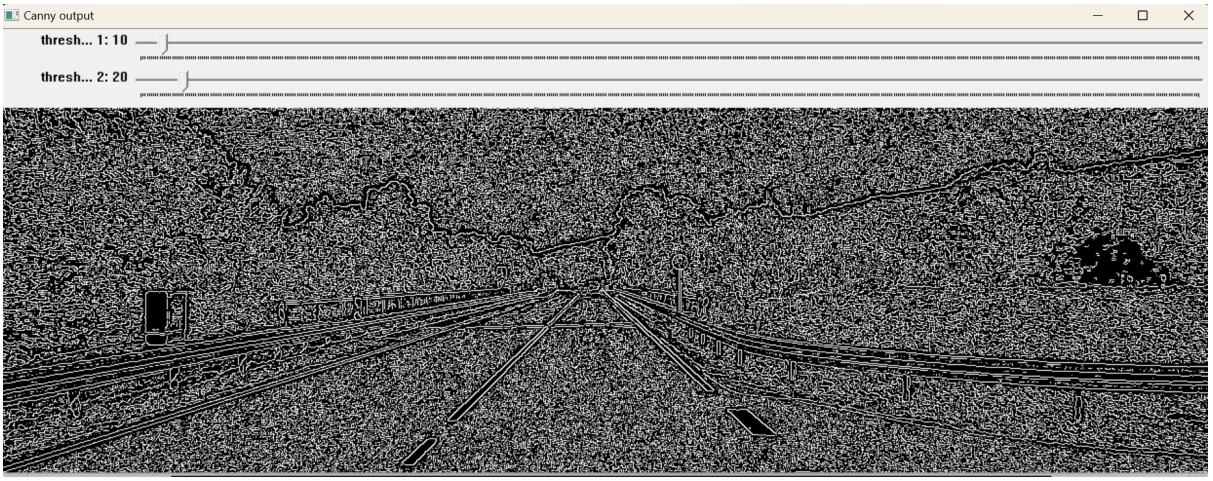
Lab report 4 Computer Vision - Matteo De Gobbi

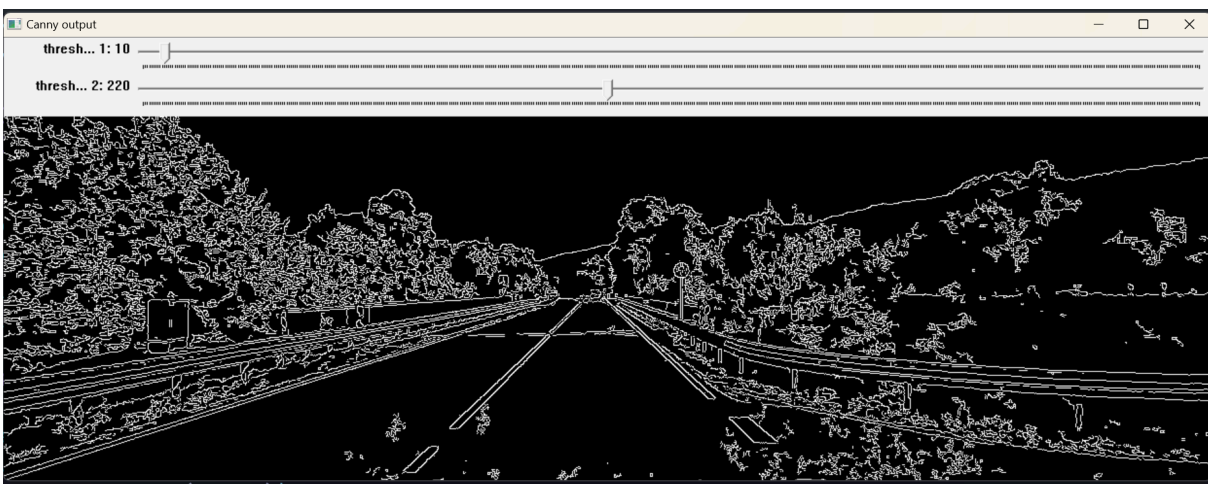
Task 1

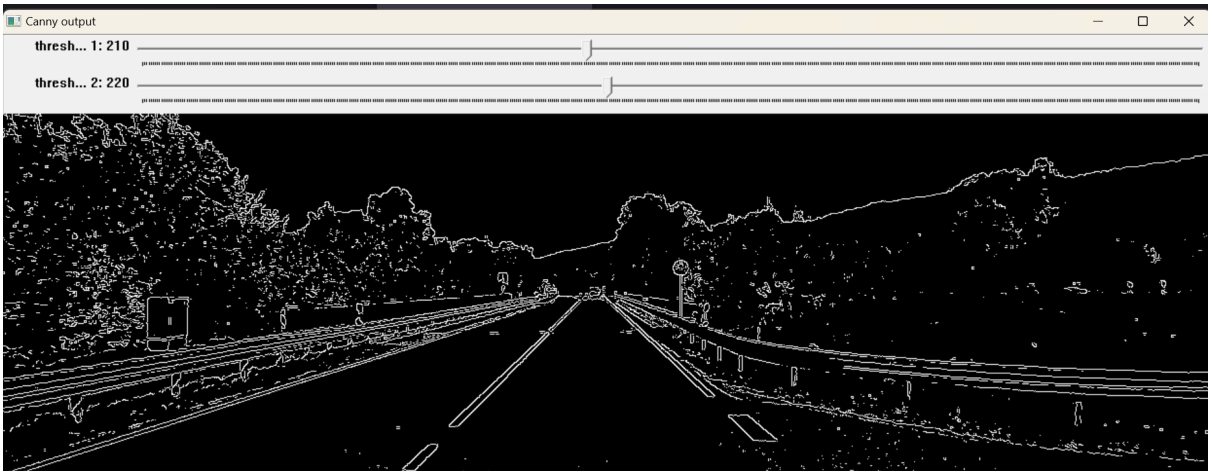
We can add two trackbars to the window with the function createTrackbar, this function works similarly to the onMouseCallback from the previous lab. We use a function pointer as a callback to execute when the slider of the trackbar is changed.
This is the callback function:

```
static void on_trackbar(int _, void *userdata) {
  userdata_t data = *static_cast<userdata_t *>(userdata);
  double thres1 = (double)*data.threshold1;
  double thres2 = (double)*data.threshold2;
  Canny(data.in, data.out, thres1, thres2);
  imshow("Canny output", data.out);
}
```

I compute the canny filtering of the image using the two thresholds that I get from the two trackbars. The second threshold is used to determine which “strong edges” are kept, and the first threshold is used to determine which “weak edges” connected to the strong ones must be kept.
Increasing the second threshold removes a lot of edges while increasing the first threshold only removes the ones connected to the stronger ones.
Examples:







Note: If we input a weak edge threshold greater than the strong edge threshold opencv automatically swaps the two, otherwise it wouldn't make sense.

Task 2

In this task we are required to find the white road lines in the picture without using the Hough transform. The approach I took was inspired by the following lecture slide:

A possible approach:

- Compute edges
- Consider all couples of edge points and evaluate the line passing through them
- Count the number of edge points on such line
- Complexity: $O(n^2)$ couples – adding comparisons: $O(n^3)$

But I had to use an improved version otherwise the algorithm would be too slow.
The algorithm consists of:

Image preprocessing

After reading the image we convert it to grayscale and threshold it to keep only the parts of the image that have a color close to white:

```
cvtColor(in, gray_in, COLOR_BGR2GRAY);
Mat threshold_in;
threshold(gray_in, threshold_in, 250, 255, THRESH_TOZERO);
```

The result of this thresholding is this, we can see how the white box on the left is still visible because its color is very similar to the white road lines.

Now we want to remove the white box on the left, we can create a new “morphed” image using an erosion followed by a dilation, the morphed image:

We can see how in this image only the box and a small piece of the line are visible, so we can subtract this morphed image from the threshold_in image (the output of the thresholding) to remove the white box on the left and keep the road lines:

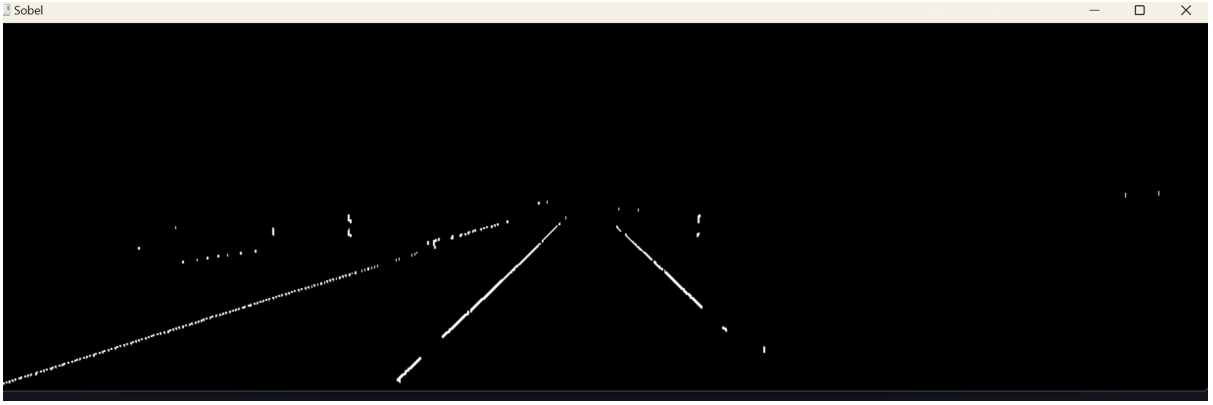
The code to achieve this is:

```
constexpr int morph_size = 7;
Mat stucturing_el =
    getStructuringElement(MORPH_RECT, Size(morph_size, morph_size));
Mat morphed;
erode(threshold_in, morphed, stucturing_el);
stucturing_el = getStructuringElement(MORPH_RECT, Size(11, 11));
dilate(morphed, morphed, stucturing_el);
Mat temp = threshold_in - morphed;
```

This works because the erosion only removes the road lines and keeps a piece of other white parts of the image that are not thin (like the white box).

Sobel filtering

The last part of the preprocessing consists in taking the Sobel derivative along the x-axis to thin the road lines, this is useful because the next part of the algorithm needs to save all the white pixels and if we kept the thick lines we would need to store a lot of redundant points, we only need a subset of them in order to compute a line passing through them, the result of taking the gradient is:



Computing the lines

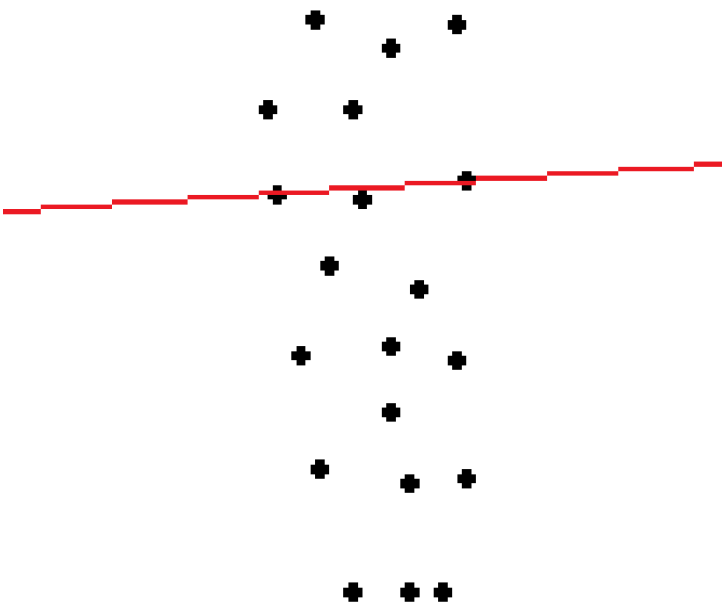
Now we use an algorithm to find the lines. First we store all the white pixels' coordinates in a (preallocated for performance) vector:

```
std::vector<Vec2i> coords;      constexpr int coords_vec_reserve = 2000
coords.reserve(coords_vec_reserve);
for (int i = 0; i < sobel_out.rows; i++) {
  for (int j = 0; j < sobel_out.cols; j++) {
    if (sobel_out.at<uchar>(i, j) != 0) {
      coords.push_back(Vec2i(i, j));
    }
  }
}
```

Then for every couple of points stored we compute the line passing through them,
we can use a few improvements to make the algorithm faster, first of all if we compute the line passing through (A,B) we don't need to compute the one passing through (B,A) so we start the index of the internal loop as the one of the external loop+1:

```
for (int i = 0; i < coords.size(); i++) {
  Vec2i p1 = coords[i];
  // NOTE we only check from i+1 becuase we dont want to check the same
  // couple twice
  for (int j = i + 1; j < coords.size(); j++) {
```

Also we can skip computing the lines passing through points close together because often the points are part of the same object and we would compute a lot of lines that pass through very few points, for example:



In this configuration the red line passing through the points doesn't represent the pattern. We can avoid these kinds of lines by having a radius in which we ignore points and don't compute the line passing through them. This can greatly reduce the number of lines we need to store to check their validity.

This is implemented in code by checking if the two points are distant more than a predefined constant:

```
if (norm(p1 - p2) > ignore_radius) {
```

Then we just compute the angular coefficient and the intercept of the line passing through the two points and count how many points are close enough to the line (up to distance_valid: a constant equal to 1.0)

```
double m = ((double)(p2[1] - p1[1])) / (p2[0] - p1[0]);
double b = (double)p2[1] - (double)m * p2[0];
int count = 0;
for (Vec2i p3 : coords) {
  assert(sqrtf(1 + m * m) != 0);
  double distance = fabs(p3[1] - m * p3[0] - b) / sqrtf(1 + m * m);
  if (distance < distance_valid) {
    count++;
  }
}
if (count > line_valid_count) {
  lines.push_back({.m = m, .b = b});
```

Then if the line passes near more than line_valid_count (which is equal to 200 in my example) we consider the line as valid and we draw it.

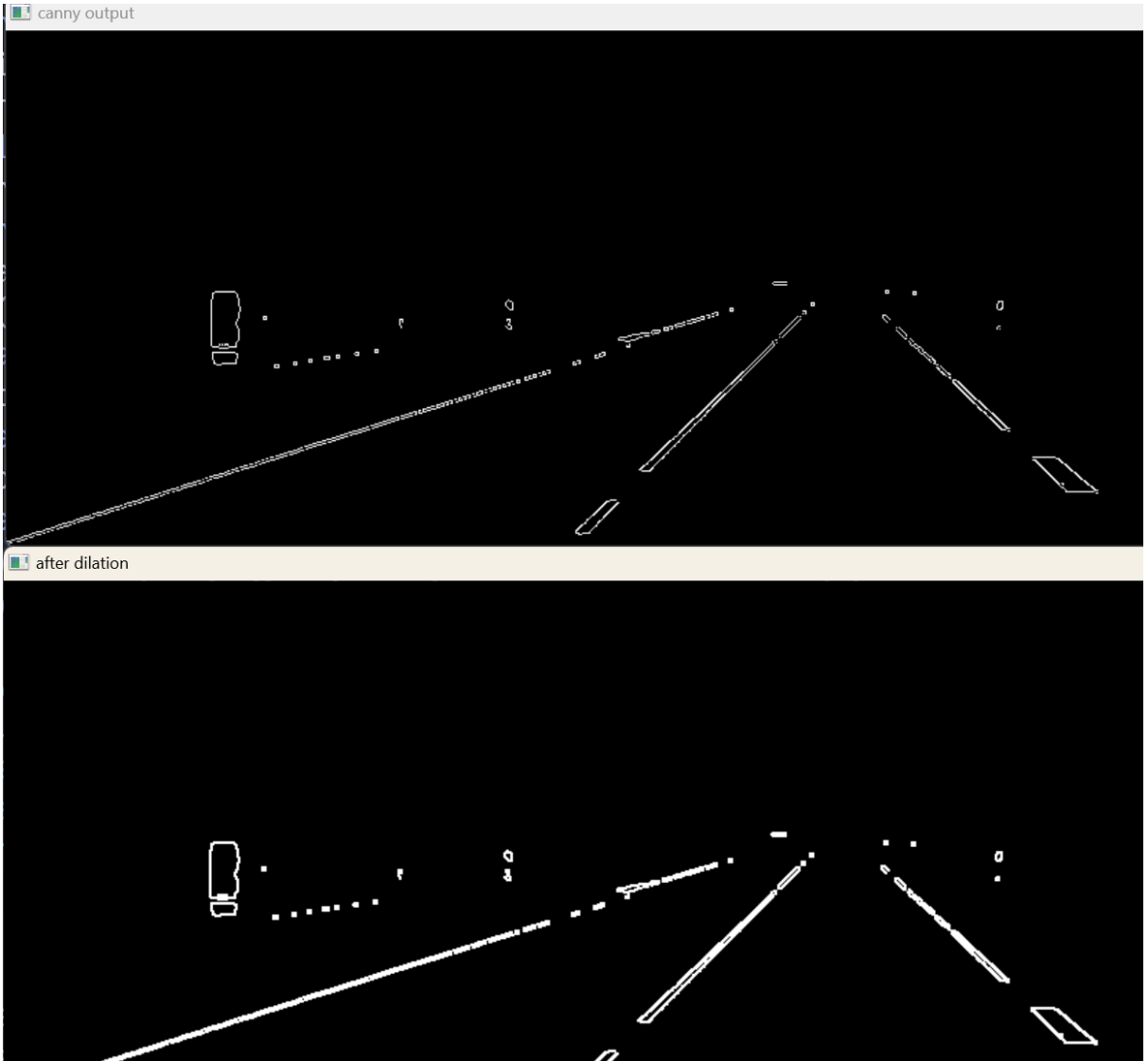
```
for (line_params pars : lines) {
  // NOTE order is y coord, x coord
  Point p = Point(pars.b, 0);
  Point q = Point((pars.m * (in.cols - 1) + pars.b), in.cols - 1);
  line(out, p, q, Scalar(0, 0, 255));
}
imshow("final image with lines", out);
```

The final output is:

The algorithm finds all 3 road lines but for the left one it finds a few different angular coefficients which are considered as valid, while the other two lines are more precise.

Task 3

In this task we have the same objective as Task 2 but using the Hough transform provided by opencv.
As in task 2 we first threshold the image to remove everything that has a dark color.
Then we could just apply the HoughLines function directly but in this way only the two leftmost lines are detected so I did some preprocessing before: I compute Canny and then use dilation to make the rightmost line (that previously wasn't detected) thicker.
The result is:



In the end the detected lines are:

Task 4

In task 4 we want to detect the road sign using HoughCircles transform provided by opencv.
We can do some preprocessing to make this task easier:
First I do a median filtering to reduce noise and avoid detecting “fake” circles.
Then I take the Laplacian to only have the edges of the shapes and remove regions of uniform color.
Then I do a dilation operation to avoid the problem of Laplacian creating two edges, using a dilation the two edges get merged together. This is the result after the dilation:

Then we can call the HoughCircles transform on this image to detect the circle outline of the road sign.

```
HoughCircles(out, circles, HOUGH_GRADIENT, 1, out.rows / 10, 100, 30, 5, 15);
```

This is the result: