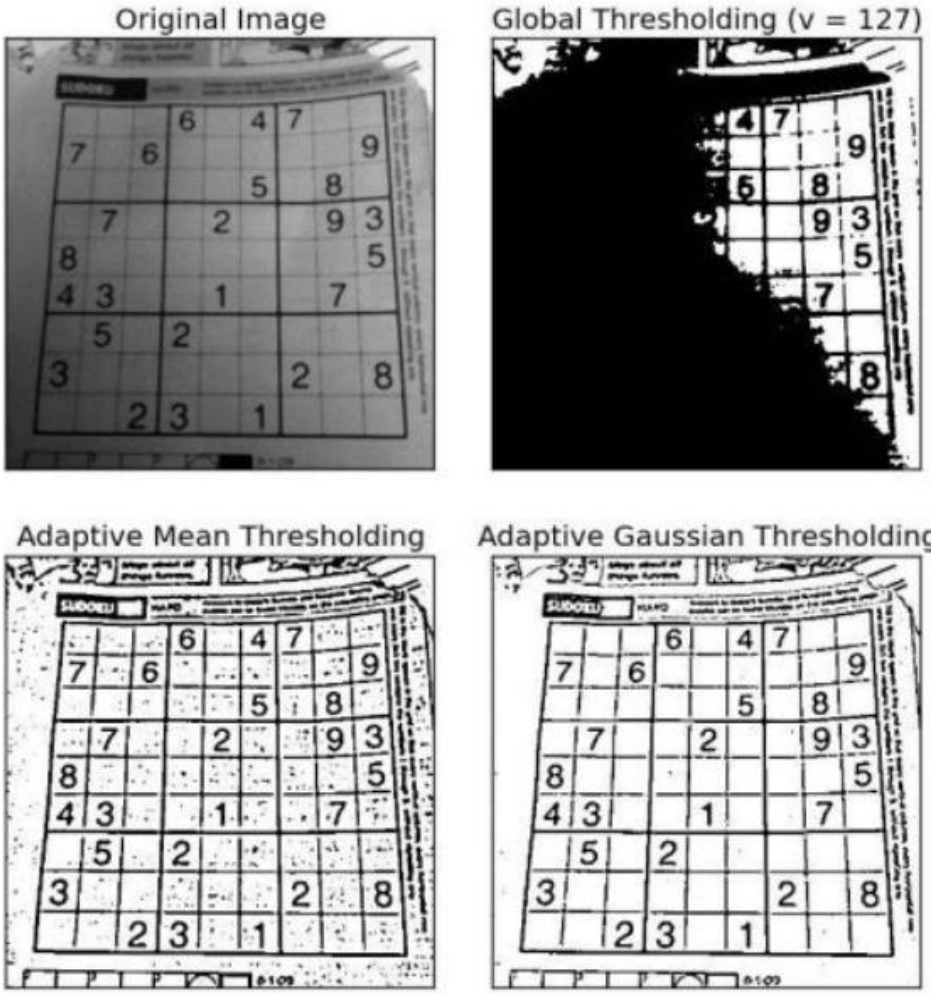
```
dst = cv2.adaptiveThreshold(src, maxValue, adaptiveMethod, thresholdType, blockSize, C)
```

Trong đó:

- src: Ảnh đầu vào (ảnh xám).
- maxValue: Giá trị gán cho các pixel nếu điều kiện phân ngưỡng thỏa mãn.
- adaptiveMethod: Phương pháp tính ngưỡng, có thể là:
  - ADAPTIVE_THRESH_MEAN_C: Ngưỡng là trung bình của vùng xung quanh pixel trừ đi một giá trị hằng số C.
  - ADAPTIVE_THRESH_GAUSSIAN_C: Ngưỡng là tổng trọng số của vùng xung quanh pixel trừ đi giá trị C, các trọng số được xác định bởi phân phối Gaussian.
- thresholdType: Loại phân ngưỡng, thường là THRESH_BINARY hoặc THRESH_BINARY_INV.
- blockSize: Kích thước của vùng (khối) dùng để tính ngưỡng. Phải là một số lẻ, ví dụ 3, 5, 7, v.v.
- C: Giá trị hằng số được trừ vào ngưỡng, giúp điều chỉnh mức độ ngưỡng.

```
adaptive_thresh_mean = cv2.adaptiveThreshold(image, 255,
                                             cv2.ADAPTIVE_THRESH_MEAN_C,
                                             cv2.THRESH_BINARY, 11, 2)
adaptive_thresh_gaussian = cv2.adaptiveThreshold(image, 255,
                                                 cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                                 cv2.THRESH_BINARY, 11, 2)
```

Hình mình họa cho các phân ngưỡng khác nhau:

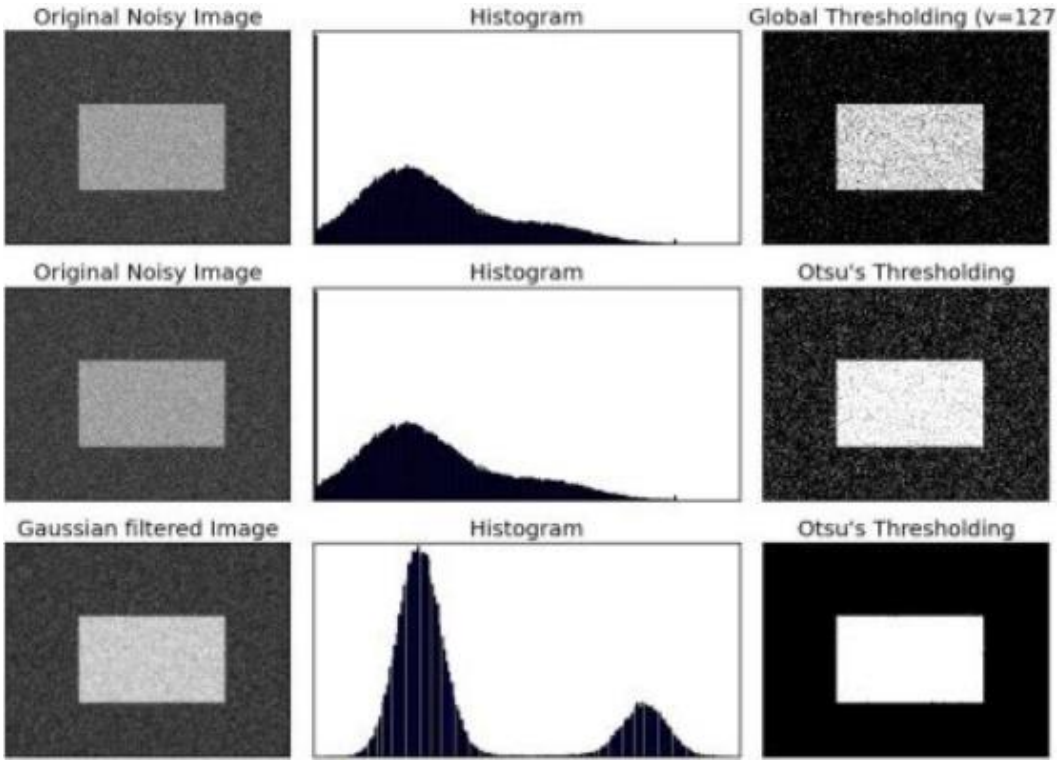


Phân ngưỡng Otsu là một phương pháp tự động tìm ngưỡng dựa trên phân tích histogram của ảnh. Phương pháp này tìm giá trị ngưỡng tối ưu sao cho độ biến thiên giữa các nhóm pixel (foreground và background) là lớn nhất. Điều này giúp xác định ngưỡng phân tách tốt giữa đối tượng và nền.

Để sử dụng phân ngưỡng Otsu trong OpenCV, đặt thresh là 0 và thêm cv2.THRESH_OTSU vào type trong hàm cv2.threshold().

```
_, otsu_thresh = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

Kết quả:



Trong đó:

- Ảnh gốc có nhiễu, các giá trị của pixel chênh lệch không nhiều nên historgram có dạng đồi thoai thoải.
- Ảnh sau khi lọc nhiễu có 2 đỉnh, đỉnh cao là những pixel sáng nhất nằm trong hình nhữ nhật nhỏ. Đỉnh thấp là những pixel sáng hơn nằm bên ngoài.

1.6. Giảm nhiễu

Trong xử lý ảnh số, giảm nhiễu là một bước quan trọng để cải thiện chất lượng ảnh và hỗ trợ các bước xử lý tiếp theo như phân ngưỡng, phát hiện cạnh, và nhận dạng đối tượng. Trong OpenCV, có nhiều phương pháp giảm nhiễu khác nhau tùy thuộc vào đặc tính của nhiễu và yêu cầu của ứng dụng.

Nhiễu là những biến dạng không mong muốn trên ảnh, thường phát sinh từ các nguồn như cảm biến, ánh sáng, môi trường, hoặc trong quá trình truyền dữ liệu. Có nhiều loại nhiễu thường gặp:

- **Nhiễu Gauss**: Nhiễu này có được do bản chất rời rạc của bức xạ (hệ thống ghi ảnh bằng cách đếm các photon (lượng tử ánh sáng). Mỗi pixel trong ảnh nhiễu là tổng giá trị pixel đúng và pixel ngẫu nhiên .
- **Nhiễu muối – tiêu (Salt & Pepper noise)**: Nhiễu này sinh ra do xảy ra sai số trong quá trình truyền dữ liệu. Những pixel đơn được đặt luân phiên mang giá trị bằng 0 hay giá trị cực đại tạo ra hình chấm dạng muối tiêu trên ảnh.
- **Nhiễu Shot hay nhiễu Poisson**: Nhiễu này sinh ra do trong quá trình thu nhận, số lượng lớn hạt photon đã tập trung vào một điểm và chúng đã tạo ra nhiễu tại điểm đó. Nhiễu được đặc trưng bởi hàm mật độ phân bố xác suất Poisson, nên được gọi là nhiễu Poisson
- **Nhiễu Speckle hay nhiễu đốm**: Là loại nhiễu phát sinh do ảnh hưởng của điều kiện môi trường lên cảm biến hình ảnh trong quá trình thu nhận hình ảnh. Nhiễu lốm đốm hầu như được phát hiện trong trường hợp ảnh y tế, ảnh Radar hoạt động và ảnh Radar khẩu độ tổng hợp (SAR).

Các phương pháp giảm nhiễu trong OpenCV:

- Bộ lọc trung bình (Mean Filter) là một kỹ thuật lọc không gian đơn giản, trong đó bộ lọc thay thế giá trị của mỗi pixel bằng giá trị trung bình của các pixel lân cận trong một cửa sổ kích thước k×k:

  ```
  mean_blur = cv2.blur(src, ksize)
  ```

  Trong đó:

  - src (image): Ảnh đầu vào.

- ksize (kernel size): Kích thước của cửa sổ vuông k×k (phải là số lẻ) để tính trung bình, ví dụ (5, 5).

```
# Áp dụng bộ lọc trung bình với kích thước kernel 5x5
mean_blur = cv2.blur(image, (5, 5))
```

Kết quả:

Ảnh gốc *Sau khi dùng blur*

- Bộ lọc Gaussian (Gaussian Filter) là một biến thể cải tiến của bộ lọc trung bình, trong đó mỗi pixel trong cửa sổ vuông được nhân với một hệ số trọng số Gaussian trước khi tính trung bình:

```
gaussian_blur = cv2.GaussianBlur(src, ksize, sigmaX, sigmaY)
```

Trong đó:

- src (image): Ảnh đầu vào.
- ksize (kernel size): Kích thước của kernel, thường là (k, k) (phải là số lẻ), ví dụ (5, 5).
- sigmaX: Độ lệch chuẩn theo trục X. Nếu sigmaX=0, OpenCV sẽ tự động tính từ kích thước kernel.
- sigmaY (tùy chọn): Độ lệch chuẩn theo trục Y (nếu không có, mặc định sigmaY=sigmaX).

```
# Áp dụng bộ lọc Gaussian với kernel 5x5 và sigmaX=0
gaussian_blur = cv2.GaussianBlur(image, (5, 5), 0)
```

Kết quả:

Ảnh gốc

Sau khi dùng GaussianBlur

- Bộ lọc trung vị thay thế giá trị của pixel bằng giá trị trung vị của các pixel lân cận trong cửa sổ vuông k×k. Phương pháp này hiệu quả trong việc loại bỏ nhiễu muối và tiêu:

```
median_blur = cv2.medianBlur(src, ksize)
```

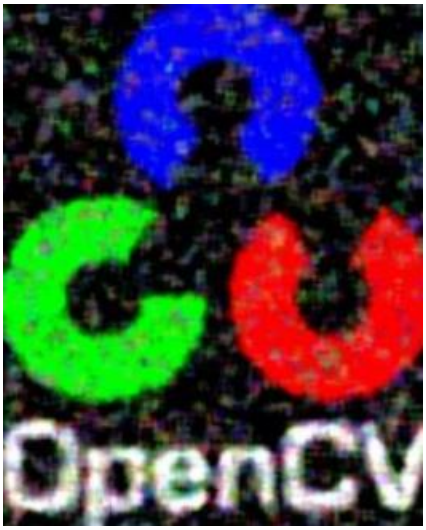
Trong đó:

- src (image): Ảnh đầu vào.
- ksize: Kích thước của kernel (phải là số lẻ), ví dụ 5.

```
# Áp dụng bộ lọc trung vị với kernel 5x5
median_blur = cv2.medianBlur(image, 5)
```

Kết quả:

Ảnh gốc



Sau khi áp dụng medianBlur

- Bộ lọc song phương (Bilateral Filter) là một bộ lọc không gian và là một trong những kỹ thuật giảm nhiễu tiên tiến. Nó làm mờ ảnh trong khi bảo toàn các cạnh bằng cách kết hợp hai yếu tố: khoảng cách không gian và cường độ màu sắc:

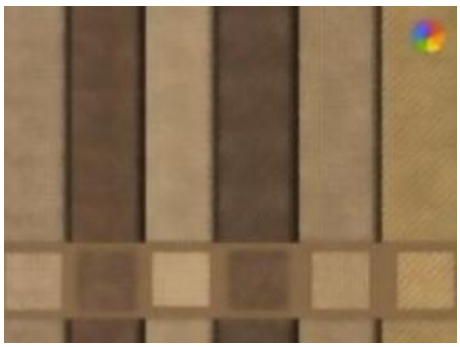
```
bilateral_blur = cv2.bilateralFilter(src, d, sigmaColor, sigmaSpace)
```

Trong đó:

  - src (image): Ảnh đầu vào.
  - d: Đường kính của vùng lân cận, xác định phạm vi của kernel. Thông thường là số nguyên dương, ví dụ 9.
  - sigmaColor: Độ lệch chuẩn trong không gian màu, điều chỉnh độ nhạy với sự thay đổi màu.
  - sigmaSpace: Độ lệch chuẩn trong không gian không gian, điều chỉnh độ nhạy với khoảng cách giữa các pixel.

```
# Áp dụng bộ lọc song phương với các tham số d=9,
# sigmaColor=75, sigmaSpace=75
bilateral_blur = cv2.bilateralFilter(image, 9, 75, 75)
```

Kết quả:



Ảnh gốc *Sau khi áp dụng bilateralFilter*

- Giảm nhiễu bằng thuật toán Fast Means Denoising thực hiện giảm nhiễu nâng cao bằng cách tìm và loại bỏ các điểm ảnh nhiễu dựa trên thống kê vùng lân cận:

```
denoised_image = cv2.fastNlMeansDenoising(src, dst, h,
                                templateWindowSize, searchWindowSize)
```

Trong đó:

- src (image): Ảnh đầu vào (chỉ chấp nhận ảnh mức xám).
- dst (None): Ảnh kết quả (thường là None vì trả về một ảnh mới).
- h: Giá trị điều chỉnh cường độ của quá trình lọc nhiễu, ví dụ 30.
- templateWindowSize: Kích thước của vùng lân cận (thường là 7).
- searchWindowSize: Kích thước vùng tìm kiếm để tính toán trọng số, ví dụ 21.

```
denoised_image = cv2.fastNlMeansDenoising(image_gray, None, 30,
                                          7, 21)
```

Kết quả:

Ảnh gốc

Denoised

- Giảm nhiễu cho anh màu. OpenCV cung cấp hàm fastNlMeansDenoisingColored để giảm nhiễu cho ảnh màu. Nó hoạt động tương tự như fastNlMeansDenoising nhưng dành cho ảnh màu:

```
denoised_image_colored = cv2.fastNlMeansDenoisingColored(src,
dst, h, hForColorComponents, templateWindowSize, searchWindowSize)
```

Trong đó:

- src (image): Ảnh đầu vào (ảnh màu).
- dst (None): Ảnh kết quả.
- h: Cường độ lọc cho kênh màu luma, ví dụ 10.
- hForColorComponents: Cường độ lọc cho các kênh màu (chromatic), ví dụ 10.
- templateWindowSize: Kích thước cửa sổ mẫu, thường là 7.
- searchWindowSize: Kích thước vùng tìm kiếm, ví dụ 21.

```
# Giảm nhiễu cho ảnh màu
denoised_image_colored = cv2.fastNlMeansDenoisingColored(image, None,
                                                         10, 10, 7, 21)
```

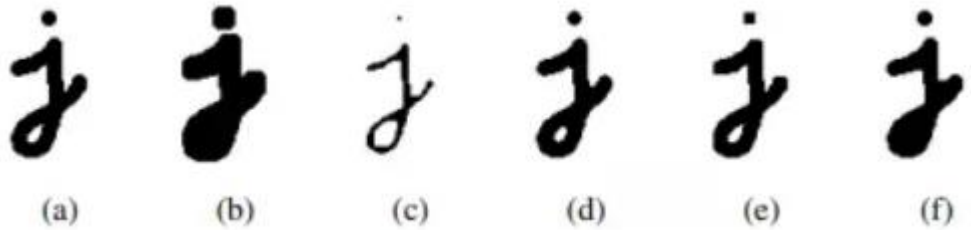
Kết quả:

Ảnh gốc *Denoised*

1.7. Biến đổi hình thái học (Morphological Transformations)

Biến đổi hình thái học (Morphological transformations) là một nhóm các kỹ thuật xử lý ảnh chủ yếu được áp dụng lên ảnh nhị phân (black & white image) hoặc ảnh xám để phân tích cấu trúc hình học của đối tượng trong ảnh. Những phép biến đổi này chủ yếu được sử dụng để thao tác với các đặc trưng hình học của ảnh, như hình dạng, kích thước và cấu trúc của các đối tượng.

Các phép biến đổi hình thái học thường dựa trên một yếu tố quan trọng là **kernel** (hạt lọc), một ma trận nhỏ được sử dụng để thao tác với các pixel trong ảnh. Các phép toán hình thái học thay đổi các pixel của ảnh đầu vào dựa trên các pixel lân cận của chúng, giúp nhấn mạnh các đặc điểm hình học của các đối tượng trong ảnh.

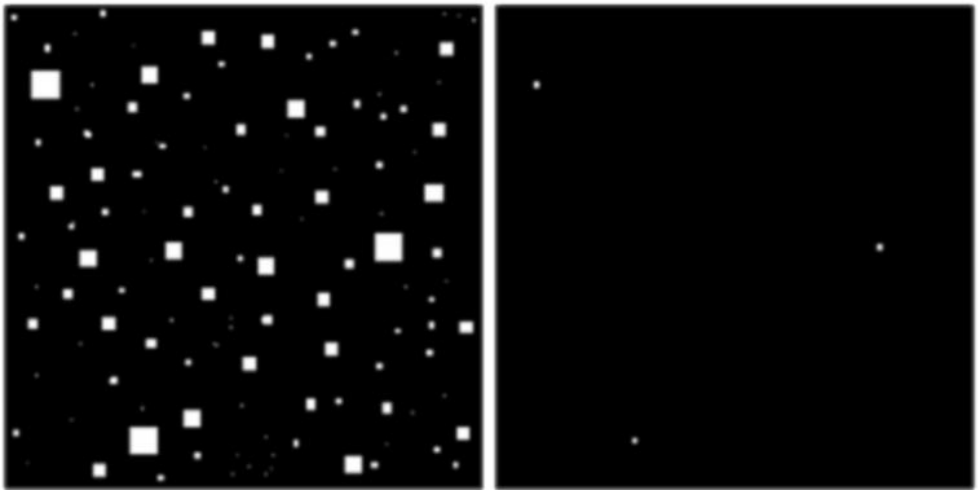


Trong đó:

- a là ảnh gốc
- b là phép (khuếch tán) dilation
- c là phép (xói mòn) erosion
- e là phép (mở) opening
- f là phép (đóng) closing

Một số phép biến đổi hình thái học cơ bản:

- Erosion (Xói Mòn) với mục đích là làm nhỏ các đối tượng sáng trong ảnh. Mỗi pixel sáng trong ảnh sẽ được thay thế bằng giá trị nhỏ nhất trong vùng lân cận của nó (theo vùng xác định bởi kernel). Erosion làm thu nhỏ các đối tượng sáng và có thể làm mất các chi tiết nhỏ trong đối tượng hoặc làm chúng bị gián đoạn. Ví dụ, nếu có một hình tròn trắng trên nền đen, sau khi xói mòn, nó có thể biến thành một hình tròn nhỏ hơn hoặc thậm chí biến mất hoàn toàn nếu kích thước hình tròn quá nhỏ:



Ảnh gốc *Ảnh sau khi xói mòn*

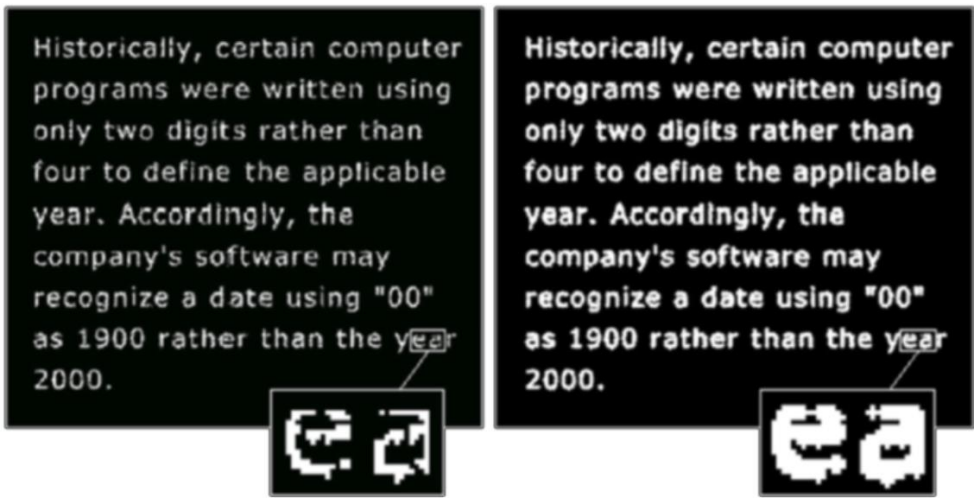
Công thức:

$$(A \ominus B)(x, y) = \min_{(i,j) \in B} A(x + i, y + j)$$

Trong đó:

- A là ảnh đầu vào,
- B là kernel (thường là một ma trận có giá trị 1 tại các điểm lân cận).
- $\ominus$ là phép toán xói mòn (erosion).

- Dilation (Khuếch Tán) với mục đích là làm lớn các đối tượng sáng trong ảnh. Mỗi pixel sáng trong ảnh sẽ được thay thế bằng giá trị lớn nhất trong vùng lân cận của nó. Dilation làm cho các đối tượng sáng mở rộng, kết nối các đối tượng gần nhau và lấp đầy các khoảng trống nhỏ trong các đối tượng. Ví dụ, nếu có một đường viền đứt đoạn giữa hai đối tượng sáng, dilation có thể kết nối chúng lại với nhau:



Công thức:

$$(A \oplus B)(x, y) = \max_{(i,j) \in B} A(x + i, y + j)$$

Trong đó:

- A là ảnh đầu vào,
- B là kernel,
- $\oplus$ là phép toán khuếch tán (dilation).

- Opening (Mở) là một phép toán kết hợp hai phép toán: xói mòn (erosion) và khuếch tán (dilation). Phép mở giúp loại bỏ nhiễu nhỏ trong ảnh. Opening làm mờ các chi tiết nhỏ hoặc các đối tượng không mong muốn, đồng thời lấp đầy các lỗ nhỏ trong đối tượng. Đây là một kỹ thuật hiệu quả để làm sạch ảnh nhị phân.

  Open = Erode next Dilate:

  $$\mathrm{open}(f, s) = \mathrm{dilate}(\mathrm{erode}(f, s), s);$$

- Closing (Đóng) là phép toán kết hợp giữa khuếch tán (dilation) và xói mòn (erosion). Phép đóng chủ yếu được dùng để lấp đầy các lỗ nhỏ hoặc kết nối các đối tượng gần nhau. Closing giúp làm mượt các đối tượng sáng, lấp đầy các khoảng trống nhỏ và kết nối các đối tượng bị đứt đoạn.

  Close = Dilate next Erode:

  $$\mathrm{close}(f, s) = \mathrm{erode}(\mathrm{dilate}(f, s), s).$$

Trong **OpenCV**, các phép biến đổi hình thái học được thực hiện thông qua các hàm như **erode**(), **dilate**(), **morphologyEx**(), v.v. Các phép biến đổi này sử dụng một "kernel" (hạt lọc), thường là một ma trận nhỏ hoặc cửa sổ (thường có kích thước 3x3, 5x5, v.v.), để thao tác với các pixel trong ảnh:

- Erosion giúp làm nhỏ các đối tượng sáng trong ảnh, giảm kích thước của chúng. Để thực hiện xói mòn trong OpenCV, sử dụng hàm **erode**():

  ```
  kernel = np.ones((3, 3), np.uint8)
  eroded image = cv2.erode(image, kernel, iterations=1)
  ```

  Trong đó:

  - **image**: Ảnh đầu vào (ảnh xám hoặc nhị phân).
  - **kernel**: Ma trận kernel (hạt lọc). Thông thường là một ma trận nhỏ với các giá trị 1 và 0.

- **iterations**: Số lần thực hiện phép toán. Nếu muốn xói mòn nhiều lần, ta có thể tăng giá trị này.

- Dilation làm lớn các đối tượng sáng trong ảnh. Để thực hiện khuếch tán trong OpenCV, sử dụng hàm **dilate**():

```
dilated_image = cv2.dilate(image, kernel, iterations=1)
```

Trong đó:

- **image**: Ảnh đầu vào (ảnh xám hoặc nhị phân).
- **kernel**: Ma trận kernel (hạt lọc), thường là ma trận vuông như 3x3 hoặc 5x5.
- **iterations**: Số lần thực hiện phép toán. Tăng giá trị này nếu muốn khuếch tán nhiều lần.

- Mở là phép toán kết hợp giữa xói mòn và khuếch tán. Nó chủ yếu được sử dụng để loại bỏ nhiễu nhỏ trong ảnh:

```
opened_image = cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)
```

Trong đó:

- **image**: Ảnh đầu vào.
- **MORPH_OPEN**: Tên hằng số biểu thị phép toán mở (opening).
- **kernel**: Ma trận kernel.

- Đóng là phép toán kết hợp giữa khuếch tán và xói mòn. Nó giúp lấp đầy các khoảng trống nhỏ trong các đối tượng:

```
closed_image = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)
```

Trong đó:

- **image**: Ảnh đầu vào.
- **MORPH_CLOSE**: Tên hằng số biểu thị phép toán đóng (closing).
- **kernel**: Ma trận kernel.

1.8. Phát Hiện Cạnh và Xử Lý Đường Viền

Phát hiện cạnh và xử lý đường viền là các bước quan trọng trong xử lý ảnh số, giúp nhận diện các cấu trúc chính trong ảnh, như biên, hình dạng và đối tượng. Việc phát hiện cạnh giúp tách biệt các đối tượng trong ảnh hoặc làm nổi bật các khu vực có sự thay đổi mạnh về màu sắc hoặc độ sáng. Phát hiện cạnh thường sử dụng các phương pháp lọc và các thuật toán toán học để xác định những nơi có sự thay đổi mạnh về cường độ sáng.

Khái niệm: Phát hiện cạnh là quá trình xác định những điểm trong ảnh nơi có sự thay đổi mạnh về độ sáng. Những thay đổi này thường chỉ ra sự hiện diện của các biên giữa các đối tượng hoặc các phần trong ảnh.

Các phương pháp phổ biến trong phát hiện cạnh bao gồm:

- **Phương pháp Gradient** (phát hiện cạnh dựa trên độ dốc của ảnh).
- **Phương pháp lọc** (sử dụng bộ lọc để phát hiện cạnh).
- **Phương pháp dựa trên biên** (như Canny, Sobel, Prewitt, Laplacian).

Phương pháp Sobel là một trong những phương pháp đơn giản và hiệu quả để phát hiện cạnh. Sobel sử dụng bộ lọc (kernel) để tính toán độ dốc trong hai hướng x và y của ảnh, sau đó kết hợp chúng để xác định độ mạnh của cạnh.

Công thức Sobel:

- **Sobel trong hướng x**:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

- **Sobel trong hướng y**:

$$\mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Độ mạnh của cạnh được tính bằng công thức:

$$G = \sqrt{G_x^2 + G_y^2}$$

Hướng của biên θ được xác định:

$$\theta = \text{atan2}\,(\mathrm{G_y}, \mathrm{G_x})$$

Thuật toán phát hiện biên theo phương pháp Sobel được thực hiện qua các bước:

- Bước 1: Tính I(x, y) × G_x = I_1 và I(x, y) × G_y = I_2
- Bước 2: Tính $|I_1|$ + $|I_2|$ = I_s
- Bước 3: Hiệu chỉnh I(x, y) = $I_s \geq \theta$? 1:0

Thuật toán Canny là một trong những thuật toán phát hiện cạnh mạnh mẽ và phổ biến nhất. Thuật toán này thực hiện qua nhiều bước:

- **Làm mờ ảnh**: Dùng bộ lọc Gaussian để giảm nhiễu.
- **Tính gradient và hướng gradient**: Để tính gradient và hướng gradient trong xử lý ảnh, ta sử dụng bộ lọc Sobel X và Sobel Y (3x3) để tính ảnh đạo hàm **Gx** và **Gy**. Sau đó, ta tính độ lớn của gradient (magnitude) và góc của gradient (angle) theo công thức::
  - **Magnitude (độ lớn của gradient)**:

$$G = \sqrt{G_x^2 + G_y^2}$$

- **Angle (góc gradient)**:

$$\theta = \text{atan2}(G_y, G_x)$$

Góc này có giá trị trong khoảng [-180, 180] độ, thể hiện hướng của cạnh. Tuy nhiên, để đơn giản hóa, ta phân chia góc này thành 4 hướng chính:

 0 độ: Hướng ngang (trục hoành).

 45 độ: Hướng chéo (góc 45 độ).

 90 độ: Hướng dọc (trục tung).

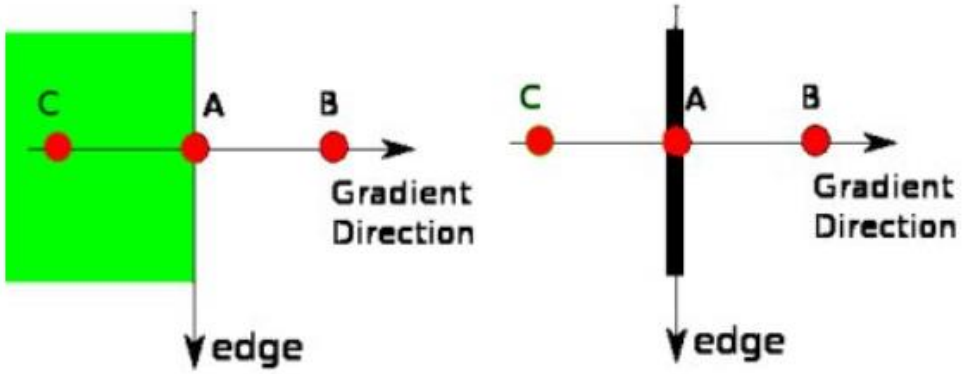
 135 độ: Hướng chéo ngược (góc 135 độ).

Kết quả là hai ma trận có kích thước giống nhau với ảnh gốc (ví dụ: 640x640):

 Ảnh gradient (magnitude) thể hiện độ mạnh của biến đổi sáng tại mỗi pixel.

 Ảnh hướng gradient (angle) thể hiện hướng của biến đổi sáng tại mỗi pixel.

- **Non-maximum suppression (viết tắt NMS)**: Chỉ giữ lại các điểm cạnh mạnh nhất (giảm bớt các điểm không quan trọng). Quá trình thực hiện:

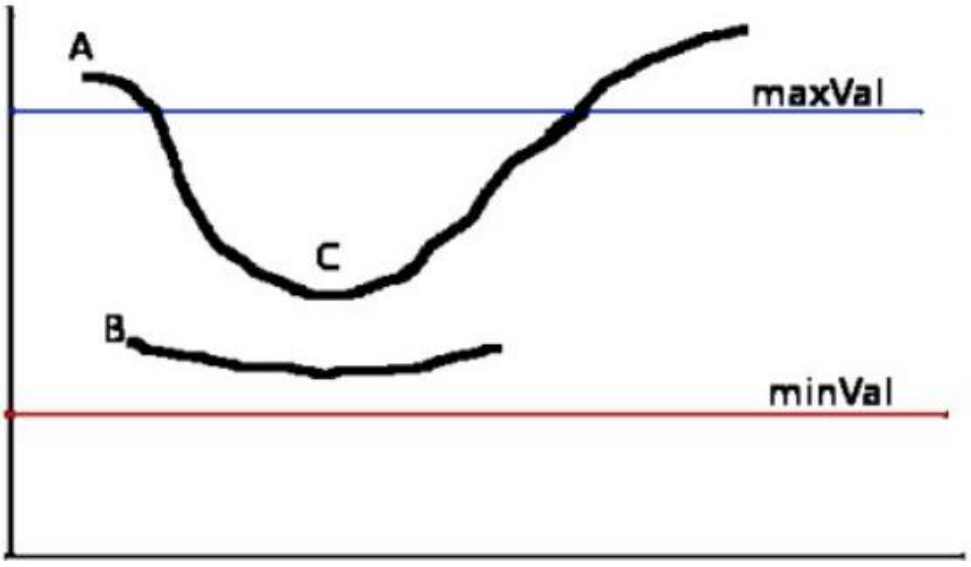


Chạy qua từng pixel trong ảnh gradient (magnitude).

- **So sánh pixel trung tâm** với 2 pixel lân cận theo hướng gradient:
  - Nếu **độ lớn gradient của pixel trung tâm** lớn hơn các pixel lân cận, giữ lại pixel đó.
  - Nếu không, **set độ lớn gradient của pixel đó về 0**.
- Các hướng gradient được chia thành 4 hướng chính (0, 45, 90, 135 độ), và ta chỉ so sánh với các pixel lân cận theo hướng đó.

Kết quả cuối cùng là một **mặt nạ nhị phân** (binary mask), trong đó các điểm cạnh mạnh nhất được giữ lại, còn các điểm không quan trọng bị loại bỏ.

- **Hysteresis thresholding**: Xác định các cạnh mạnh và yếu bằng cách sử dụng hai ngưỡng (ngưỡng cao và ngưỡng thấp) để phân biệt cạnh chính thức và các điểm không rõ ràng. Sau khi có mặt nạ nhị phân từ bước trước, ta làm như sau:



- Xét các pixel có giá trị gradient dương:
  - Nếu giá trị gradient lớn hơn ngưỡng max_val, pixel đó chắc chắn là cạnh.

 Nếu giá trị gradient nhỏ hơn ngưỡng min_val, pixel đó sẽ bị loại bỏ.

- Các pixel có giá trị gradient **nằm giữa hai ngưỡng**:

  Ta kiểm tra xem nó có liền kề với pixel chắc chắn là cạnh không (tức là có ít nhất một pixel cạnh xung quanh).

  Nếu có, ta giữ lại pixel đó. Nếu không, loại bỏ pixel.

Kết quả là chỉ giữ lại những pixel chắc chắn là cạnh hoặc những pixel liên kề với chúng, còn lại sẽ bị loại bỏ.

Cách thực hiện sobel và canny trong OpenCV:

- Sobel:

```
# Áp dụng Sobel để tính gradient theo chiều ngang và dọc
# Gradient theo chiều ngang
sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
# Gradient theo chiều dọc
sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)

# Tính độ lớn và hướng của gradient
gradient_magnitude = cv2.magnitude(sobel_x, sobel_y)
gradient_angle = cv2.phase(sobel_x, sobel_y, angleInDegrees=True)

# Hiển thị kết quả
cv2.imshow('Sobel Gradient Magnitude', gradient_magnitude)
cv2.imshow('Sobel Gradient Angle', gradient_angle)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

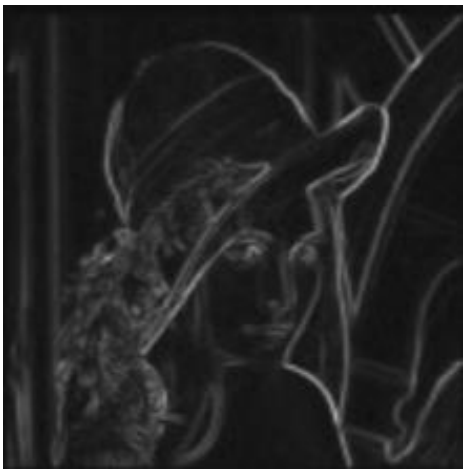
Trong đó:

- image: ảnh gốc (chuyển thành ảnh xám).
- cv2.Sobel: hàm tính gradient Sobel với các tham số: ảnh, kiểu dữ liệu, số bước gradient theo chiều X và Y, và kích thước của kernel (ksize).
- sobel_x, sobel_y: ảnh gradient theo chiều ngang và dọc.
- gradient_magnitude: độ lớn của gradient.

- gradient_angle: góc gradient.
- angleInDegrees=True: Góc trả về sẽ được tính bằng **độ** (degrees).
- angleInDegrees=False: Góc trả về sẽ được tính bằng **radian** (radians).

Kết quả:

Ảnh gốc



Magnitude

- Canny:

```
# Áp dụng Canny để phát hiện cạnh
edges = cv2.Canny(blurred_image, 100, 200)

# Hiển thị kết quả
cv2.imshow('Canny Edge Detection', edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Trong đó:

- Canny: hàm phát hiện cạnh với tham số đầu vào là ảnh, và hai ngưỡng min và max.
- edges: ảnh với các cạnh đã được phát hiện.

- Các tham số của Canny:

  minVal: ngưỡng dưới (dùng để xác định "cạnh chắc chắn").

  maxVal: ngưỡng trên (dùng để xác định "cạnh có thể là đúng").

  Canny sẽ giữ lại các cạnh nếu giá trị gradient của pixel lớn hơn ngưỡng trên, và loại bỏ nếu nhỏ hơn ngưỡng dưới.

Kết quả:

Ảnh gốc

Edges

Sobel thường được sử dụng để tính toán gradient ở các hướng khác nhau, giúp xác định các điểm thay đổi mạnh trong ảnh. Nó phù hợp cho các ứng dụng yêu cầu phát hiện cạnh theo các hướng cụ thể.

Canny là phương pháp phát hiện cạnh mạnh mẽ hơn, vì nó kết hợp cả làm mờ ảnh, tính gradient, non-maximum suppression và hai ngưỡng để lọc các cạnh, mang lại kết quả chính xác và ít bị nhiễu hơn.

1.9. Xấp Xỉ Hình Dạng và Vẽ Hình

Xấp xỉ hình dạng và vẽ hình trong xử lý ảnh số là một kỹ thuật quan trọng để nhận diện và phân tích các đối tượng trong ảnh, phục vụ cho các ứng

dụng như nhận dạng khuôn mặt, phát hiện vật thể, phân đoạn hình ảnh, hay phân tích hình học của đối tượng.

OpenCV cung cấp các hàm để vẽ các hình cơ bản như sau:

- **line**(): Vẽ một đường thẳng.
- **circle**(): Vẽ một hình tròn.
- **rectangle**(): Vẽ một hình chữ nhật.
- **polylines**(): Vẽ một đa giác (polygon).

Cách sử dụng:

- Vẽ đường thẳng:

  ```
  cv2.line(image, start point, end point, color, thickness)
  ```

  Trong đó:

  - start_point: Điểm bắt đầu của đường thẳng (x1, y1).
  - end_point: Điểm kết thúc của đường thẳng (x2, y2).
  - color: Màu sắc đường thẳng (BGR).
  - thickness: Độ dày đường thẳng.

- Vẽ hình tròn:

  ```
  cv2.circle(image, center, radius, color, thickness)
  ```

  Trong đó:

  - center: Tọa độ tâm của hình tròn (x, y).
  - radius: Bán kính của hình tròn.
  - color: Màu sắc hình tròn (BGR).
  - thickness: Độ dày viền (thickness=-1 sẽ vẽ hình tròn đầy).

- Vẽ hình chữ nhật:

  ```
  cv2.rectangle(image, pt1, pt2, color, thickness)
  ```

  Trong đó:

  - pt1: Tọa độ của điểm trên bên trái (x1, y1).
  - pt2: Tọa độ của điểm dưới bên phải (x2, y2).

- color: Màu sắc của hình chữ nhật.
- thickness: Độ dày của viền (thickness=-1 sẽ vẽ hình chữ nhật đầy).

- Vẽ đa giác (polygon):

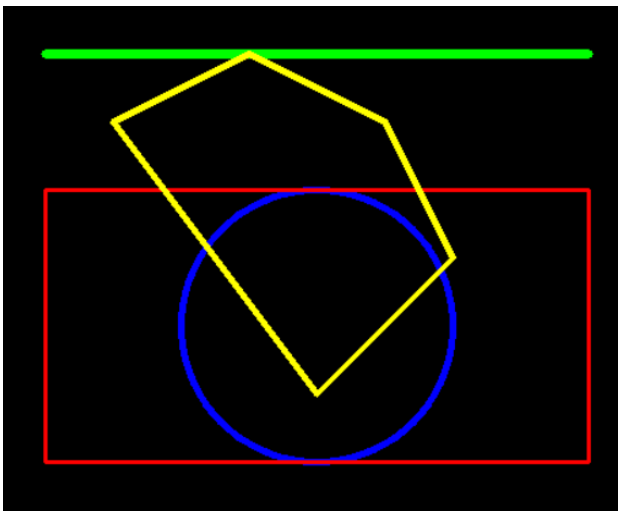
```
cv2.polylines(image, [pts], isClosed, color, thickness)
```

Trong đó:

- pts: Mảng chứa các điểm của đa giác.
- isClosed: Nếu True, các điểm sẽ được nối lại thành một hình khép kín.
- color: Màu sắc của đa giác.
- thickness: Độ dày của đường viền.

```
# Tạo một ảnh đen
image = np.zeros((500, 500, 3), dtype="uint8")

# Vẽ một đường thẳng
cv2.line(image, (50, 50), (450, 50), (0, 255, 0), 5)

# Vẽ một hình tròn
cv2.circle(image, (250, 250), 100, (255, 0, 0), 3)

# Vẽ một hình chữ nhật
cv2.rectangle(image, (50, 150), (450, 350), (0, 0, 255), 2)

# Vẽ một đa giác
pts = np.array([[100, 100], [200, 50], [300, 100], [350, 200], [250, 300]], np.int32)
pts = pts.reshape((-1, 1, 2))
cv2.polylines(image, [pts], isClosed=True, color=(0, 255, 255), thickness=3)

# Hiển thị kết quả
cv2.imshow('Shapes', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Kết quả:



Trong OpenCV, ta sử dụng hàm **findContours**() để tìm các contours trong ảnh. Hàm này trả về danh sách các đường viền (contours) được phát hiện:

```
contours, hierarchy = cv2.findContours(image, mode, method)
```

Trong đó:

- **image**: Ảnh đầu vào (thường là ảnh nhị phân sau khi áp dụng thresholding).
- **mode**: Chế độ tìm kiếm đường viền:
  - RETR_EXTERNAL: Tìm các đường viền bên ngoài (không bao gồm đường viền trong các đối tượng con).
  - RETR_LIST: Tìm tất cả các đường viền mà không xác định mối quan hệ giữa các đường viền.
  - RETR_TREE: Tìm tất cả các đường viền và xác định mối quan hệ giữa chúng (có cây phân cấp).
- **method**: Phương pháp xấp xỉ các đường viền:
  - CHAIN_APPROX_SIMPLE: Lưu trữ chỉ các điểm cần thiết để mô tả đường viền.

- CHAIN_APPROX_NONE: Lưu trữ tất cả các điểm của đường viền (có thể gây tốn bộ nhớ).

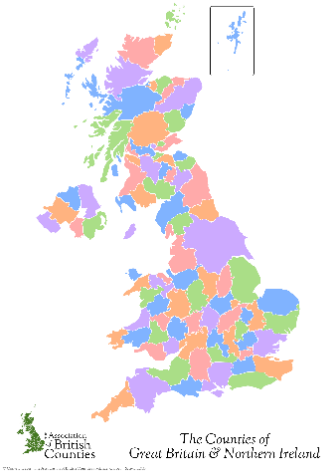
- Hàm findContours() trả về hai giá trị:
  - **contours**: Danh sách các đường viền tìm được, mỗi phần tử là một mảng các điểm tạo thành đường viền.
  - **hierarchy**: Mối quan hệ giữa các đường viền (chỉ cần trong một số trường hợp khi bạn làm việc với các đường viền lồng nhau).

```
# Áp dụng threshold để tạo ảnh nhị phân
_, thresh = cv2.threshold(image, 127, 255, cv2.THRESH_BINARY)

# Tìm các đường viền
contours, hierarchy = cv2.findContours(thresh, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

# Vẽ các đường viền lên ảnh gốc
cv2.drawContours(image, contours, -1, (0, 255, 0), 2)

# Hiển thị ảnh kết quả
cv2.imshow('Contours', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Trong đó:

- **drawContours**: Vẽ đường viền lên ảnh:
  - **[approx]**: Danh sách các đường viền coutours (ở đây chỉ có một đường viền xấp xỉ).
  - **-1**: Vẽ tất cả các đường viền (ở đây chỉ có một đường viền).
  - **(0, 255, 0)**: Màu sắc của đường viền (xanh lá cây).
  - **2**: Độ dày của đường viền.

Kết quả:



Ảnh gốc

Draw coutours

Xấp xỉ hình dạng là quá trình sử dụng các thuật toán để chuyển đổi các đường viền phức tạp thành một hình dạng đơn giản hơn, dễ dàng xử lý hoặc phân tích. Quá trình này đặc biệt hữu ích khi làm mượt đường viền của đối tượng mà không làm mất đi thông tin quan trọng.

Trong OpenCV, có một hàm rất phổ biến để thực hiện xấp xỉ hình dạng đó là **approxPolyDP()**.

Hàm **approxPolyDP()** sử dụng thuật toán Douglas-Peucker để giảm số lượng các điểm trong một đường viền, đồng thời cố gắng giữ lại hình dạng ban đầu của đối tượng:

```
cv2.approxPolyDP(contour, epsilon, closed)
```

Trong đó:

- **contour**: Đầu vào là một mảng chứa các điểm của đường viền (contour).
- **epsilon**: Tính chất chính của thuật toán. Đây là độ chính xác tối đa mà đường xấp xỉ có thể lệch so với đường viền gốc. Epsilon càng lớn, đường xấp xỉ sẽ càng đơn giản.

- **closed**: Nếu giá trị là True, đường viền được xem như là một đường khép kín (polygon). Nếu là False, đường viền không phải là một polygon.

```
# Tìm các đường viền trong ảnh
contours, _ = cv2.findContours(thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

# Kiểm tra nếu tìm thấy bất kỳ đường viền nào
if len(contours) > 0:
    # Lấy đường viền đầu tiên
    contour = contours[0]

    # Xấp xỉ đường viền
    epsilon = 0.02 * cv2.arcLength(contour, True)
    approx = cv2.approxPolyDP(contour, epsilon, True)

    if len(approx) == 3:
        print("Hình tam giác")

    if len(approx) == 4:
        # Kiểm tra nếu là hình vuông hoặc chữ nhật
        # Lấy khung chứa của contour
        x, y, w, h = cv2.boundingRect(approx)

        aspectRatio = float(w) / h  # Tỷ lệ chiều rộng/chiều cao

        if 0.95 < aspectRatio < 1.05:
            print("Hình vuông")
        else:
            print("Hình chữ nhật")

    # Vẽ đường viền xấp xỉ
    cv2.drawContours(image, [approx], -1, (0, 255, 0), 2)

    # Hiển thị ảnh
    cv2.imshow('Approximation', image)
    cv2.waitKey(0)
    cv2.destroyAllWindows()
else:
    print("Không tìm thấy đường viền trong ảnh.")
```

Trong đó:

- **cv2.arcLength(contour, True)**: Tính chu vi của đường viền (độ dài của đường viền). True chỉ định đường viền là kín.

- **epsilon = 0.02 * cv2.arcLength(contour, True)** xác định độ chính xác của xấp xỉ. epsilon là tỷ lệ phần trăm của chu vi, ở đây là 2%. Tức là, xấp xỉ sẽ giảm số điểm của đường viền sao cho nó có độ chính xác 2% so với chu vi ban đầu.
- **approxPolyDP(contour, epsilon, True)** giúp giảm số lượng điểm trong đường viền, giữ lại hình dạng gần giống với ban đầu. Hàm này sẽ xấp xỉ đường viền theo các đoạn thẳng đơn giản. Nếu đường viền có nhiều điểm, hàm này sẽ giảm số điểm này, giúp đơn giản hóa đường viền mà vẫn giữ được hình dạng cơ bản.
- **len(approx) == 3**: Nếu sau khi xấp xỉ, đường viền có 3 điểm, thì đây là hình tam giác (hình dạng có 3 cạnh).
- **len(approx) == 4**: Nếu đường viền có 4 điểm, có thể là hình vuông hoặc hình chữ nhật:
  - **boundingRect(approx)**: Tính toán một hình chữ nhật bao quanh các điểm của đường viền xấp xỉ. x, y là tọa độ góc trên bên trái của hình chữ nhật, w, h là chiều rộng và chiều cao.
  - **aspectRatio = float(w) / h**: Tính tỷ lệ chiều rộng và chiều cao của hình chữ nhật. Nếu tỷ lệ này gần 1, tức là chiều rộng và chiều cao gần bằng nhau, thì đó là một hình vuông. Nếu tỷ lệ này khác 1, thì đó là hình chữ nhật.

2. Phương pháp thực hiện

2.1. Nhận diện biển báo giao thông

Việc nhận diện biển báo giao thông trong video, ta sẽ thực hiện bằng cách xử lý từng khung hình xuất hiện trong video và tìm ra các đối tượng được xem là giống với biển báo nhất và đánh dấu lại bằng một khung bao lấy các biển báo, các thao tác xử lý với các khung hình sẽ bao gồm:

- Thiết lập và định nghĩa màu sắc: Định nghĩa các khoảng màu HSV cần phát hiện (đỏ, xanh dương, vàng, nâu đỏ) là màu đặc trưng của các biển báo giao thông bằng cách sử dụng một từ điển color_ranges:

```
color_ranges = {
    'red1': [(0, 100, 100), (10, 255, 255)],
    'red2': [(160, 100, 100), (179, 255, 255)],
    'blue': [(100, 150, 0), (140, 255, 255)],
    'yellow': [(20, 100, 100), (30, 255, 255)],
    'brown_red': [(5, 70, 50), (20, 200, 150)]
}
```

- Tạo hàm tạo mask cho từng màu: Hàm **mask_color** tạo một mask cho từng màu dựa trên giá trị HSV của khung hình.

```
def mask_color(hsv, lower_bound, upper_bound):
    mask = cv2.inRange(hsv, lower_bound, upper_bound)
    return mask
```

- Tạo hàm mask tổng hợp cho tất cả các màu:
  - Hàm **create_color_mask** chuyển đổi khung hình từ không gian màu BGR (Blue, Green, Red) sang không gian màu HSV (Hue, Saturation, Value) để dễ dàng phát hiện màu sắc.:

```
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
```

  - Tạo mask cho mỗi màu và kết hợp lại thành một mask chung. Giúp khung hình giữ lại một lớp mặt nạ (mask) chỉ chứa các phần màu trong từ điển color_ranges:

    Khởi tạo một mask trống (combined_mask) cùng kích thước với khung hình. Mask này ban đầu có tất cả giá trị là 0 và sẽ được cập nhật để chứa các vùng màu đã được xác định trong color_ranges.

```
combined_mask = np.zeros_like(cv2.inRange(hsv, (0, 0, 0), (0, 0, 0)))
```

 Duyệt qua từng màu trong color_ranges, tạo mask cho mỗi màu dựa trên các giá trị HSV đã chỉ định, sau đó kết hợp từng mask riêng lẻ vào combined_mask bằng phép bitwise_or. Mỗi lần bitwise_or sẽ làm cho combined_mask chứa các phần màu mới được phát hiện:

```
for color, (lower, upper) in color_ranges.items():
    mask = mask_color(hsv, np.array(lower), np.array(upper))
    combined_mask = cv2.bitwise_or(combined_mask, mask)
```

- Sử dụng phép biến đổi hình thái học cv2.morphologyEx với cấu trúc hình elip để thực hiện phép MORPH_OPEN, loại bỏ nhiễu và làm cho mask sạch hơn, giúp loại bỏ các điểm nhiễu nhỏ không thuộc phần màu cần phát hiện:

```
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
morph = cv2.morphologyEx(combined mask, cv2.MORPH OPEN, kernel)
```

- Áp dụng thresholding để chuyển mask về ảnh đen trắng, làm nổi bật các vùng có màu đã chọn:

```
return cv2.threshold(morph, 0, 255, cv2.THRESH BINARY + cv2.THRESH OTSU)[1]
```

Đoạn code hoàn chỉnh:

```
def create_color_mask(frame):
    hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

    combined_mask = np.zeros_like(cv2.inRange(hsv, (0, 0, 0), (0, 0, 0)))
    for color, (lower, upper) in color_ranges.items():
        mask = mask_color(hsv, np.array(lower), np.array(upper))
        combined_mask = cv2.bitwise_or(combined_mask, mask)

    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
    morph = cv2.morphologyEx(combined_mask, cv2.MORPH_OPEN, kernel)

    return cv2.threshold(morph, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)[1]
```

Tác dụng của hàm này là giúp mỗi khung hình chỉ giữ lại các phần hình ảnh có màu sắc là màu của biển báo, giúp loại bỏ các khung hình dư thừa, giảm bớt việc nhận diện sai đối tượng.

Khi này mỗi khung hình trong video chỉ giữ lại các phần hình ảnh có chứa màu sắc của biển báo, ta cần phải xác định lại hình dạng của biển báo vì nhiều vật thể khác có thể cũng có màu sắc của biển báo:

Phát hiện và đánh dấu hình dạng:

- Hàm **detect_shapes** dùng **mask màu** để tạo ra một khung hình chỉ chứa các đối tượng có màu sắc đã chọn:
  - Sử dụng hàm create_color_mask để tạo ra một mask chỉ chứa các đối tượng có màu sắc đã được xác định trong từ điển color_ranges. Mask này giúp giữ lại chỉ các phần của khung hình có màu mong muốn.

```
color_mask = create_color_mask(frame)
masked_frame = cv2.bitwise_and(frame, frame, mask=color_mask)
```

- Dùng kỹ thuật làm mờ (blur) và Canny Edge Detection để phát hiện các cạnh của đối tượng:
  - Áp dụng bộ lọc Gaussian để làm mờ ảnh xám, giảm thiểu nhiễu giúp cho việc phát hiện cạnh chính xác hơn.
  - Sử dụng cv2.Canny để phát hiện các cạnh trong ảnh. Các cạnh được phát hiện sẽ được hiển thị bằng các đường viền trắng trên nền đen.

```
gray = cv2.cvtColor(masked_frame, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(gray, (3, 3), 0)
edged = cv2.Canny(blurred, 30, 150)
```

- Tìm các đường viền (contour) của đối tượng và vẽ hình chữ nhật quanh các hình dạng phù hợp với kích thước đã quy định. Bằng việc sử dụng cv2.findContours để tìm các đường viền của đối tượng dựa trên các cạnh đã phát hiện trong bước trước. Mỗi contour là một đường bao của các vùng màu trắng.:

```
contours,   = cv2.findContours(edged, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

- Lặp qua từng đường viền để phát hiện hình dạng:
  - Sử dụng cv2.arcLength để tính chu vi (perimeter) của contour, giúp xác định độ phức tạp của hình dạng.
  - Sử dụng cv2.approxPolyDP để tìm ra số lượng đỉnh của hình dạng dựa trên chu vi.

```
for contour in contours:
    perimeter = cv2.arcLength(contour, True)
    approx = cv2.approxPolyDP(contour, 0.02 * perimeter, True)
    (x, y, w, h) = cv2.boundingRect(approx)
```

- Nếu hình dạng có từ 3 đến 4 cạnh, hoặc có nhiều cạnh hơn, sẽ được đánh dấu trên khung hình:

```
if len(approx) == 3 or len(approx) == 4:
    cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
elif len(approx) > 4:
    cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
```

- min_height và min_width nhằm giới hạn lại kích thước của biển báo, giảm bớt nhận diện sai vật thể, Kiểm tra xem chiều rộng w và chiều cao h của hình dạng có nằm trong phạm vi min_width, max_width, min_height, max_height đã chỉ định không. Nếu không, bỏ qua hình dạng đó.:

```
if w < min_width or w > max_width or h < min_height or h > max_height:
    continue
```

Đoạn code hoàn chỉnh:

```
def detect_shapes(frame, min_width=300, max_width=500, min_height=300, max_height=500):
    color_mask = create_color_mask(frame)
    masked_frame = cv2.bitwise_and(frame, frame, mask=color_mask)
    gray = cv2.cvtColor(masked_frame, cv2.COLOR_BGR2GRAY)
    blurred = cv2.GaussianBlur(gray, (3, 3), 0)
    edged = cv2.Canny(blurred, 30, 150)
    contours, _ = cv2.findContours(edged, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

    for contour in contours:
        perimeter = cv2.arcLength(contour, True)
        approx = cv2.approxPolyDP(contour, 0.02 * perimeter, True)
        (x, y, w, h) = cv2.boundingRect(approx)

        if w < min_width or w > max_width or h < min_height or h > max_height:
            continue

        if len(approx) == 3 or len(approx) == 4:
            cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
        elif len(approx) > 4:
            cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)

    return frame
```

Sau khi đã nhận diện được các vật thể xuất hiện trong mỗi khung hình có màu sắc và hình dạng giống với biển báo nhất ta cần, **xử lý video và ghi kết quả**:

- Sử dụng cv2.VideoCapture để mở video từ đường dẫn input_video_path. Nếu không thể mở video, thông báo lỗi sẽ hiển thị và hàm sẽ kết thúc:

```
cap = cv2.VideoCapture(input_video_path)
if not cap.isOpened():
    print(f"Error: Could not open video file {input_video_path}")
    return
```

- Thiết lập thông số cho video đầu ra, trong đó:
  - Định dạng codec fourcc (ở đây sử dụng XVID).
  - Lấy chiều rộng (frame_width) và chiều cao (frame_height) của khung hình từ video đầu vào.
  - Khởi tạo VideoWriter để tạo video đầu ra với đường dẫn output_video_path, codec fourcc, và kích thước khung hình tương tự như video đầu vào.

```
fourcc = cv2.VideoWriter_fourcc(*'XVID')
frame_width, frame_height = int(cap.get(3)), int(cap.get(4))
out = cv2.VideoWriter(output_video_path, fourcc, 20.0, (frame_width, frame_height))
```

- Xác định tổng số khung hình trong video (total_frames) và khởi tạo bộ đếm frame_counter để theo dõi quá trình xử lý:

```
total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
frame counter = 0
```

- Sử dụng while để duyệt qua từng khung hình của video:

```
print("Processing video...")
while cap.isOpened():
    ret, frame = cap.read()
    if not ret:
        break
```

- Dòng chữ "52100551" sẽ được thêm vào khung hình với các tham số:
  - font: kiểu font chữ (ở đây là FONT_HERSHEY_SIMPLEX)
  - position: vị trí hiển thị (ở góc trên bên trái của khung hình)
  - font_scale: kích thước font chữ
  - color: màu chữ (xanh lá cây)
  - thickness: độ dày nét chữ

```
try:
    # Add the number to the frame
    font = cv2.FONT_HERSHEY_SIMPLEX
    text = "52100551"
    position = (10, 30)
    font_scale = 1
    color = (0, 255, 0)
    thickness = 2
    frame = cv2.putText(frame, text, position, font, font_scale, color, thickness)
```

- Gọi hàm detect_shapes với các tham số min_width, max_width, min_height, và max_height để phát hiện và đánh dấu các hình dạng phù hợp. Kết quả là khung hình đã được xử lý output_frame:

```
output_frame = detect_shapes(frame, min_width=min_dim,
max_width=max_dim, min_height=min_dim, max_height=max_dim)
```

- Sử dụng out.write(output_frame) để ghi khung hình đã xử lý vào video đầu ra và cập nhật bộ đếm frame_counter:

```
out.write(output_frame)
frame_counter += 1
print(f"Processing frame {frame_counter}/{total_frames}", end="\r")
```

- Nếu có lỗi xảy ra khi xử lý khung hình, hiển thị thông báo lỗi và tiếp tục với khung hình tiếp theo:

```
except Exception as e:
    print(f"Error processing frame {frame_counter}: {e}")
```

- Khi hoàn tất quá trình xử lý, giải phóng cap và out để đóng video đầu vào và đầu ra, đồng thời hiển thị thông báo hoàn tất:

```
cap.release()
out.release()
print("\nProcessing complete. Video saved as", output_video_path)
```

Đoạn code hoàn chỉnh:

```python
def task1(input_video_path, output_video_path, min_dim=30, max_dim=1000):
    cap = cv2.VideoCapture(input_video_path)
    if not cap.isOpened():
        print(f"Error: Could not open video file {input_video_path}")
        return

    fourcc = cv2.VideoWriter_fourcc(*'XVID')
    frame_width, frame_height = int(cap.get(3)), int(cap.get(4))
    out = cv2.VideoWriter(output_video_path, fourcc, 20.0, (frame_width, frame_height))
    total_frames = int(cap.get(cv2.CAP_PROP_FRAME_COUNT))
    frame_counter = 0

    print("Processing video...")
    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            break
        try:
            # Add the number to the frame
            font = cv2.FONT_HERSHEY_SIMPLEX
            text = "52100551"
            position = (10, 30)
            font_scale = 1
            color = (0, 255, 0)
            thickness = 2
            frame = cv2.putText(frame, text, position, font, font_scale, color, thickness)

            # Process the frame for shapes
            output_frame = detect_shapes(frame, min_width=min_dim,
            max_width=max_dim, min_height=min_dim, max_height=max_dim)
            out.write(output_frame)
            frame_counter += 1
            print(f"Processing frame {frame_counter}/{total_frames}", end="\r")
        except Exception as e:
            print(f"Error processing frame {frame_counter}: {e}")

    cap.release()
    out.release()
    print("\nProcessing complete. Video saved as", output_video_path)
```

Ta thực hiện gọi hàm nhận diện biển báo bằng:

```python
task1('task1.mp4', '52100551.avi')
```

Trong đó:

- task1.mp4 là tên file video cần phát hiển biển báo.
- 52100551.avi là kết quả đầu ra sau khi nhận diện biển báo.

2.2. Nhận diện chữ số

Để nhận diện các chữ số trong một ảnh (ảnh bị nhiễu, …) ta cần làm sạch các vết nhiễu có trong ảnh, phát hiện cạnh của các chữ số và vẽ khung hình bao quanh các chữ số:

- Đầu tiên để xử lý một tấm ảnh ta cần phải đọc được tấm ảnh đó. Bằng cách sử dụng os.path.isfile để kiểm tra xem tệp đầu vào (input_path) có tồn tại hay không. Nếu không, hiển thị thông báo lỗi và kết thúc hàm:

```
if not os.path.isfile(input_path):
    print(f"Error: File '{input_path}' does not exist.")
    return
```

Sử dụng cv2.imread để tải hình ảnh từ đường dẫn input_path. Nếu hình ảnh không thể tải được, hàm sẽ hiển thị thông báo lỗi và kết thúc:

```
image = cv2.imread(input_path)
if image is None:
    print(f"Error: Cannot load image from '{input_path}'. Please check the file format and path.")
    return
```

- Sau khi đọc được tấm ảnh thành công ta cần chuyển đổi hình ảnh sang định dạng grayscale (ảnh xám) bằng cv2.cvtColor. Ảnh xám chỉ chứa các thông tin độ sáng, giúp giảm bớt độ phức tạp của việc xử lý hình ảnh:

```
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

- Sử dụng hàm cv2.fastNlMeansDenoising để loại bỏ nhiễu khỏi ảnh xám (gray_image). Các thông số:
  - h: cường độ khử nhiễu (ở đây là 110).
  - templateWindowSize: kích thước cửa sổ mẫu (ở đây là 7).
  - searchWindowSize: kích thước cửa sổ tìm kiếm (ở đây là 50). Kết quả là ảnh noiseless_image_bw, một phiên bản ảnh grayscale đã giảm nhiễu.

```
noiseless_image_bw = cv2.fastNlMeansDenoising(gray_image, None, 110, 7, 50)
```

- Dùng cv2.divide để chia pixel trong ảnh xám gốc cho ảnh không nhiễu với một hệ số scale=255. Phép chia này giúp tăng cường chi tiết của các vùng có sự thay đổi sáng tối rõ ràng:

```
divide = cv2.divide(gray_image, noiseless_image_bw, scale=255)
```

- Tạo một kernel hình elip với kích thước 3x3 để làm mịn chi tiết nhỏ trong ảnh chia (divide) bằng phép MORPH_CLOSE. Phép này giúp giảm bớt các vùng đứt gãy trong ảnh:

```
kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
morph_image = cv2.morphologyEx(divide, cv2.MORPH_CLOSE, kernel)
```

- Sử dụng hàm cv2.Canny để phát hiện các cạnh trong ảnh morph_image:
  - threshold1 và threshold2: ngưỡng dưới và trên cho phát hiện cạnh (100 và 200).
  - apertureSize: kích thước của Sobel kernel (ở đây là 7).
  - L2gradient=True: sử dụng công thức L2 để tính gradient chính xác hơn. Kết quả là một ảnh chứa các cạnh được phát hiện (edges).

```
edges = cv2.Canny(morph_image, 100, 200, apertureSize=7, L2gradient=True)
```

- Sử dụng cv2.findContours để tìm các đường viền từ ảnh cạnh (edges). Chỉ lấy các đường viền ngoài (RETR_EXTERNAL) và sử dụng phương pháp CHAIN_APPROX_SIMPLE để đơn giản hóa các điểm của đường viền.

```
contours,   = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
```

- Duyệt qua từng đường viền c trong contours:
  - Tạo một hình chữ nhật bao quanh đối tượng với cv2.boundingRect.

- Kiểm tra nếu chiều rộng w và chiều cao h của hình chữ nhật đáp ứng kích thước tối thiểu (ở đây là w > 24 và h > 45).
- Nếu đáp ứng, tạo một points là một mảng chứa tọa độ bốn góc của hình chữ nhật.
- Sử dụng cv2.polylines để vẽ hình chữ nhật lên image với màu xanh lá và độ dày nét vẽ là 2 pixel.

```
for c in contours:
    x, y, w, h = cv2.boundingRect(c)
    if w > 24 and h > 45:
        points = np.array([[x, y], [x + w, y], [x + w, y + h], [x, y + h]])
        cv2.polylines(image, [points], isClosed=True, color=(0, 255, 0), thickness=2)
```

- Lưu ảnh đầu ra vào đường dẫn output_path bằng cv2.imwrite. Nếu lưu thành công, hiển thị thông báo xác nhận. Nếu không, hiển thị thông báo lỗi:

```
success = cv2.imwrite(output_path, image)
if success:
    print(f"Output image saved to '{output_path}'")
else:
    print(f"Error: Unable to save image to '{output_path}'")
```

- Sử dụng try-except để xử lý bất kỳ lỗi nào xảy ra trong quá trình xử lý hình ảnh và in ra thông báo lỗi:

```
except Exception as e:
    print(f"An error occurred during processing: {e}")
```

Đoạn mã đầy đủ:

```python
def task2(input_path, output_path):
    if not os.path.isfile(input_path):
        print(f"Error: File '{input_path}' does not exist.")
        return

    image = cv2.imread(input_path)
    if image is None:
        print(f"Error: Cannot load image from '{input_path}'.
        Please check the file format and path.")
        return

    try:
        gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        noiseless_image_bw = cv2.fastNlMeansDenoising(gray_image, None, 110, 7, 50)

        divide = cv2.divide(gray_image, noiseless_image_bw, scale=255)

        kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
        morph_image = cv2.morphologyEx(divide, cv2.MORPH_CLOSE, kernel)

        edges = cv2.Canny(morph_image, 100, 200, apertureSize=7, L2gradient=True)
        contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)

        for c in contours:
            x, y, w, h = cv2.boundingRect(c)
            if w > 24 and h > 45:
                points = np.array([[x, y], [x + w, y], [x + w, y + h], [x, y + h]])
                cv2.polylines(image, [points], isClosed=True, color=(0, 255, 0), thickness=2)

        success = cv2.imwrite(output_path, image)
        if success:
            print(f"Output image saved to '{output_path}'")
        else:
            print(f"Error: Unable to save image to '{output_path}'")

    except Exception as e:
        print(f"An error occurred during processing: {e}")
```

Ta có thể thực hiện gọi hàm task2 để phát hiện chữ số trong ảnh bằng:

```python
task2("input.png", "output.png")
```

- Trong đó:
  - input.png là tên file ảnh cần đọc để phát hiện chữ số.
  - output.png là tên file cần ghi sau khi đã phát hiện các chữ số có trong ảnh.

CHƯƠNG 2. KẾT QUẢ NHIỆM VỤ

1. Kết quả nhận diện biển báo giao thông

Hình 1: Nhận diện biển báo giao thông

Chú thích: Các biển báo xuất hiện trong khung hình đều được nhận diện chính xác bắt đầu từ khoảng cách khoảng 10m. Vị trí xuất hiện là hai biển báo ở giữa màn hình và một biển báo phía ngoài bên tay phải. Các vật thể xung quanh (không phải biển báo) chỉ bị nhận diện nhầm lẫn một vài vật thể.

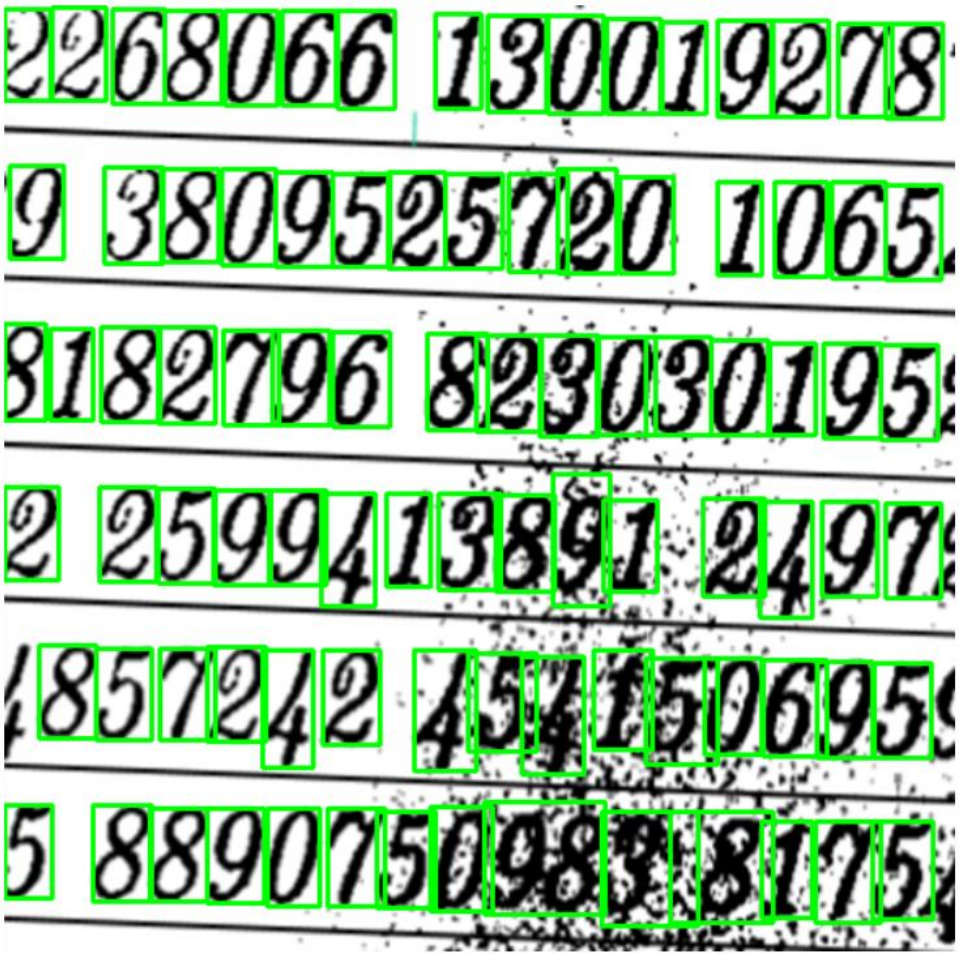
Hình 2: Nhận diện biển báo giao thông

Chú thích: Các biển báo trong khung hình đều được nhận diện chính xác từ khoảng cách khoảng 10m. Vị trí xuất hiện là hai biển báo ở giữa màn hình và một biển báo phía ngoài bên tay phải. Các vật thể xung quanh (không phải biển báo) chỉ bị nhận diện nhầm lẫn một vài vật thể.

Hình 3: Nhận diện biển báo giao thông

Chú thích: Các biển báo trong khung hình đều được nhận diện chính xác từ khoảng cách khoảng 10m. Vị trí xuất hiện là hai biển báo ở giữa màn hình và một biển báo phía ngoài bên tay phải. Các vật thể xung quanh (không phải biển báo) chỉ bị nhận diện nhầm lẫn vài vật thể.

2. Kết quả nhận diện chữ số



Hình 4: Kết quả nhận diện chữ số

Chú thích: Hầu hết các chữ số xuất hiện trong ảnh đều được nhận diện chính xác, không bị bỏ sót. Chỉ có một trường hợp hai chữ số bị nhận diện trùng nhau.

Link video output:

https://drive.google.com/file/d/1L8YZ2KWvjHe82ORfbqLnueHk73Ih73o-/view?usp=sharing

TÀI LIỆU THAM KHẢO

[1] – Tham khảo về thư viện OpenCV – **topdev.vn** – Link

[2] – Tham khảo các lý thuyết trong – **opencv.org** – Link

[3] – Tham khảo lý thuyết về ngưỡng ảnh – **thigiacmaytinh.com** – Link

[4] – Một số phương pháp khử nhiễu ảnh – **viblo.asia** – Link

[5] – Tiền xử lý ảnh – **viblo.asia** – Link

[6] – Phát hiện cạnh – **minhng.info** – Link

[7] – Giảm nhiễu ảnh – **opencv.org** – Link

[8] – **Gonzalez, R. C., & Woods, R. E.** (2008). *Digital Image Processing*. Prentice Hall.

[9] – **Image Processing in OpenCV**. *docs.opencv.org*.

[10] – **Gose, E., Johnsonbaugh, R., & Jost, S.** (1996). *Pattern Recognition and Image Analysis*. Prentice Hall.