Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

ОТЧЁТ О НАУЧНО-ИССЛЕДОВАТЕЛЬСКОЙ РАБОТЕ

Обнаружение меток на изображении средствами алгоритмов компьютерного зрения

по направлению подготовки 15.03.06 «Мехатроника и робототехника»
направленность (профиль) 15.03.06_01 «Проектирование и конструирование мехатронных модулей и механизмов роботов»

Выполнил
студент гр. 3331506/10101 Обшатко А.И.

Научный руководитель
доцент ВШАиР К.Ф.-М.Н. Ананьевский М.С.

«___» _____ 2024 г.

Санкт-Петербург
2024

Содержание

ВВЕДЕНИЕ

Область применения нейронных сетей для распознавания объектов на изображении огромна. Однако для каждого решения, использующего нейросети, для её тренировки требуется база исходных данных, состоящая из изображений, содержащих объект, который необходимо обнаруживать и информации о том, где на данном изображении находится этот объект. Обычно создание этой базы данных делается вручную, что может занимать много времени, так как количество изображений, необходимое для высокой точности работы нейросети, исчисляется тысячами для каждого объекта. С целью автоматизации процесса разметки, была придумана методика, позволяющая обозначить границы объекта еще в процессе его съемки. Установив специальные предметы (далее - метки), можно обозначить границы проекции объекта на сенсор камеры, благодаря чему программа сможет обнаружить метки и по их положению определить границы объекта. В данной работе исследуется функционал библиотеки алгоритмов компьютерного зрения OpenCV с целью создания программы, способной обнаруживать метки на изображении.

Цель работы – создание программы, способной с высокой надежностью и производительностью определять метки на изображении.

Задачи:

1) выбор конфигурации метки,
2) создание алгоритма для обнаружения метки,
3) оптимизация алгоритма

1. Метка

Точность обнаружения метки программой будет зависеть не только от самой программы, но и от того, как выглядит метка. К ней необходимо предъявить следующие требования:

А) Небольшой размер. Метки должны быть достаточно малыми, чтобы после того, как они были расставлены по границам объекта, они могли уместиться в область видимости камеры при максимальном необходимом приближении камеры к объекту.

Б) Внешний вид метки должен быть таким, чтобы схожая конфигурация встречалась в случайном окружении редко.

В) Изображение метки не должно претерпевать значительных искажений при отдалении от камеры. Метка на большом расстоянии должна все еще выглядеть как метка.

Так вариант использовать широко распространённые qr-коды в качестве меток может на первый взгляд показаться подходящим решением, но при отдалении от камеры его изображение значительно искажается, и обнаружить его уже не является возможным. Используя qi-коды пришлось бы создавать множество копий меток разного размера для съемок объекта на разном расстоянии, чего хотелось бы избежать.

Главным образом отталкиваясь от требования В) была выбрана следующая конфигурация метки: квадрат, разделенный на 4 равные части двумя осями симметрии, соединяющими середины противоположных сторон. Левая верхняя часть и правая нижняя закрашены в черный цвет, а левая нижняя и правая верхняя белая. Изображение метки представлено на рисунке 1.

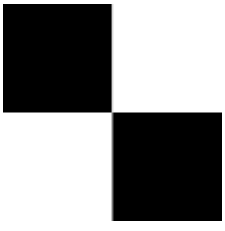


Рисунок 1 – метка

Данная конфигурация обладает важным свойством: её центральная часть на изображении выглядит всегда одинаково, вне зависимости от расстояния между камерой и меткой. Это позволяет разделить задачу об обнаружении метки на изображении на две части: обнаружение её центра, а затем поиск её границ.

В дальнейшем под наименованием «метка» будет подразумеваться именно такая конфигурация.

2. Алгоритм обнаружения

Предъявим требования к точности и производительности алгоритма. Для оценки точности будем использовать формулу

$$P = \frac{N_c}{N},$$

, где N – общее количество обнаруженных меток, N_c – количество верно обнаруженных меток.

Минимальной требуемой точностью выберем значение $P = 0{,}95$. Такой точности будет достаточно для того, чтобы в процессе съемки изображений с объектом, с учетом дополнительных условий, например близкого значения размеров обнаруженных на изображении меток, все размеченные фрагменты содержали этот объект.

Для оценки производительности будем использовать среднее время, за которое алгоритм полностью проверяет изображение на наличие меток и возвращает результат. Пусть это время для изображения разрешением 720 на 1280 пикселей не превышает 0,033 секунды, что соответствует времени между кадрами видео при частоте 30 кадров в секунду.

2.1 Обнаружение центра метки

Как было сказано выше, используя выбранную конфигурацию метки, задача о её обнаружении разделяется на две части: обнаружение центра и обнаружение границ. Точность алгоритма в большей степени будет зависеть от точности обнаружения центра метки.

Так как в идеальном случае квадратный фрагмент изображения, меньший чем изображение метки, центр которого совпадает с центром метки, всегда отображается одинаковым образом (цвета пикселей в левой верхней и правой нижней четвертях фрагмента близки к черному, а в левой нижней и правой верхней к белому), то задачу обнаружения центра можно свести к поиску схожих фрагментов.

По той же причине нет смысла рассматривать исходное трехканальное изображение, имеет смысл перевести его в поле оттенков серого.

Создадим функцию, принимающую изображение, на котором необходимо обнаружить метки, и возвращающую вектор обнаруженных меток, в котором каждому элементу соответствуют координаты левого верхнего и правого нижнего углов метки. Сразу переведем исходное изображение в поле оттенков серого и сохраним в новой переменной:

```
std::vector<std::vector<Point>> find_marks(Mat input_image)
{
    std::vector<std::vector<Point>> marks_coordinates = {};
    Mat img_grey;

    cvtColor(input_image, img_grey, COLOR_BGR2GRAY);
}
```

В изображении, полученном с камеры, чаще всего присутствует шум. Для повышения точности распознавания имеет смысл сгладить изображение. Однако в то же время важно, чтобы границы между белыми и черными частями изображения метки оставались четкими. В библиотеке OpenCV присутствует множество методов сглаживания, один из которых – билатеральный фильтр, который позволяет избавить изображение от шума, при этом оказывая малое влияние на резкость переходов между значительно отличающихся в цвете частей изображения. Применим данный фильтр к черно-белому изображению:

```
bilateralFilter(img_grey, img_blur, 5, 10, 10);
```

Для поиска центра метки предлагается следующий алгоритм: последовательно выбирать квадратные фрагменты изображения небольшого размера, применять к ним бинаризацию, выбрав значение порога на основе значений пикселей, присутствующих в этом фрагменте, так, чтобы пиксели, находящиеся в темных частях потенциальной метки, стали полностью черными (значение пикселей 0), а в светлых частях полностью белыми (значения пикселей 255). После этого применим операцию исключающего

«или» между выбранным фрагментом и изображением метки такого же размера, где так же её темные части полностью черные, а светлые полностью белые. Далее такое изображение будет называться идеальным изображением метки. Количество белых пикселей будет показывать, сколько пикселей в выбранном фрагменте не соответствуют изображению метки. Размер этого фрагмента будет определять минимальный размер изображения метки, который будет возможно распознать. Выберем значение стороны квадратного фрагмента равное 10 пикселей и зададим его глобальной переменной: `int min_mark_size = 10`. Также создадим переменную `int half_mark_size = 5`, хранящую в себе половину размера фрагмента. Создадим функцию, которая будет на вход принимать изображение, затем будет последовательно выбирать фрагменты изображения, проверять, находится ли в нем центр изображения метки, и возвращать вектор всех обнаруженных меток, в котором каждому элементу соответствуют координаты левого верхнего и правого нижнего углов метки:

```
std::vector<std::vector<Point>> search(const Mat scanning_image)
{
    std::vector<std::vector<Point>> output_coordinates = {};
    Mat crop_image;
    Range vertical_side;
    Range horizontal_side;

    int frame_width = static_cast<int>(scanning_image.cols);
    int frame_height = static_cast<int>(scanning_image.rows);

    for(int y_bottom_right = min_mark_size; y_bottom_right <= 
frame_height; y_bottom_right += 1) {
        for(int x_bottom_right = min_mark_size; x_bottom_right <= 
frame_width; x_bottom_right += 1) {
            vertical_side = Range(y_bottom_right – min_mark_size, 
y_bottom_right);
            horizontal_side = Range(x_bottom_right – min_mark_size, 
x_bottom_right);
            crop_image = scanning_image(vertical_side, 
horizontal_side);
}
```

Выполним операцию бинаризации над фрагментом. В качестве порогового значения выберем среднее значение цвета пикселей фрагмента:

```
threshold(crop_image, crop_image, mean(crop_image), 255, THRESH_BINARY);
```

Создадим функцию для инициализации изображения идеальной метки, принимающую размер метки в пикселях, и вызовем её. Важно, чтобы этот размер выражался четным числом пикселей. Изображение идеальной метки при размере в 10 пикселей представлено на рисунке 2:

```
void init_perfect_mark(int side_lenth)
{
    perfect_mark = Mat(min_mark_size, min_mark_size, CV_8UC1, Scalar(255, 255, 255));
    rectangle(perfect_mark, Point(0, 0), Point(half_mark_size - 1, half_mark_size - 1), Scalar(0, 0, 0), -1);
    rectangle(perfect_mark, Point(half_mark_size, half_mark_size), Point(min_mark_size - 1, min_mark_size - 1), Scalar(0, 0, 0), -1);
}
```

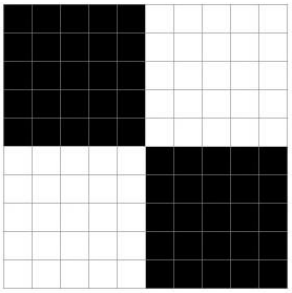


Рисунок 2 – изображение идеальной метки размером 10 пикселей

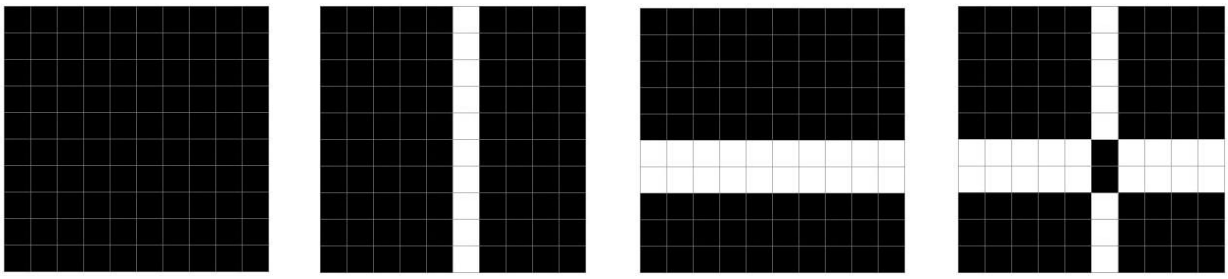
Теперь необходимо выполнить операцию исключающего «или» между фрагментом после бинаризации и идеальным изображением метки. Количество пикселей со значением не равным 0 укажет, сколько пикселей на выбранном фрагменте отклоняется от изображения идеальной метки:

```
Mat mark_xor;
bitwise_xor(input_image, perfect_mark, mark_xor);
int difference = countNonZero(mark_xor);
```

Если значение difference равно 0, то утверждаем, что фрагмент находится в центре изображения метки.

Можно заметить, что, применяя данный алгоритм, в случае, когда фрагмент не находится в центре изображения метки, но смещен на определенное количество пикселей, не меньшее половины размера фрагмента, направление и размер этого смещения можно определить по изображению, полученному после применения операции исключающее «или».

Так в случае, когда центр фрагмента лежит на центре изображения метки, изображение, полученное после операции `bitwise_xor` состоит полностью из черных пикселей (рисунок 3 а)) Если центр фрагмента смещен относительно центра изображения метки на один пиксель вправо, то один столбец пикселей, примыкающий справа к вертикальной оси симметрии полученного изображения, будет полностью состоять из белых пикселей (рисунок 3 б)). Если же имеет место смещение на два пикселя вниз, то две строки пикселей, примыкающих снизу к горизонтальной оси симметрии полученного изображения, будут полностью состоять из белых пикселей (рисунок 3 в)). Если же центры смещены одновременно на один пиксель вправо и на два пикселя вниз, то результирующее изображение выглядит как результат исключающего «или» между предыдущими двумя результатами (рисунок 3 г)).



а) б) в) г)

Рисунок 3 – результат операции исключающее «или» при различных значениях смещения

Данную закономерность можно описать следующим образом:

А) Если имеет место смещение только в горизонтальном направлении, то его значение в пикселях равно количеству полностью белых вертикальных столбцов, примыкающих к вертикальной оси симметрии результирующего изображения. Направление смещения соответствует тому, с какой стороны эти столбцы примыкают к оси симметрии.

Б) Если имеет место смещение только в вертикальном направлении, то его значение в пикселях равно количеству полностью белых горизонтальных строк, примыкающих к горизонтальной оси симметрии результирующего изображения. Направление смещения соответствует тому, с какой стороны эти столбцы примыкают к оси симметрии.

В) В случае смещения в обоих направлениях одновременно его значение в каждом направлении определяется аналогично предыдущим случаям, при том условии, что на пересечении белых столбцов и строк находятся черные пиксели.

Считая, что если в результирующем изображении присутствует данная закономерность, то была обнаружена метка, смещение центра которой было вычислено, дает возможность перебирать фрагменты исходного изображения с шагом большим, чем один пиксель, с целью повышения производительности. Можно выбрать размер шага, равный размеру перебираемых фрагментов, так, где бы на исходном изображении не находился центр метки, он обязательно попадет ровно в один фрагмент. Но в таком случае, если изображение метки достаточно мало, чтобы во фрагменте могли одновременно оказаться и центр метки, и её край, то её обнаружить не удастся несмотря на то, что размер изображения метки при этом может быть больше рассматриваемого фрагмента. Такой ситуации можно избежать, если размер шага будет равен половине размера фрагмента.

Отредактируем тело функции `search`, изменив значение шага циклов:

```
std::vector<std::vector<Point>> search(const Mat scanning_image)
{
    std::vector<std::vector<Point>> output_coordinates = {};
    Mat crop_image;
    Range vertical_side;
    Range horizontal_side;

    int frame_width = static_cast<int>(scanning_image.cols);
    int frame_height = static_cast<int>(scanning_image.rows);

    for(int y_bottom_right = min_mark_size; y_bottom_right <=
frame_height; y_bottom_right += half_mark_size) {
        for(int x_bottom_right = min_mark_size; x_bottom_right <=
frame_width; x_bottom_right += half_mark_size) {
            vertical_side = Range(y_bottom_right – min_mark_size,
y_bottom_right);
            horizontal_side = Range(x_bottom_right – min_mark_size,
x_bottom_right);
            crop_image = scanning_image(vertical_side,
horizontal_side);

            threshold(crop_image, crop_image, mean(crop image), 255,
THRESH_BINARY);
}
```

Отметим тот факт, что теперь есть возможность, увеличивая размер фрагмента, значительно влиять на производительность, так как это будет уменьшать количество итераций, увеличивая при этом минимальный размер изображения метки, которое возможно обнаружить.

Создадим функцию для вычисления смещения. Она будет принимать рассматриваемый фрагмент и возвращать вектор целых чисел, состоящий из смещения по вертикальной оси и по горизонтальной оси в пикселях, если фрагмент соответствует изображению метки, или пустой вектор в противном случае. Алгоритм вычисления смещения следующий:

1) Выполнить операцию исключающего «или» между полученным изображением и изображением идеальной метки. Вычислим количество белых пикселей.

2) Создать матрицу, размер которой равен размеру изображения, принятого на вход функцией, а элементы соответствуют черному цвету.

3) Заполнить столбец матрицы, примыкающий к её вертикальной оси симметрии справа значениями, соответствующими белому цвету.

4) Выполнить операцию исключающее «или» между этим изображением и изображением, полученным при выполнении пункта 1. Если после этой операции количество белых пикселей уменьшилось, перезапишем изображение отклонения от метки результатом этой операции и увеличим значение переменной, хранящее вычисление по горизонтальной оси, на единицу. В противном случае, при условии, что смещение по горизонтальной оси равно нулю, выполнить пункты 3 – 5 для столбца, примыкающего к вертикальной оси симметрии слева, перемещая его влево и уменьшая смещение по горизонтальной оси на единицу, иначе закончить вычисление.

5) Если смещение на единицу меньше половины размера метки, закончить вычисление, вернув пустой вектор. Иначе переместить белый столбец на один пиксель вправо и вернуться к пункту 4.

Аналогично вычисляется смещение по вертикальной оси, считая положительным направлением смещения вниз.

Код данного алгоритма выглядит следующим образом:

```
std::vector<int> calculate_offset(const Mat input_image)
{
    int x_offset = 0;
    int y_offset = 0;

    Mat mark_xor = apply_tile_mask(input_image);
    int difference = countNonZero(mark_xor);

    Mat sample = Mat(min_mark_size, min_mark_size, CV_8UC1, Scalar(0,
0, 0));
    Mat xor_line;
    int new_difference;
    int col = half_mark_size;
    int row = half_mark_size;
```

```
bool run = true;
while(run) {
        line(sample, Point(col, 0), Point(col, min_mark_size - 1),
Scalar(255, 255, 255), 1);
        bitwise_xor(mark_xor, sample, xor_line);
        line(sample, Point(col, 0), Point(col, min_mark_size - 1),
Scalar(0, 0, 0), 1);
        new_difference = countNonZero(xor_line);

        if(col >= half_mark_size) {
            col++;
        }else {
            col--;
        }

        if(new_difference < difference) {
            mark_xor = xor_line.clone();
            difference = new_difference;
            if(col >= half_mark_size) {
                x_offset += 1;
            } else {
                x_offset -= 1;
            }
        } else {
            if(x_offset == 0 && col >= half_mark_size) {
                col = half_mark_size - 1;
            } else {
                run = false;
            }
        }

        if(col == min_mark_size - 1 || col == 0) {
            return {};
        }
    }

   run = true;
    while(run) {
        line(sample, Point(0, row), Point(smallest_mark_size - 1,
row), Scalar(255, 255, 255), 1);
        bitwise_xor(mark_xor, sample, xor_line);
```

```
        line(sample, Point(0, row), Point(smallest_mark_size - 1,
row), Scalar(0, 0, 0), 1);
        new_difference = countNonZero(xor_line);

        if(row >= half_mark_size) {
            row++;
        }else {
            row--;
        }

        if(new_difference < difference) {
            mark_xor = xor_line.clone();
            difference = new_difference;
            if(row >= half_mark_size) {
                y_offset += 1;
            } else {
                y_offset -= 1;
            }
        } else {
            if(y_offset == 0 && row >= half_mark_size) {
                row = half_mark_size - 1;
            } else {
                Run = false;
            }
        }

        if(row == min_mark_size - 1 || row == 0) {
            return {};
        }
    }

    if(difference != 0) {
            return {};
    }

    return {x_offset, y_offset};
}
```

Данный алгоритм распознает метку только в том случае, если изображение метки не претерпело искажений. Чаще всего имеют место смещение вертикальных или горизонтальных границ между черными и

белыми частями метки, расположенных в разных её половинах. В таком случае места отклонения изображения от идеальной метки после выполнения операции исключающее «или» будут отображены белыми пикселями. Пример искаженного изображения и результата этой операции с ним изображены на рисунке 4.

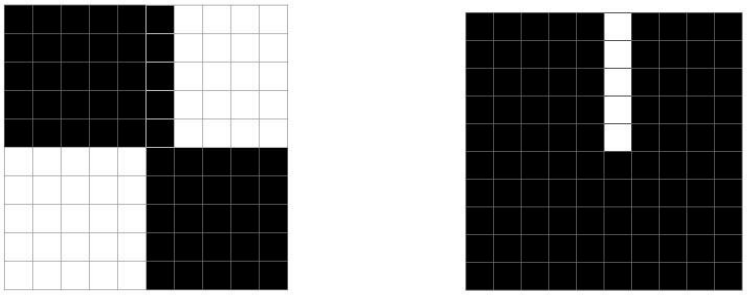


Рисунок 4 – искаженное изображение метки и результат операции исключающее «или»

Подобные искажения особенно часто возникают, если во время записи видео камера движется относительно метки. Так как алгоритм создания базы изображений объекта подразумевает перемещение камеры, необходимо ослабить критерии отбора меток. В связи с тем, что искажения в большинстве случаев возникают на границе цветов, то чаще всего отклонения будут присутствовать не более чем в двух четвертях метки. Поэтому предлагается после вычисления смещения в той же функции проверить, в скольких четвертях находятся отклонения от идеальной метки, а также что они находятся только в столбцах, примыкающих к границам между черными и белыми частями метки. Если данные условия не выполняются – утвердить, что в рассматриваемом фрагменте метка отсутствует, вернув пустой вектор. Программная реализация данных проверок выглядит следующим образом:

```
    if(difference != 0) {
        int bad_segments_counter = 0;

        Mat top_left_segment = mark_xor(Range(0, half_mark_size +
y_offset), Range(0, half_mark_size + x_offset));
        if(countNonZero(top_left_segment) != 0) {
            bad_segments_counter++;
```

```
        };
        Mat top_right_segment = mark_xor(Range(0, half_mark_size +
y_offset), Range(half_mark_size + x_offset, min_mark_size));
        if(countNonZero(top_right_segment) != 0) {
            bad_segments_counter++;
        };
        Mat bottom_left_segment = mark_xor(Range(half_mark_size +
y_offset, min_mark_size), Range(0, half_mark_size + x_offset));
        if(countNonZero(bottom_left_segment) != 0) {
            bad_segments_counter++;
        };
        Mat bottom_right_segment = mark_xor(Range(half_mark_size +
y_offset, min_mark_size), Range(half_mark_size + x_offset,
min_mark_size));
        if(countNonZero(bottom_right_segment) != 0) {
            bad_segments_counter++;
        };

        if(bad_segments_counter > 2) {
            return {};
        }

        line(mark_xor, Point(half_mark_size + x_offset - 1, 0),
            Point(half_mark_size + x_offset - 1, min_mark_size - 1),
            Scalar(0, 0, 0), 1);
        line(mark_xor, Point(half_mark_size + x_offset, 0),
            Point(half_mark_size + x_offset, min_mark_size - 1),
            Scalar(0, 0, 0), 1);
        line(mark_xor, Point(0, half_mark_size + y_offset - 1),
            Point(min_mark_size - 1, half_mark_size + y_offset - 1),
            Scalar(0, 0, 0), 1);
        line(mark_xor, Point(0, half_mark_size + y_offset),
            Point(min_mark_size - 1, half_mark_size + y_offset),
            Scalar(0, 0, 0), 1);
        difference = countNonZero(mark_xor);

        if(difference != 0) {
            return {};
        }
    return {x_offset, y_offset};
```

Теперь будем считать, что если функция `calculate_offset` возвращает не пустой вектор, то в текущем фрагменте изображения был обнаружен центр метки. Прибавив значения смещения к угловым точкам рассматриваемого фрагмента, получим координаты угловых точек фрагмента, лежащего в центре метки. Добавим в функцию `search` следующие строки:

```
            std::vector<int> offset =
calculate_offset(crop_thresholded);
            if(offset.empty()) {
                continue;
            }
            Point center_top_left = Point(horizontal_side.start +
offset[0], vertical_side.start + offset[1]);
            Point center_bottom_right = Point(horizontal_side.end +
offset[0], vertical_side.end + offset[1]);
```

2.2 Обнаружение границ метки

Реализуем данную задачу следующим образом: двигаясь от краев обнаруженной метки, будем искать резкое изменение среднего цвета пикселей. Для этого в функции `search` проведем бинаризацию исследуемого изображения, используя то же значение порога что и ранее:

```
            threshold(scanning_image, image_thresgolded,
mean(crop_image), 255, THRESH_BINARY);
```

Создадим функцию, принимающую изображение после бинаризации и угловые точки фрагмента с центром метки и возвращающую вектор точек, где первая будет соответствовать левой верхней точке всей метки, а вторая правой нижней. Первым делом в данной функции создадим переменные, хранящие в себе размеры изображения, чтобы не выйти за его границы при движении от центра метки:

```
std::vector<Point> get_mark_borders(Mat scanning_image, Point
top_left, Point bottom_right) {
    int frame_width = static_cast<int>(scanning_image.cols);
    int frame_height = static_cast<int>(scanning_image.rows);
```

Нам потребуется функция, возвращающая среднее значение цвета пикселей на крайних столбцах или строках рассматриваемой части изображения, то есть по её границе. Для большей надежности будем считать среднее значение отдельно для каждой половины крайней стороны изображения. Так алгоритм будет более чувствителен к небольшим изменениям цвета, чем если рассматривать стороны целиком. Если изменение хотя бы одного среднего значения превысит порог, будем считать, что края метки обнаружены. Предполагая, что метка лежит в плоскости обзора камеры, считаем, что края её изображения равноудалены от центра. Таким образом функция должна возвращать вектор из восьми значений:

```
std::vector<Scalar> get_edges_mean(Mat img, Point top_left, Point
bottom_right) {
    int top_middle_x = top_left.x + (bottom_right.x - top_left.x + 1)
/ 2;
    int right_middle_y = top_left.y + (bottom_right.y - top_left.y +
1) / 2;

    return {
        mean(img(Range(top_left.y, top_left.y + 1), Range(top_left.x,
top_middle_x))),
        mean(img(Range(top_left.y, top_left.y + 1),
Range(top_middle_x, bottom_right.x + 1))),
        mean(img(Range(top_left.y, right_middle_y),
Range(bottom_right.x, bottom_right.x + 1))),
        mean(img(Range(right_middle_y, bottom_right.y + 1),
Range(bottom_right.x, bottom_right.x + 1))),
        mean(img(Range(bottom_right.y, bottom_right.y + 1),
Range(top_middle_x, bottom_right.x + 1))),
        mean(img(Range(bottom_right.y, bottom_right.y + 1),
Range(top_left.x, top_middle_x))),
        mean(img(Range(right_middle_y, bottom_right.y + 1),
Range(top_left.x, top_left.x + 1))),
        mean(img(Range(top_left.y, right_middle_y), Range(top_left.x,
top_left.x + 1)))
    };
}
```

Также необходима функция, проверяющая, что текущие координаты точек находятся в пределах изображения:

```
bool if_in_bounds(Point top_left_point, Point bottom_right_point, int
frame_width, int frame_height) {
    if(top_left_point.x < 0) {
        return false;
    }
    if(top_left_point.y < 0) {
        return false;
    }
    if(frame_width - 1 < bottom_right_point.x) {
        return false;
}
    if(frame_height – 1 < bottom_right_point.y) {
        return false;
}
    return true;
}
```

Вернемся к функции `get_mark_borders`. Пока координаты угловых точек рассматриваемого фрагмента находятся в пределах изображения, и пока не найдены граны метки, будем сдвигать левую верхнюю точку на один пиксель вверх и влево, а правую нижнюю на один пиксель вниз и вправо. Вычислим новые средние значения цветов пикселей на краях фрагмента, и если хотя бы одно из них претерпело сильное изменение, считаем, что край метки найден:

```
Point point_offset = Point(half_mark_size, half_mark_size);
    std::vector<Scalar> new_edges_mean;
    while(if_in_bounds(top_left, bottom_right, frame_width,
frame_height)) {
        top_left -= point_offset;
        bottom_right += point_offset;
        new_edges_mean = get_edges_mean(scanning_image, top_left,
bottom_right);

        for(int edge = 0; edge < 8; edge++) {
            if(abs((edges_mean[edge][0] - new_edges_mean[edge][0])) >
80) {
return {top_left, bottom_right};
        }
```

```
edges_mean = new_edges_mean;
 }
return {};
```

В случаях, когда обнаруживается ложная метка, можно считать, что ее окружение состоит из пикселей случайных цветов, и в такой ситуации часто происходит так, что резкое изменение среднего цвета обнаруживается на большом расстоянии от центра ложной метки. Так как данная программа будет использоваться с целью автоматизации разметки данных, когда на изображении находятся одновременно не менее двух меток и исследуемый объект, имеет смысл прекращать поиск границ метки, если её размер превышает определенное значение. Добавим это условие в цикл:

```
        if(bottom_right.x - top_left.x > 200) {
            return {};
        }
```

После того как функция `get_mark_borders` вернет угловые точки метки, внутри функции `search` добавим их в вектор, хранящий координаты всех меток на изображении:

```
            std::vector<Point> mark_points =
get_mark_borders(image_thresgolded, center_top_left,
center_bottom_right);
            if(mark_points.empty()) {
                continue;
            }
            output_coordinates.push_back(mark_points);
        }
```

3. Оптимизация алгоритма

Алгоритм выполняет свою функцию, однако его выполнение занимает большое количество времени.

Не имеет смысла считать смещение для каждого выбранного фрагмента, если некоторые из них можно отсеять заранее по более простым признакам. Можно заметить, что, если центр метки находится во фрагменте, и её размер

достаточно большой, чтобы её обнаружить, в левом верхнем и правом нижнем пикселях обязательно будет отображаться её черная часть, а в левом нижнем и правом верхнем – белая. Имеет смысл определить минимальное значение пикселя, которое можно считать белым цветом и максимальное значение, которое можно считать черным и проверить соответствие им угловых точек. Также можно определить минимальную разницу между значениями черного и белого пикселей и максимальную разницу между двумя пикселями одного цвета и сверить разности значений угловых точек с ним. Добавим следующий код в функцию `search` внутрь вложенного цикла:

```
if(top_right_value < 50) {
                continue;
            }
            if(bottom_right_value > 150) {
                continue;
            }
            if(abs(top_left_value - bottom_right_value) > 30) {
            }
            if(abs(top_right_value - bottom_left_value) > 30) {
                continue;
            }
            if(top_right_value - bottom_right_value < 30) {
                continue;
            }
            if(bottom_left_value - top_left_value < 30) {
                continue;
            }
```

Нет необходимости проверять области, которые уже были обнаружены как часть метки. Учтем и это:

```
            for(size_t mark_number = 0; mark_number <
output_coordinates.size(); mark_number++) {
                if(x_bottom_right - min_mark_size >=
output_coordinates[mark_number][0].x &&
                   x_bottom_right - min_mark_size <
output_coordinates[mark_number][1].x &&
                   y_bottom_right - min_mark_size >=
output_coordinates[mark_number][0].y &&
```

```
                    y_bottom_right - min_mark_size <
output_coordinates[mark_number][1].y) {
                     skip = true;
                     break;
                    }

                if(x_bottom_right >=
output_coordinates[mark_number][0].x &&
                    x_bottom_right <
output_coordinates[mark_number][1].x &&
                    y_bottom_right >=
output_coordinates[mark_number][0].y &&
                    y_bottom_right <
output_coordinates[mark_number][1].y) {
                     skip = true;
                     break;
                    }
            }

            if(skip) {
                skip = false;
                continue;
            }
```

Зная предполагаемые значения черного и белого цветов, можно не считать среднее значение цвета всего фрагмента для определения порога бинаризации, а вычислить среднее значение между черным пикселем и белым:

```
double threshold_value = (top_right_value + bottom_right_value) / 2;
```

Исключительно с целью повышения удобства пользования написанной программой, изменим код входной функции `find_marks` так, чтобы минимальный размер метки был задаваемым параметром класса `Mark_Finder`, методами которого будут являться вышеописанные функции:

```
std::vector<std::vector<Point>> find_marks(const Mat input_image,
const int min_mark_size = 10)
{
    std::vector<std::vector<Point>> marks_coordinates = {};
    Mat img_grey;
    Mat img_blur;
```

```
    Mark_Finder finder(min_mark_size);

    cvtColor(input_image, img_grey, COLOR_BGR2GRAY);
    bilateralFilter(img_grey, img_blur, 5, 10, 10);
    marks_coordinates = finder.search(img_grey);

    return marks_coordinates;
}
```

Пример использования программы представлен ниже. Найденные на изображении метки выделяются квадратом синего цвета и измененное изображение выводится на экран:

```
int main()
{
    cv::Mat img = cv::imread("test.jpg");
    std::vector<std::vector<Point>> marks_coordinates;
    marks_coordinates = find_marks(img);

    for(size_t mark=0; mark < marks_coordinates.size(); mark++) {
        rectangle(img, marks_coordinates[mark][0],
marks_coordinates[mark][1], Scalar(255, 0, 0), 1);
    }

    cv::imshow("img", img);
    cv::waitKey(0);
    return 0;
}
```

Заключение

После последних изменений скорость работы программы превысила поставленную цель. Среднее время обработки одного кадра разрешение 720 на 1280 при минимальном размере метки 10 пикселей составило около 0,015 секунды. Точность распознавания также превысило требуемое. Из 456 изображений, на которых программа обнаружила метку, на 454 она была найдена верно, что соответствует значению точности $P \approx 0{,}996$. При необходимости программа позволяет влиять на данные параметры, задавая различные размеры минимального размера метки. Чем это значение больше, тем больше скорость обработки и точность, однако уменьшится расстояние, на котором метку можно обнаружить.

СПИСОК ЛИТЕРАТУРЫ

1. Laganiere R. [и др.]. OpenCV 4 Computer Vision Application Programming Cookbook: Build complex computer vision applications with OpenCV and C++ // Packt Publishing, 4th edition. 2019. С. 142 – 195.

2. OpenCV modules [Электронный ресурс]. URL: https://docs.opencv.org/4.x/index.html (дата обращения: 12.02.2024).

ПРИЛОЖЕНИЕ

Код программы

```
#include <stdio.h>
#include <chrono>
#include <map>
#include <opencv2/opencv.hpp>

using namespace std::chrono;
using namespace cv;


class Mark_Finder {
private:
    int min_mark_size;
    int half_mark_size;
    Mat perfect_mark;


    void init_perfect_mark();
    Mat apply_mask(const Mat);
    bool if_in_bounds(const Point, const Point, const int, const int);
    std::vector<int> calculate_offset(const Mat);
    std::vector<Scalar> get_edges_mean(Mat, Point, Point);
    std::vector<Point> get_mark_borders(Mat, Point, Point);

public:
    Mark_Finder();
    Mark_Finder(const int);
    std::vector<std::vector<Point>> search(const Mat scanning_image);
};


Mark_Finder::Mark_Finder() {
    min_mark_size = 10;
    half_mark_size = 5;
    init_perfect_mark();
}


Mark_Finder::Mark_Finder(const int size) {
    min_mark_size = size;
    half_mark_size = int(size / 2);
    init_perfect_mark();
}
```

```
void Mark_Finder::init_perfect_mark() {
    perfect_mark = Mat(min_mark_size, min_mark_size, CV_8UC1,
Scalar(255, 255, 255));
    rectangle(perfect_mark, Point(0, 0), Point(half_mark_size - 1,
half_mark_size - 1), Scalar(0, 0, 0), -1);
    rectangle(perfect_mark, Point(half_mark_size, half_mark_size),
Point(min_mark_size - 1, min_mark_size - 1), Scalar(0, 0, 0), -1);
}


Mat Mark_Finder::apply_mask(const Mat input_image) {
    Mat mark_xor;
    bitwise_xor(input_image, perfect_mark, mark_xor);
    return mark_xor;
}


bool Mark_Finder::if_in_bounds(Point top_left_point, Point
bottom_right_point, int frame_width, int frame_height) {
    if(top_left_point.x < 0) {
        return false;
    }
    if(top_left_point.y < 0) {
        return false;
    }
    if(frame_width - 1 < bottom_right_point.x) {
        return false;
}
    if(frame_height - 1 < bottom_right_point.y) {
        return false;
}
    return true;
}



std::vector<int> Mark_Finder::calculate_offset(const Mat input_image)
{
    int x_offset = 0;
    int y_offset = 0;
    Mat resized_xor;
    Mat resized_xor_line;
```

```
    Mat resized_line;

    Mat mark_xor = apply_mask(input_image);
    int difference = countNonZero(mark_xor);

    Mat sample = Mat(min_mark_size, min_mark_size, CV_8UC1, Scalar(0,
0, 0));
    Mat xor_line;
    int new_difference;
    int col = half_mark_size;
    int row = half_mark_size;

    // вычисление смещения по x (смещение вычисляется либо влево, либо
вправо)
    bool run = true;
    while(run) {
        line(sample, Point(col, 0), Point(col, min_mark_size - 1),
Scalar(255, 255, 255), 1);  // добавить белую линию
        bitwise_xor(mark_xor, sample, xor_line);
        line(sample, Point(col, 0), Point(col, min_mark_size - 1),
Scalar(0, 0, 0), 1);  // убрать белую линию
        new_difference = countNonZero(xor_line);

        if(col >= half_mark_size) {
            col++;
        }else {
            col--;
        }

        if(new_difference < difference) {
            mark_xor = xor_line.clone();
            difference = new_difference;
            if(col >= half_mark_size) {
                x_offset += 1;
            } else {
                x_offset -= 1;
            }
        } else {
            if(x_offset == 0 && col >= half_mark_size) {
                col = half_mark_size - 1;
            } else {
                run = false;
                break;
```

```
            }
        }

        if(col == min_mark_size - 1 || col == 0) {
            return {};
        }
    }

    // вычисление смещения по y (смещение вычисляется либо вверх, либо
вниз)
    run = true;
    while(run) {
        line(sample, Point(0, row), Point(min_mark_size - 1, row),
Scalar(255, 255, 255), 1);  // добавить белую линию
        bitwise_xor(mark_xor, sample, xor_line);
        line(sample, Point(0, row), Point(min_mark_size - 1, row),
Scalar(0, 0, 0), 1);  // убрать белую линию
        new_difference = countNonZero(xor_line);

        if(row >= half_mark_size) {
            row++;
        }else {
            row--;
        }

        if(new_difference < difference) {
            mark_xor = xor_line.clone();
            difference = new_difference;
            if(row >= half_mark_size) {
                y_offset += 1;
            } else {
                y_offset -= 1;
            }
        } else {
            if(y_offset == 0 && row >= half_mark_size) {
                row = half_mark_size - 1;
            } else {
                run = false;
                break;
            }
        }

        if(row == min_mark_size - 1 || row == 0) {
```

```
            return {};
        }
    }

    if(difference != 0) {
        // проверка на то, что все отклонения от метки находятся не
более чем в двух четвертях
        int bad_segments_counter = 0;

        Mat top_left_segment = mark_xor(Range(0, half_mark_size +
y_offset), Range(0, half_mark_size + x_offset));
        if(countNonZero(top_left_segment) != 0) {
            bad_segments_counter++;
        };
        Mat top_right_segment = mark_xor(Range(0, half_mark_size +
y_offset), Range(half_mark_size + x_offset, min_mark_size));
        if(countNonZero(top_right_segment) != 0) {
            bad_segments_counter++;
        };
        Mat bottom_left_segment = mark_xor(Range(half_mark_size +
y_offset, min_mark_size), Range(0, half_mark_size + x_offset));
        if(countNonZero(bottom_left_segment) != 0) {
            bad_segments_counter++;
        };
        Mat bottom_right_segment = mark_xor(Range(half_mark_size +
y_offset, min_mark_size), Range(half_mark_size + x_offset,
min_mark_size));
        if(countNonZero(bottom_right_segment) != 0) {
            bad_segments_counter++;
        };

        if(bad_segments_counter > 2) {
            return {};
        }


        // закрашивание пикселей на линиях, соседствующих с осями
симметрии метки
        line(mark_xor, Point(half_mark_size + x_offset - 1, 0),
            Point(half_mark_size + x_offset - 1, min_mark_size - 1),
            Scalar(0, 0, 0), 1);
        line(mark_xor, Point(half_mark_size + x_offset, 0),
            Point(half_mark_size + x_offset, min_mark_size - 1),
            Scalar(0, 0, 0), 1);
```

```
        line(mark_xor, Point(0, half_mark_size + y_offset - 1),
            Point(min_mark_size - 1, half_mark_size + y_offset - 1),
            Scalar(0, 0, 0), 1);
        line(mark_xor, Point(0, half_mark_size + y_offset),
            Point(min_mark_size - 1, half_mark_size + y_offset),
            Scalar(0, 0, 0), 1);
        difference = countNonZero(mark_xor);

        if(difference != 0) {
            return {};
        }
    }

    return {x_offset, y_offset};
}


std::vector<Scalar> Mark_Finder::get_edges_mean(Mat img, Point
top_left, Point bottom_right) {
    int top_middle_x = top_left.x + (bottom_right.x - top_left.x + 1)
/ 2;
    int right_middle_y = top_left.y + (bottom_right.y - top_left.y +
1) / 2;

    return {
        mean(img(Range(top_left.y, top_left.y + 1), Range(top_left.x,
top_middle_x))),
        mean(img(Range(top_left.y, top_left.y + 1),
Range(top_middle_x, bottom_right.x + 1))),
        mean(img(Range(top_left.y, right_middle_y),
Range(bottom_right.x, bottom_right.x + 1))),
        mean(img(Range(right_middle_y, bottom_right.y + 1),
Range(bottom_right.x, bottom_right.x + 1))),
        mean(img(Range(bottom_right.y, bottom_right.y + 1),
Range(top_middle_x, bottom_right.x + 1))),
        mean(img(Range(bottom_right.y, bottom_right.y + 1),
Range(top_left.x, top_middle_x))),
        mean(img(Range(right_middle_y, bottom_right.y + 1),
Range(top_left.x, top_left.x + 1))),
        mean(img(Range(top_left.y, right_middle_y), Range(top_left.x,
top_left.x + 1)))
    };
}
```

```
std::vector<Point> Mark_Finder::get_mark_borders(Mat scanning_image,
Point top_left, Point bottom_right) {
    int frame_width = static_cast<int>(scanning_image.cols);
    int frame_height = static_cast<int>(scanning_image.rows);

    std::vector<Scalar> edges_mean = get_edges_mean(scanning_image,
top_left, bottom_right);

    Point point_offset = Point(half_mark_size, half_mark_size);
    std::vector<Scalar> new_edges_mean;
    while(if_in_bounds(top_left, bottom_right, frame_width,
frame_height)) {
        top_left -= point_offset;
        bottom_right += point_offset;

        new_edges_mean = get_edges_mean(scanning_image, top_left,
bottom_right);

        for(int edge = 0; edge < 8; edge++) {
            if(abs((edges_mean[edge][0] - new_edges_mean[edge][0])) >
100) {
                return {top_left, bottom_right};
            }
        }
        edges_mean = new_edges_mean;
        if(bottom_right.x - top_left.x > 200) {
            return {};
        }
    }
    return {};
}


std::vector<std::vector<Point>> Mark_Finder::search(const Mat
scanning_image)
{
    std::vector<std::vector<Point>> output_coordinates = {};
    Mat crop_image;
    Mat crop_thresholded;
    Mat image_thresgolded;
    Range vertical_side;
    Range horizontal_side;
    double top_right_value;
    double bottom_right_value;
```

```
    double top_left_value;
    double bottom_left_value;
    bool skip = false;

    int frame_width = static_cast<int>(scanning_image.cols);
    int frame_height = static_cast<int>(scanning_image.rows);

    for(int y_bottom_right = min_mark_size; y_bottom_right <=
frame_height; y_bottom_right += half_mark_size-1) {  // шаг в половину
размера минимальной метки
        for(int x_bottom_right = min_mark_size; x_bottom_right <=
frame_width; x_bottom_right += half_mark_size-1) {

            vertical_side = Range(y_bottom_right - min_mark_size,
y_bottom_right);
            horizontal_side = Range(x_bottom_right - min_mark_size,
x_bottom_right);

            crop_image = scanning_image(vertical_side,
horizontal_side);

            top_right_value = crop_image.at<uchar>(0, min_mark_size-
1);
            bottom_right_value = crop_image.at<uchar>(min_mark_size-1,
min_mark_size-1);
            top_left_value = crop_image.at<uchar>(0, 0);
            bottom_left_value = crop_image.at<uchar>(min_mark_size-1,
0);

            // проверка углов на соответствие метке
            if(top_right_value < 50) {  // минимальное значение белого
цвета
                continue;
            }
            if(bottom_right_value > 150) {  // максимальное значение
черного цвета
                continue;
            }
            if(abs(top_left_value - bottom_right_value) > 30) {  //
близкое значение черных областей
                continue;
            }
```

```
            if(abs(top_right_value - bottom_left_value) > 30) {  //
близкое значение белых областей
                continue;
            }
            if(top_right_value - bottom_right_value < 30) {  //
различие значения черной и белой области справа
                continue;
            }
            if(bottom_left_value - top_left_value < 30) {  // различие
значения черной и белой области слева
                continue;
            }

            for(size_t mark_number = 0; mark_number <
output_coordinates.size(); mark_number++) {
                if(x_bottom_right - min_mark_size >=
output_coordinates[mark_number][0].x &&
                   x_bottom_right - min_mark_size <
output_coordinates[mark_number][1].x &&
                   y_bottom_right - min_mark_size >=
output_coordinates[mark_number][0].y &&
                   y_bottom_right - min_mark_size <
output_coordinates[mark_number][1].y) {
                    skip = true;
                    break;
                   }

                if(x_bottom_right >=
output_coordinates[mark_number][0].x &&
                   x_bottom_right <
output_coordinates[mark_number][1].x &&
                   y_bottom_right >=
output_coordinates[mark_number][0].y &&
                   y_bottom_right <
output_coordinates[mark_number][1].y) {
                    skip = true;
                    break;
                   }
            }

            if(skip) {
                skip = false;
                continue;
```

```
            }

            double threshold_value = (top_right_value +
bottom_right_value) / 2;
            threshold(crop_image, crop_thresholded, threshold_value,
255, THRESH_BINARY);  // бинаризация в соответсвии со значениями в
углах

            std::vector<int> offset =
calculate_offset(crop_thresholded);  // вычисление смещения центра
метки от центра сканера (и заодно ее проверка)
            if(offset.empty()) {
                continue;
            }

            Point center_top_left = Point(horizontal_side.start +
offset[0], vertical_side.start + offset[1]);
            Point center_bottom_right = Point(horizontal_side.end +
offset[0], vertical_side.end + offset[1]);
            std::vector<Point> min_mark_coordinates =
{center_top_left, center_bottom_right};

            threshold(scanning_image, image_thresgolded,
threshold_value, 255, THRESH_BINARY);
            std::vector<Point> mark_points =
get_mark_borders(image_thresgolded, center_top_left,
center_bottom_right);
            if(mark_points.empty()) {
                continue;
            }
            output_coordinates.push_back(mark_points);
        }
    }
    return output_coordinates;
}


std::vector<std::vector<Point>> find_marks(const Mat input_image,
const int min_mark_size = 10)
{
    std::vector<std::vector<Point>> marks_coordinates = {};
    Mat img_grey;
    Mat img_blur;
```

```
    Mark_Finder finder(min_mark_size);

    cvtColor(input_image, img_grey, COLOR_BGR2GRAY);
    bilateralFilter(img_grey, img_blur, 5, 10, 10);
    marks_coordinates = finder.search(img_grey);

    return marks_coordinates;
}
```