

Министерство науки и высшего образования Российской Федерации  
Санкт-Петербургский политехнический университет Петра Великого  
Высшая школа прикладной математики и вычислительной физики

Работа допущена к защите  
Директор высшей школы  
\_\_\_\_\_ Уткин Л.В.  
«\_\_\_\_\_» 2020 г.

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА  
ДИПЛОМНАЯ РАБОТА  
ПОИСК ПЕРЕСЕЧЕНИЙ ЛИНИЙ НА ИЗОБРАЖЕНИИ**

по направлению подготовки 01.03.02 Прикладная математика и информатика  
Направленность (профиль) 01.03.02\_04 Системное программирование

Выполнил  
студент гр. 3630102/60401

Туников Д.А.

Руководитель  
доцент,  
к.б.н.

Шубников В.Г.

Санкт-Петербург  
2020

## **РЕФЕРАТ**

38 с., 36 рисунков, 0 таблиц, 0 приложений.

### **ПРЕОБРАЗОВАНИЕ ХАФА, МАШИНА ОПОРНЫХ ВЕКТОРОВ, ГИСТОГРАММА ОРИЕНТИРОВАННЫХ ГРАДИЕНТОВ**

В данной работе были рассмотрены два алгоритма поиска пересечений линий на изображении: алгоритм на основе преобразования Хафа для прямых и алгоритм с обучением машины опорных векторов. В качестве прикладной проблемы решалась задача поиска железнодорожных стрелок на изображениях, сделанных с головы локомотива. В процессе работы был создан датасет из 223 размеченных изображений. Данный датасет использовался для проверки качества работы обоих алгоритмов и обучения машины опорных векторов.

## **ABSTRACT**

38 p., 36 figures, 0 tables, 0 appendices.

### **HOUGH TRANSFORM, SUPPORT VECTOR MACHINE, HISTOGRAM OF ORIENTED GRADIENTS**

This paper introduce two algorithms for searching the intersections of lines in an image: algorithm based on the Hough transform for straight lines and algorithm with training the support vector machine. As an applied problem, searching for railway arrows on images taken from the head of a locomotive was solved. Dataset of 223 labeled images was created. This dataset was used to check the quality of both algorithms and to train the support vector machine.

## СОДЕРЖАНИЕ

Введение .....	4
Глава 1. Введение	4
1.1. Постановка задачи .....	5
1.2. Обзор литературы .....	6
1.2.1. Поиск объекта по его геометрическим свойствам .....	6
1.2.2. Поиск объекта по ключевым точкам .....	14
1.2.3. Обучение классификатора на размеченных данных .....	15
Глава 2. Основная часть	19
2.1. Метод, основанный на преобразовании Хафа.....	19
2.1.1. Преобразование Хафа.....	19
2.1.2. Алгоритм метода.....	20
2.1.3. Поиск границ на изображении.....	20
2.1.4. Разбиение изображения на горизонтальные блоки .....	22
2.1.5. Поиск рельсов .....	23
2.1.6. Поиск стрелок .....	25
2.1.7. Описание данных для тестирования.....	31
2.1.8. Результаты .....	32
Список литературы.....	37
Литература	37

## ГЛАВА 1. ВВЕДЕНИЕ

Проблема поиска пересечений линий на изображении является очень распространенной и её решение может быть использовано в таких задачах компьютерного зрения как:

- A. Автоматическое определение полей во время спортивных трансляций
- B. Детектирование ориентиров при автоматическом управлении роботом
- C. Детектирование железнодорожных стрелок на изображении, сделанном с локомотива поезда

В данной работе будут рассмотрены подходы к решению задачи применительно к детектированию ж/д стрелок.

Для автоматизации процесса управления поездом необходимо создать систему, которая позволяла бы стать надежным автоматическим ассистентом/помощником для машиниста локомотива (стать неким driving assistance), но ни в коем случае не заменить действия человека полностью. Одной из задач такой системы является обнаружение железнодорожных стрелок для обеспечения движения поезда по правильному пути. Решение этой задачи позволит обнаруживать разветвления пути заранее (визуально) и анализировать возможные дальнейшие пути движения поезда, выдавая полезную информацию для машиниста локомотива, такую как: "на правой ветке обнаружены вагоны - путь занят!". Это решение значительно повысит безопасность движения на железных дорогах.

## 1.1. Постановка задачи

На вход подается изображение — фотография сделанная с головы поезда. Необходимо найти на входном изображении места пересечений железно-дорожных путей. Пересечением будем называть точку, в которой пересекаются рельсы одной ж/д колеи с другой. На выходе алгоритма должен быть набор точек, в которых было обнаружено пересечение.



Рис.1.1. Пример входного изображения

Замечания:

- A. Точность поиска стрелок - окрестность от 20x20 до 50x50 пикселей(в зависимости от дальности стрелки)
- B. Необходимо правильно находить стрелки в пределах 20-30 метрах перед поездом. Нет задачи детектировать стрелки, которые находятся слишком далеко от поезда(40 метров и далее).

## **1.2. Обзор литературы**

Существует несколько глобальных подходов к поиску объектов на изображении:

- A. Сопоставление границ, градиентов и чёрно-белых пикселей изображения. Этот подход устойчив к изменению света, но не устойчив к изменению положения объекта на изображении(вращение, угол наклона).[9].
- B. Поиск объекта по геометрическим свойствам. Если форма искомого объекта может быть задача аналитически, то такие объекты можно искать с помощью преобразования Хафа [1].
- C. Подход, основанный на сопоставлении признаков ключевых точек изображения с ключевыми точками искомого объекта. Данный подход реализован в следующих методах: SIFT[8], SURF[10].
- D. Обучение классификатора на размеченных данных. В роли классификатора может быть: SVM классификатор[11], дерево решений [12], KNN классификатор [13]. В качестве входных векторов для классификаторов могут быть использованы различные виды признаков, полученные из размеченных изображений. Например, гистограмма ориентированных градиентов(HOG)[7], SURF[8]/SIFT[10] дескрипторы и так далее.

Рассмотрим данные подходы подробнее.

### ***1.2.1. Поиск объекта по его геометрическим свойствам***

Для поиска объектов, геометрическая форма которых может быть задана некоторым уравнением(н-р прямая, круг, эллипс) используется следующий подход:

- A. Из изображения извлекаются границы. Это можно сделать с помощью алгоритма Canny [2].

Пример применения алгоритма Canny к входному изображению:

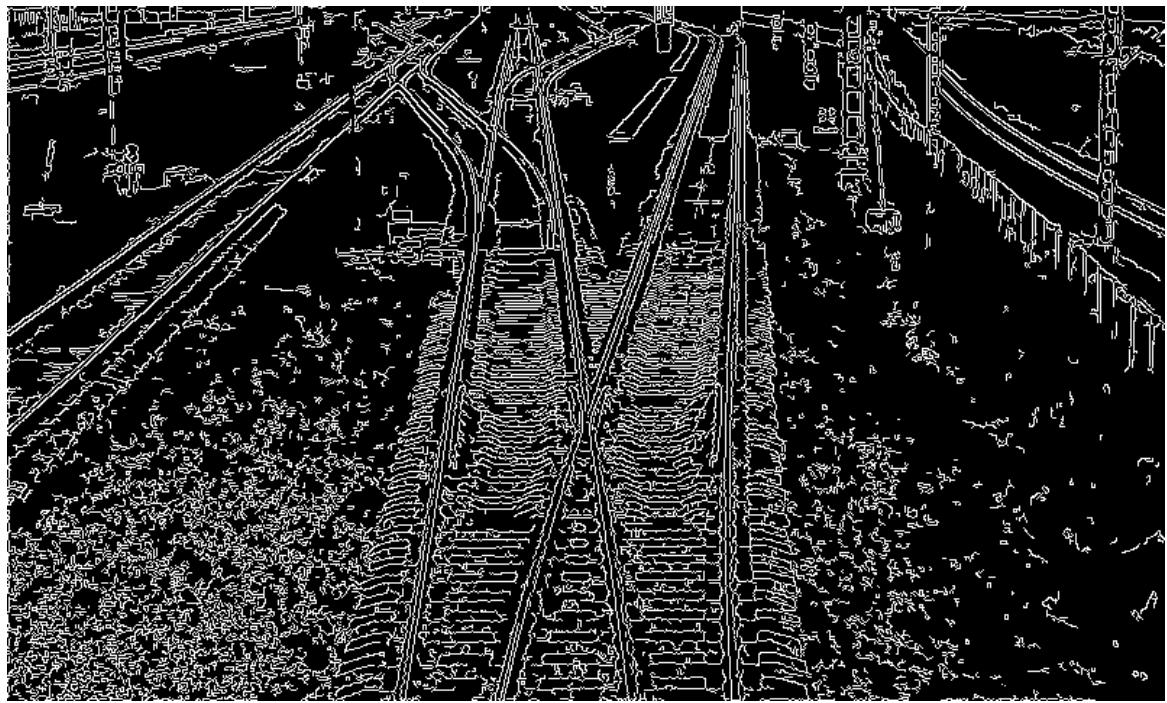


Рис.1.2

В. После чего к полученным границам применяется алгоритма Хафа 2.1.1. Если целью является поиск прямых, то алгоритм Хафа даст следующий результат(красным обозначены найденные линии):

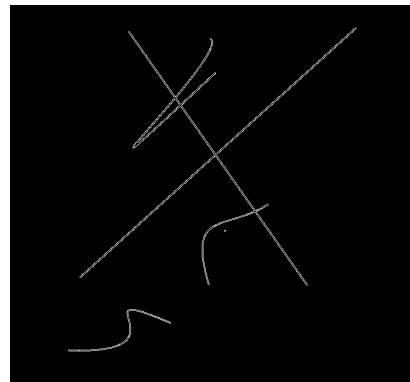


Рис.1.3

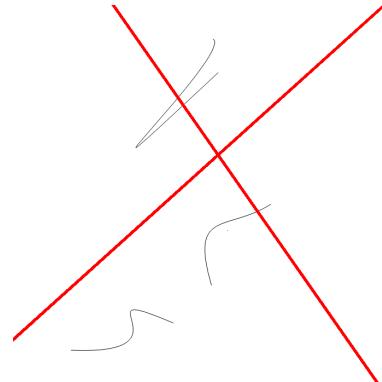


Рис.1.4

Таким образом, если требуется искать пересечения прямых линий на изображении, то для этого отлично подойдет следующий алгоритм:

- Поиск границ [2]
  - Поиск линий 2.1.1
  - Поиск пересечений линий, найденных в предыдущем шаге. Это можно сделать аналитически. Для каждой пары найденных прямых составляем систему уравнений, тогда если эти прямые не параллельны, то решением системы будет точка пересечения прямых.
- $$\begin{cases} a_1x + b_1y = c_1 \\ a_2x + b_2y = c_2 \end{cases}$$

## Предыдущие работы, посвященные поиску ж/д путей и их пересечений с помощью применения геометрически ориентированного подхода

**Vision based rail track and switch recognition for self-localization of trains in a rail network [6].** Подход, описанный в данной работе основан на анализе изображения с камеры, стоящей в голове поезда. Условиями для корректного определения стрелок на железной дороге является то, что расстояние между параллельными рельсами всегда одинаковое, а также радиус кривизны рельсов достаточно большой.

В данной работе съёмка ж/д путей происходит достаточно часто, и к каждому вновь сделанному снимку применяется следующий алгоритм:

1. Из изображения вырезается близкая к поезду полоса.



Рис.1.5

2. К выделенному участку применяется алгоритм поиска ребер:
  - А. Рассчитывается вектор градиента в каждой точке изображения
  - Б. Далее рассматриваются окрестности по 3x3 пикселя. И для каждого пикселя с величиной градиента большей, чем у половины соседей в окрестности выполняется следующая процедура: Строится матрица ковариаций данного пикселя с соседями в окрестности 3x3.  $S = \begin{pmatrix} Cov_{xx} & Cov_{xy} \\ Cov_{yx} & Cov_{yy} \end{pmatrix}$ ,  $Cov_{ab} = \frac{1}{n} \sum_1^N (g_a, g_b)$
  - С. Вычисляются собственные числа этой матрицы, и если выполнены неравенства:  $\begin{cases} \lambda_1 \geq t_l \\ \lambda_2 \geq m_l \\ \frac{\lambda_2}{\lambda_1} \leq m_l \end{cases}$ , где  $t_l, m_l$  вычисленные эмпирическим путём константы, то

данный пиксель считается краевым. Также из матрицы ковариации получаются два собственных вектора:  $e_1, e_2$ , больший из которых перпендикулярен краю, а меньший параллелен краю в данной точке.

В итоге границы найденные границы выделенного блока выглядят следующим образом:



Рис.1.6

### 3. Поиск прямых.

Строится массив аккумулятора  $A(\theta, x)$ (по аналогии с алгоритмом Хафа[1]). И для каждого краевого пикселя( $K$ ) вычисляется угол  $\theta$  между большим собственным вектором  $e_2$  и осью ОY. Также необходимо вычислить значение  $x_c$ . Строится дополнительная прямая  $y_c = \frac{H}{2}$ , где  $H$  - высота рассматриваемого изображения. Тогда  $x_c$  будет координатой  $x$  точки пересечения построенной прямой  $y_c$  и продолжения собственного вектора  $e_2$ . Увеличиваем значения аккумулятора  $A(\theta, x_c) = A(\theta, x_c) + 1$ . В итоге максимумы аккумулятора будут соответствовать параметрам прямых линий на изображении. На рисунке найденные линии обозначены черными точками.

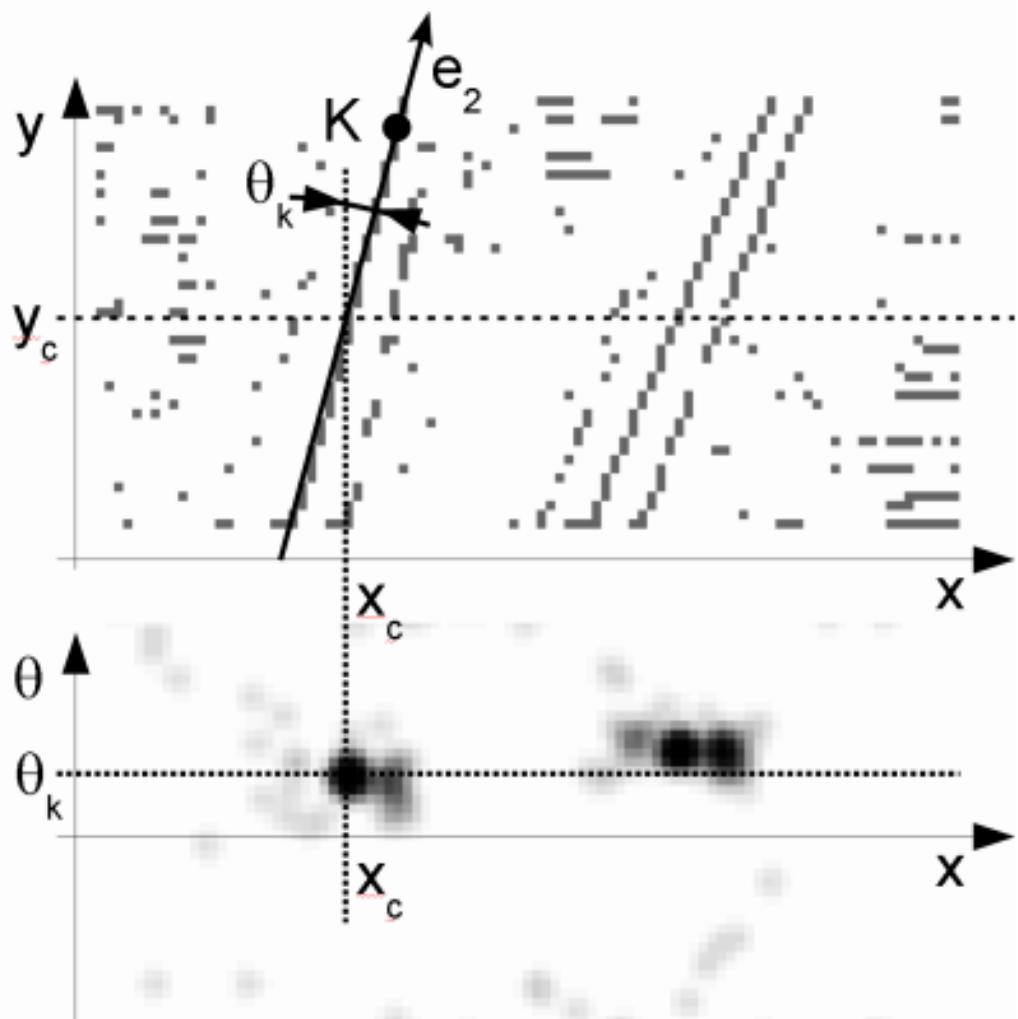


Рис.1.7

Данный алгоритм работает значительно быстрее преобразования Хафа(ускорение достигается за счет сохранения времени, которое в алгоритме Хафа тратится на перебор  $\theta \in [0, \pi]$ ). При этом алгоритм качественно находит прямые линии, проходящие через центральную вертикальную ось изображения.

#### 4. Выделение ж/д колеи по найденным прямым.

В данном алгоритме предполагается, что ширина железнодорожной колеи заранее зафиксирована и приходит на вход алгоритму.

Поэтому чтобы найти пары рельсов, принадлежащих к одной колее нужно выполнить перебор по всем найденным рельсам, сравнивая при этом расстояние между ними с эталонным расстоянием между рельсами(с некоторой погрешностью  $\delta R$ ).

Но нельзя забывать про то, что изображение делается с камеры и нужно учитывать перспективную проекцию [4] точек изображения на точки в реальном мире, чтобы корректно сравнивать расстояние между рельсами на изображении с

расстоянием в мировой системе координат. В итоге результат поиска ж/д колеи следующий(на рисунке выделен горизонтальный блок и линии, соответствующие центрам найденных пар рельсов):



Рис.1.8

##### 5. Определение стрелок.

При поиске пересечений найденных рельсов учтем, что нам известен минимальный радиус кривой, по которой может двигаться поезд(является параметром алгоритма). Из данного радиуса можно вычислить максимальное боковое отклонение  $s_1$ , на которое должен отклониться центр колеи, чтобы считаться НЕ основной колеёй.

Таким образом, мы ожидаем, что одна из наблюдавшихся нами колея всегда имеет боковое отклонение от центра поезда принадлежащее интервалу  $[-s_1, s_1]$ . Если центр колеи становится  $< -s_1$ , то это означает, что данная колея является левой веткой основной колеи, в обратном случае правой веткой. В момент времени, когда происходит переход данной границы мы можем отметить, что поезд прошел стрелку.

На рисунке обозначены границы  $s_1$ (теоретическая граница) и  $s_2$ (экспериментальная граница), где  $s_2 = s_1 + \delta s$ (подобрана экспериментально). Пересечение фиксируется в момент времени, когда центр колеи переходит границу  $|s_2|$ (это место показано стрелками на рисунке).

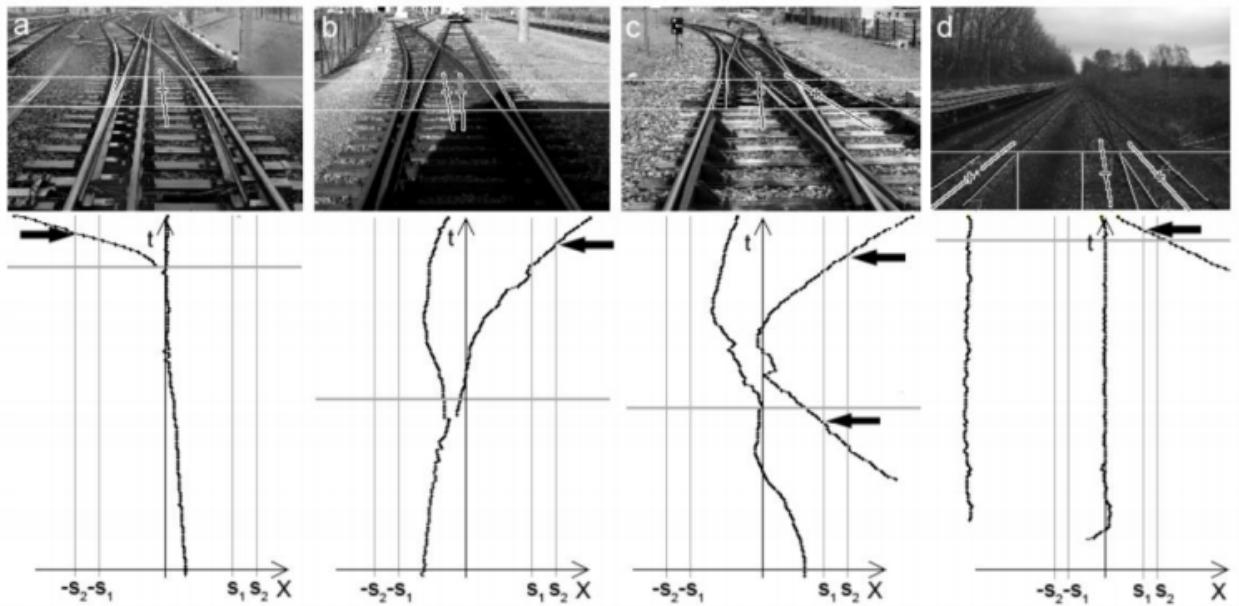


Рис.1.9

На левом рисунке видно, что есть отклоняющаяся налево колея и поезд пойдет прямо. На втором слева рисунке видно, что линия соответствующая прямой колее отклоняется вправо, следовательно поезд прошёл по левой колее и т.д.

### **Efficient railway tracks detection and turnouts recognition method using HOG features [5]**

В данной работе описывается метод поиска железнодорожных путей с использованием Histogram of oriented gradients[7](гистограмма ориентированных градиентов).

Алгоритм следующий:

1. Входное изображение разбивается на сетку, с уменьшением размера блока в зависимости от координаты Y на изображении(это сделано чтобы имитировать перспективную проекцию).



Рис.1.10

2. Начиная с нижней строки сетки применяется алгоритм growing up, который на основе похожести HOG соседних верхних клеток расширяет множество схожих кусочкой изображения. В итоге на выходе алгоритма growing up имеем множество рельсов(на рисунке обозначены разными цветами).

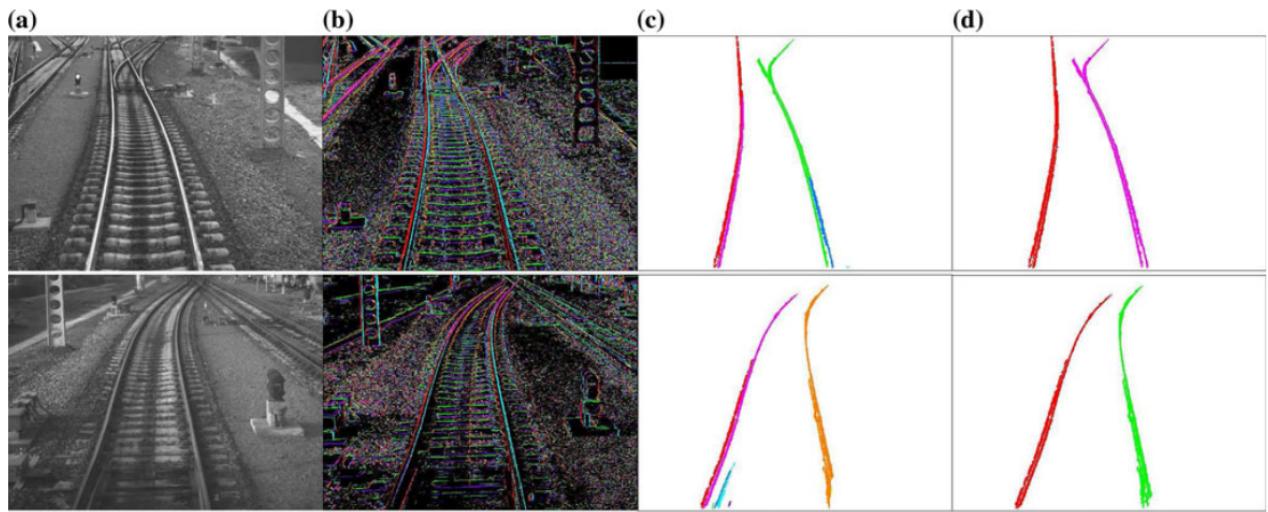


Рис.1.11

### 1.2.2. Поиск объекта по ключевым точкам

Одним из методов поиска объектов на изображении является построение SIFT[8] дескриптора и сопоставление этого дескриптора с SIFT дескриптором искомого объекта.

Сначала изображение преобразуется в большой набор векторов признаков(SIFT дескриптор), каждый из которых инвариантен относительно параллельного переноса изображения, масштабирования и вращения, частично инвариантен изменению освещения и устойчив к локальным геометрическим искажениям.

После чего SIFT дескриптор текущего изображения и искомого объекта сопоставляются и на выходе можно увидеть соответствие ключевых точек на изображении и ключевых точек искомого объекта:

Если говорить о применении техники SIFT для поиска пересечений ж/д путей на изображении. То можно создать некоторую базу с различными видами стрелок и их SIFT дескрипторами.

Тогда алгоритм для поиска пересечений на входном изображении будет следующий:

- A. Вычислять SIFT дескриптор входного изображения
- B. Сопоставить его с каждым из SIFT дескрипторов, находящихся в базе данных.
- C. Если было найдено достаточно точное сопоставление - выполнить уточнение места пересечения и добавить его в выходной результат

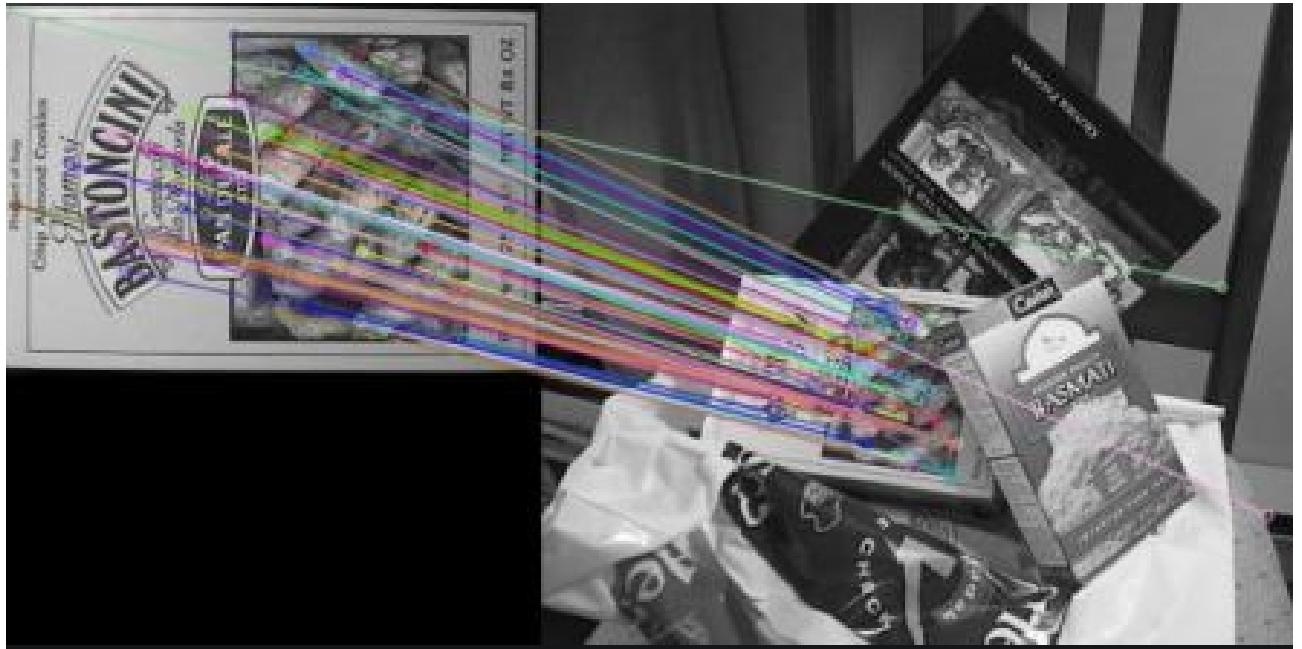


Рис.1.12

### ***1.2.3. Обучение классификатора на размеченных данных***

Распространенным способом поиска объекта на изображении является обучение классификатора на размеченных данных.

Рассмотрим классификацию с помощью SVM(машина опорных векторов)[11]. Постановка задачи:

Необходимо классифицировать данные, каждый объект которых представляется как точка  $x^p$  в  $p$ -мерном пространстве. Каждая из точек принадлежит одному из двух классов. Необходим построить гиперплоскость размерности  $p - 1$ , которая линейно разделяла бы входные точки. Таких гиперплоскостей может быть много, но необходимо найти ту, расстояние до которой от любого класса был бы максимальным. Что эквивалентно тому, что сумма расстояний от крайних точек классов до плоскости максимальна. Если такая гиперплоскость существует, она называется оптимальной разделяющей гиперплоскостью, а соответствующий ей линейный классификатор называется оптимально разделяющим классификатором.

На вход алгоритма подается набор пар:  $[(x_1, c_1), (x_1, c_1), \dots, (x_n, c_n)]$

Уравнение искомой гиперплоскости плоскости имеет вид:  $wx - b = 0$ , где  $w$  - перпендикуляр к разделяющей плоскости, а  $b$  - расстояние от плоскости до начала координат.

$wx - b = 1$  и  $wx - b = -1$ , соответствуют плоскостям, проходящим через крайние точки классов:

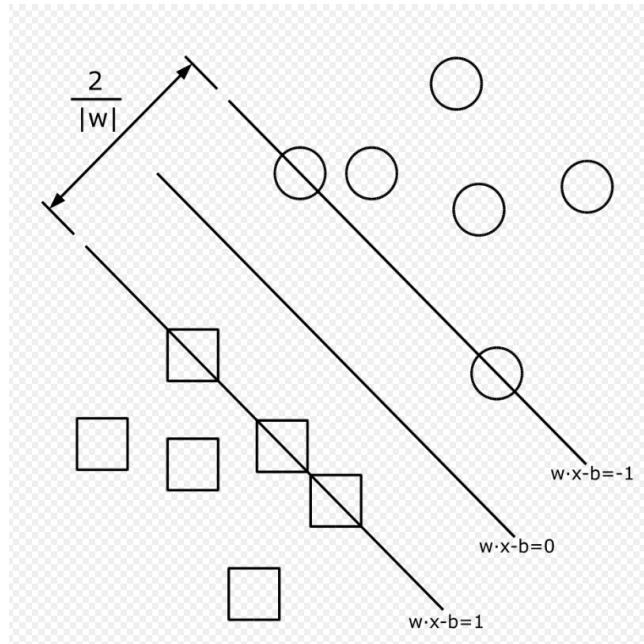


Рис.1.13

Для полного разделения классов необходимо чтобы для всех  $i \in [1, n]$  были выполнены неравенства:

$$\begin{cases} wx_i - b \geq 1, c_i == 1 \\ wx_i - b \leq -1, c_i == -1 \end{cases} \quad (1.1)$$

Тогда для того, чтобы найти оптимальную гиперплоскость нужно минимизировать  $\|w\|$ , при условиях 1.1, что соответствует такой задаче минимизации:

$$\begin{cases} \|w\|^2 > \min \\ c_i(wx_i - b) \geq 1, i \in [1, n] \end{cases} \quad (1.2)$$

В случае, если классы не разделимы линейно, вводится дополнительный параметр  $\varepsilon_i \geq 0$ , характеризующий ошибку на объектах  $x_i$ . Таким образом, смягчим ограничение во втором уравнении системы 1.2 и введём штраф за ошибку  $C$  - данный параметр позволяет регулировать соотношение между максимизацией

ширины разделяющей полосы и минимизацией ошибки:

$$\begin{cases} \frac{1}{2} \|w\|^2 + C \sum_{i=1}^n \varepsilon_i \rightarrow \min_{wb\varepsilon_i} \\ c_i(wx_i - b) \geq 1 - \varepsilon_i, i \in [1, n] \\ \varepsilon_i \geq 0 \end{cases} \quad (1.3)$$

Теперь, понимая математический аппарат работы SVM рассмотрим процесс построения классификатора для поиска объекта на изображении.

Входными векторами для SVM могут выступать различные виды признаков искомого объекта. Например, гистограмма ориентированных градиентов(HOG)[7], SIFT[8]/SURF[10] дескрипторы. Таким образом для решения задачи поиска объекта с помощью SVM необходимо выполнить следующие шаги:

- A. Разметить некоторое количество искомых объектов на изображении. Такие объекты будем относить к первому классу.
- B. Сгенерировать(или разметить) некоторое количество отрицательных примеров, которые точно не являются искомым объектом. Такие объекты отнесем ко второму классу.
- C. Рассчитать вектор признаков по которым мы будем обучать классификатор в каждой из размеченных(сгенерированных) точках изображения.
- D. Построить разделяющую объекты разных классов гиперплоскость, используя посчитанные в предыдущем пункте вектора признаков объектов.
- E. Для нахождения объекта на входном изображении, используя технику "скользящее окно"[14] рассчитать вектор признаков во всевозможных окнах и определить по какую сторону от гиперплоскости находится вычисленный вектор. Таким образом в итоге получится некоторый набор окон, в которых был найден искомый объект:

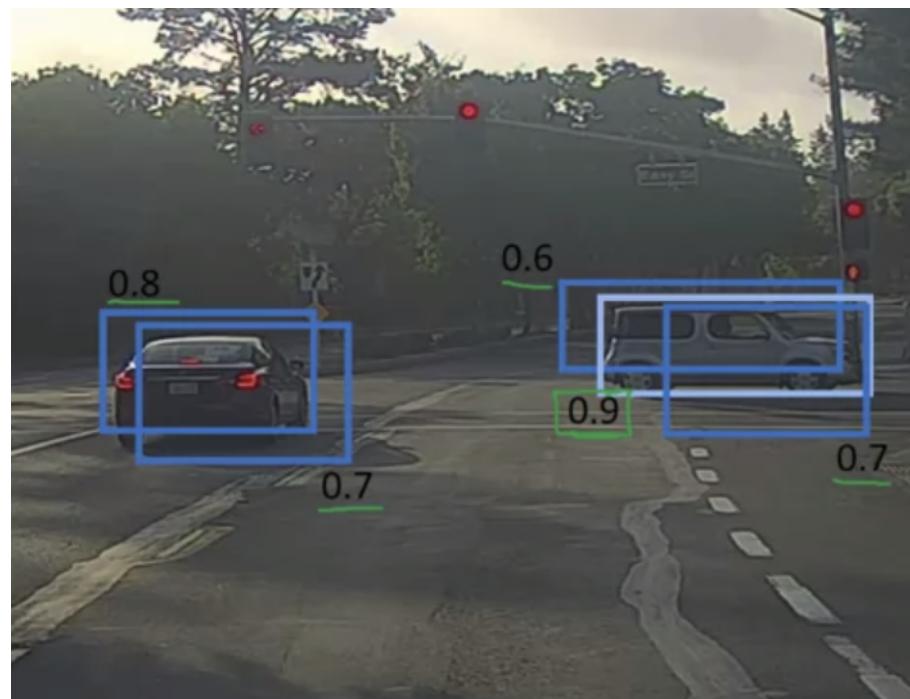


Рис.1.14

F. Из найденных окон выбрать наиболее точно описывающее искомый объект. Это можно сделать выбрав окно с максимальной вероятностью нахождения искомого объекта. Например, на рисунке 1.14 для правой машины это будет окно с вероятностью 0,9.

## ГЛАВА 2. ОСНОВНАЯ ЧАСТЬ

### 2.1. Метод, основанный на преобразовании Хафа

#### 2.1.1. Преобразование Хафа

Преобразование Хафа [1] - алгоритм для поиска геометрических объектов на изображении(линии, круги и т.д). В нашем случае понадобится алгоритм Хафа для поиска прямых линий на изображении.

На вход алгоритму подается набор точек, на выходе имеется набор прямых в координатах  $(\rho, \theta)$ , где  $\rho$  - расстояние от начала координат до прямой,  $\theta$  - угол между перпендикуляром к прямой, проведенным из начала координат, и осью абсцисс.

Алгоритм:

Имеется массив аккумулятор  $A[M, N]$ , где  $M$  - квантованные значения параметра  $\rho$ ,  $N$  - квантованные значения параметра  $\theta$ . Изначально в каждой ячейке  $A$  находится значения ноль.

Через каждую входную точку проводятся прямые с различными параметрами  $\theta \in [0, \pi]$ . 2.1

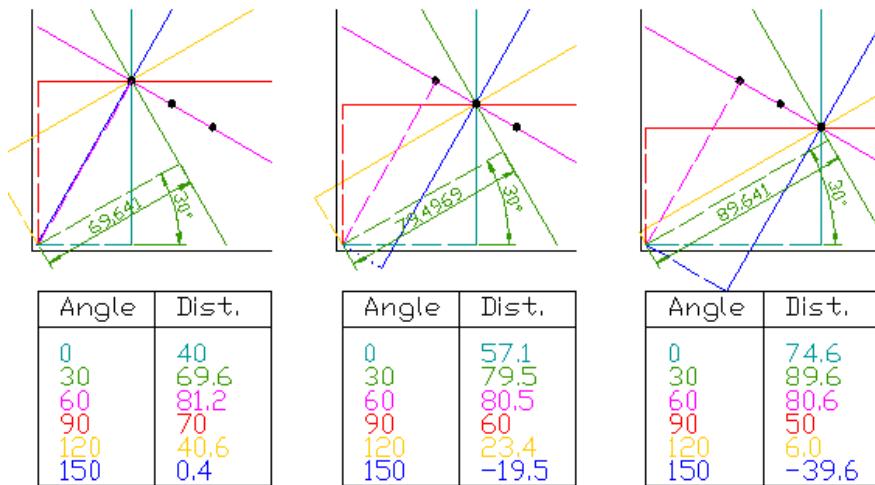


Рис.2.1

Каждая прямая голосует за свои параметры  $(\rho, \theta)$ , то есть значение значения аккумулятора  $A[\rho, \theta]$  увеличивается на единицу. После голосования максимумы в массиве аккумуляторе соответствуют параметрам итоговых прямых.

### 2.1.2. Алгоритм метода

### 2.1.3. Поиск границ на изображении

Первым шагом метода является поиск границ на входном изображении. Это делается с помощью алгоритма Canny [2]. Алгоритм состоит из пяти важных шагов:

- A. Сглаживание изображения для удаления шумов. Это делается фильтрацией изображения с использованием фильтра Гаусса:

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \exp \frac{-(x^2 + y^2)}{2\sigma^2}$$

- B. Поиск градиентов. Границы выделяются там, где находятся максимумы градиентов. Градиенты ищутся с помощью оператора Собеля, который является аналогом производной в дискретном пространстве. Псевдокод поиска градиентов с помощью оператора Собеля:

$$\text{MGx} := \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix} \quad \text{MGy} := \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

Реализация (сопоставляет каждой точке вектор градиента):

```
SobelOperator(Matrix) := | for iY ∈ 1..rows(Matrix) - 2
                           |   for iX ∈ 1..cols(Matrix) - 2
                           |     A ← submatrix(Matrix, iY - 1, iY + 1, iX - 1, iX + 1)
                           |     GX ← ∑_{y=0}^2 ∑_{x=0}^2 (A_{y,x} MGx_{y,x})
                           |     GY ← ∑_{y=0}^2 ∑_{x=0}^2 (A_{y,x} MGy_{y,x})
                           |     G ← √(GX² + GY²)
                           |     θ ← round(atan2(GX, GY) / (π/4)) · π/4 - π/2 if G ≠ 0
                           |     θ ← ErrCode otherwise
                           |     SobMtrix_{iY, iX} ← G
                           |     SobMtrix_{iY, iX+1+cols(Matrix)} ← θ
                           |
                           | return SobMtrix
```

Рис.2.2

- C. Подавление не-максимумов. Среди всех максимумов оставляем только те, что являются локальными максимумами в окрестности.
- D. Двойная пороговая фильтрация. Если значения градиента в точке максимума ниже порога - эта точка отсекается. Таким образом, чем выше значение порога, тем меньше границ будет найдено на изображении. Фильтрация в алгоритме Canny является двойной, лучше всего этот момент разъясняет иллюстрация 2.3, здесь только зелёные точки будут выбраны граничными. В нашем случае значение верхнего порога вычисляется с помощью бинари-

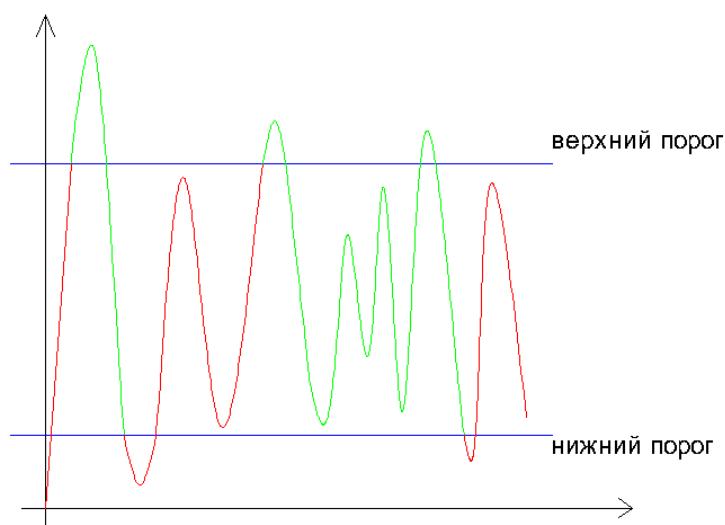


Рис.2.3

зации Отцу[3]. Значение нижнего порога берется как половина от значения верхнего порога. За счёт такого выбора порога достигается нормальное выявление границ на изображения с различным уровнем освещенности.

Е. Трассировка области неоднозначности. Итоговые границы определяются путём удаления всех краёв, несвязанных с "сильными" границами. Проще говоря, пиксели, которые не относятся ни к какой группе - подавляются.

Пример применения алгоритма Canny к входному изображению:

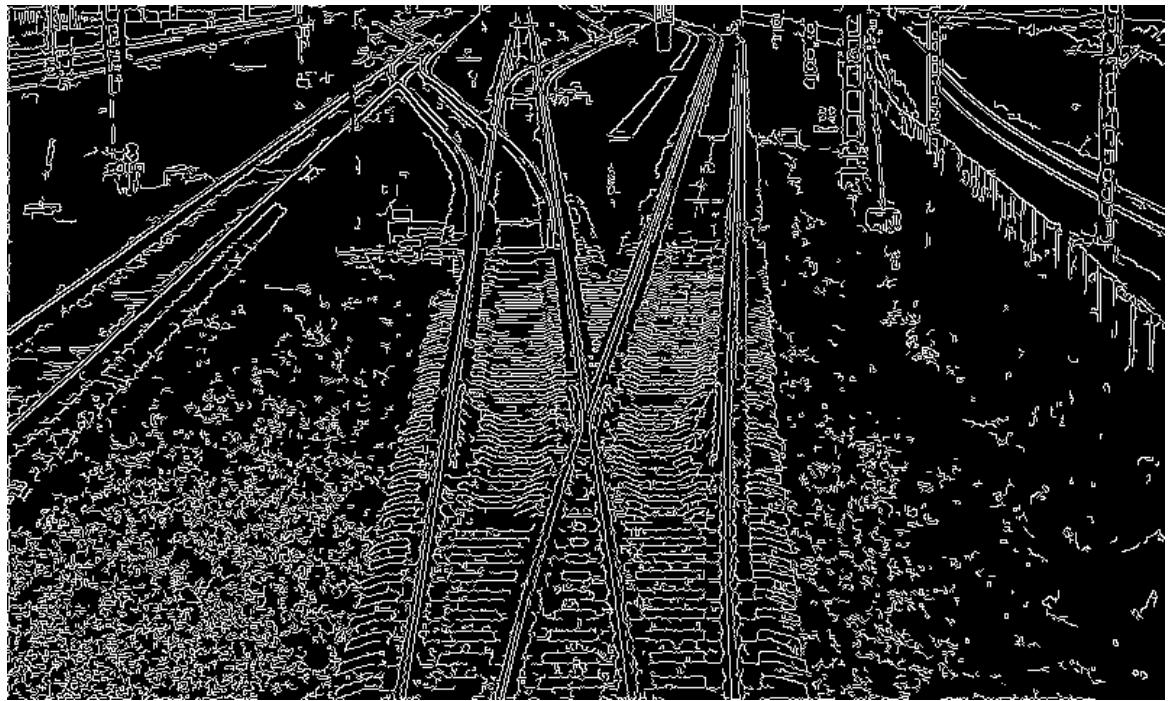


Рис.2.4

#### **2.1.4. Разбиение изображения на горизонтальные блоки**

Сначала разбиваем изображение на горизонтальные блоки 2.5. Размер блоков уменьшается в зависимости от координаты Y на изображении, это сделано для учета перспективной проекции объектов из реального мира на изображении - чем дальше объект, тем меньше он на изображении.

Формула для расчёта размера блока в зависимости от координаты y:

$$\text{blockSize} = \text{minBlockSize} + \frac{\text{maxBlockSize} - \text{minBlockSize}}{\text{imageHeight} - \text{maxBlockSize}} y, \text{ где } \text{minBlockSize} = 7\text{pix}, \text{maxBlockSize} = 45\text{pix}$$

Данные значения были выбраны из соображений минимизации процента ошибочных предсказаний стрелок и максимизации процента предсказания реальных стрелок. В идеале размер горизонтального блока нужно рассчитывать с использованием правил перспективного проецирования[4], но для этого необходимо знать параметры камеры(угол наклона, расстояния до земли). В нашем случае эти параметры не известны, поэтому blockSize рассчитывается просто с помощью линейной интерполяции в зависимости от координаты Y.



Рис.2.5

### 2.1.5. Поиск рельсов

Теперь рассмотрим процесс поиска рельсов на изображении, полученном после применения алгоритма Canny2.4.

После разбиения 2.5 в каждом из горизонтальных блоков ищутся прямые линии с помощью алгоритма Хафа [1]. Можно заметить, что прямые близкие к горизонтальным можно отсекать, т.к. они не могут относиться к рельсам. Таким образом, будем искать прямые с помощью модифицированной алгоритма Хафа, в котором  $\theta \in [0; \frac{\pi}{3}] \cup [\frac{2\pi}{3}; \pi]$ . В качестве порогового значения аккумулятора будем использовать:  $\frac{2 \cdot blockSize}{3}$ , это нужно для отсечения маленьких прямых.

Результат применения алгоритма(красным выделены найденные прямые линии в каждом блоке):

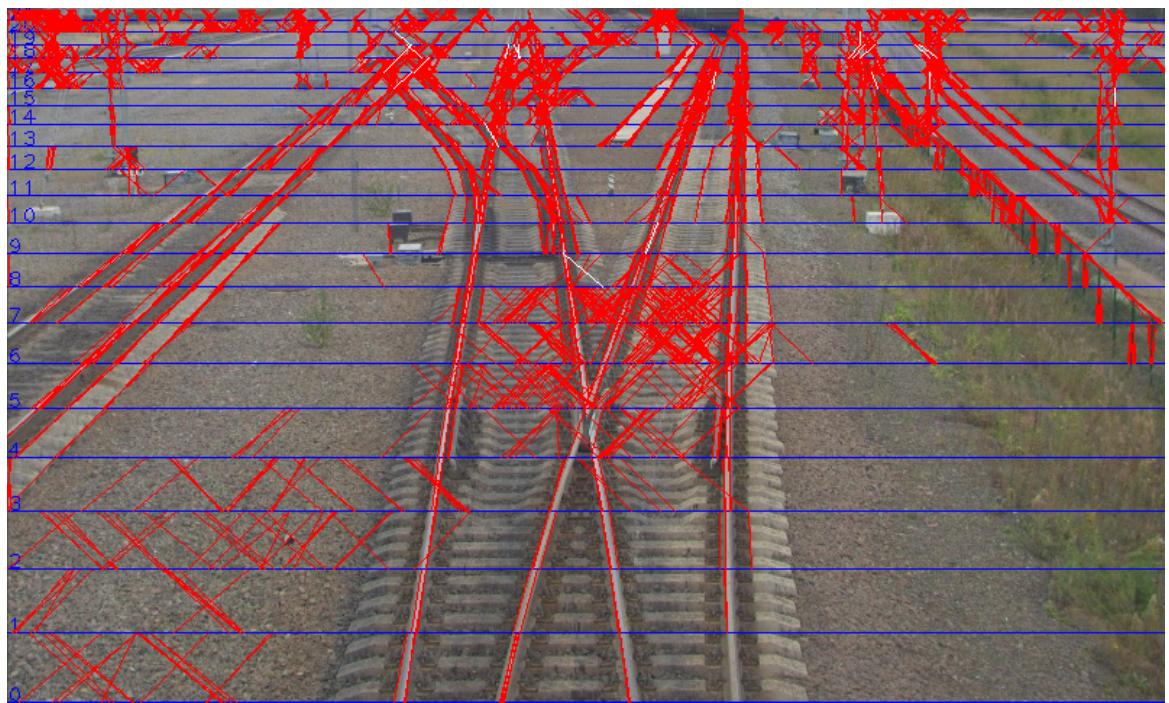


Рис.2.6

Можно заметить много прямых образованных на шпалах(между рельсами). Это связано с тем, что на шпалах находится много граничных точек<sup>2.4</sup>. Чтобы минимизировать влияние таких прямых произведем простое удаление горизонтальных рёбер: из изображения, полученного с помощью преобразования Canny удалим такие горизонтальные участки, на которых ребро встречается 3 и более пикселей подряд. В результате получим такую картинку ребер на изображении:

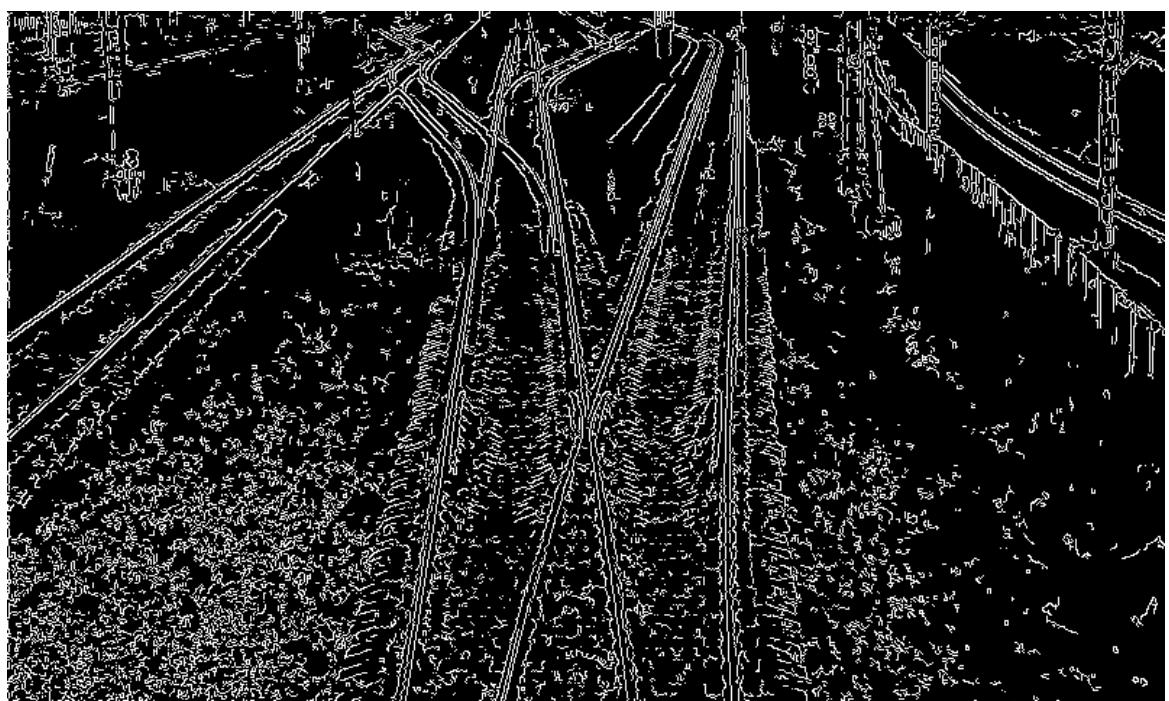


Рис.2.7

Также видно, что от одного рельса может появляться сразу несколько прямых. Чтобы этого избежать применим аппроксимацию близких прямых одной средней прямой(координаты такой прямой будут считаться, как центр масс всех близких прямых). Прямые будут считаться близкими, если расстояние между верхними и нижними точками соответствующих прямых меньше заданного:

$$\text{LinesEps} = \text{minLinesEps} + \frac{\text{maxLinesEps} - \text{minLinesEps}}{\text{imageHeight} - \text{maxLinesEps}} y,$$

где  $\text{minLinesEps} = 10$  и  $\text{maxLinesEps} = 20$  - значения полученные с учетом подсчета средней ширины рельса на изображении в зависимости от координаты Y.

Тогда после описанных выше преобразований получим следующий результат:



Рис.2.8

### 2.1.6. Поиск стрелок

**Построение графа прямых** Построим граф, вершинами которого будут найденные прямые. Каждая вершина будет иметь структуру:

$(p1, p2, \text{high}_neighs, \text{low}_neighs)$ , где  $p1$  - нижняя точка прямой,  $p2$  - верхняя точка прямой,  $\text{high}_neighs$  - верхние соседи(прямые, которые выходят из прямой, соответствующей текущей вершине),  $\text{low}_neighs$  - нижние соседи(прямые, которые входят в прямую, соответствующую текущей вершине).

Ребра в графе будут строиться по следующему принципу: начиная с нижнего горизонтального блока, для каждой прямой(*curLine*) ищем верхних соседей в следующем блоке. Прямая(*nextLine*) будет считать верхним соседом, если:

$|nextLine.p1.x - curLine.p2.x| < LinesEps$ , где LinesEps - считается также как в параграфе 2.1.5.

Найденные вершины добавляются, как верхние соседи, в текущую вершину. А также текущая вершина добавляется, как нижний сосед, в каждого из верхних соседей.

Например, на рисунке 2.9:

- A. прямые 1, 2 являются верхними соседями для прямой 4. А прямая 4 является нижним соседом для прямых 1 и 2.
- B. Прямая 3 лежит дальше, чем blockEps от прямой 4, поэтому между ними нет связи в графе.
- C. прямая 4 является верхним соседом для прямых 5 и 6, но не является верхним соседом для прямой 7. А прямые 5 и 6 в свою очередь являются нижними соседями для прямой 4.

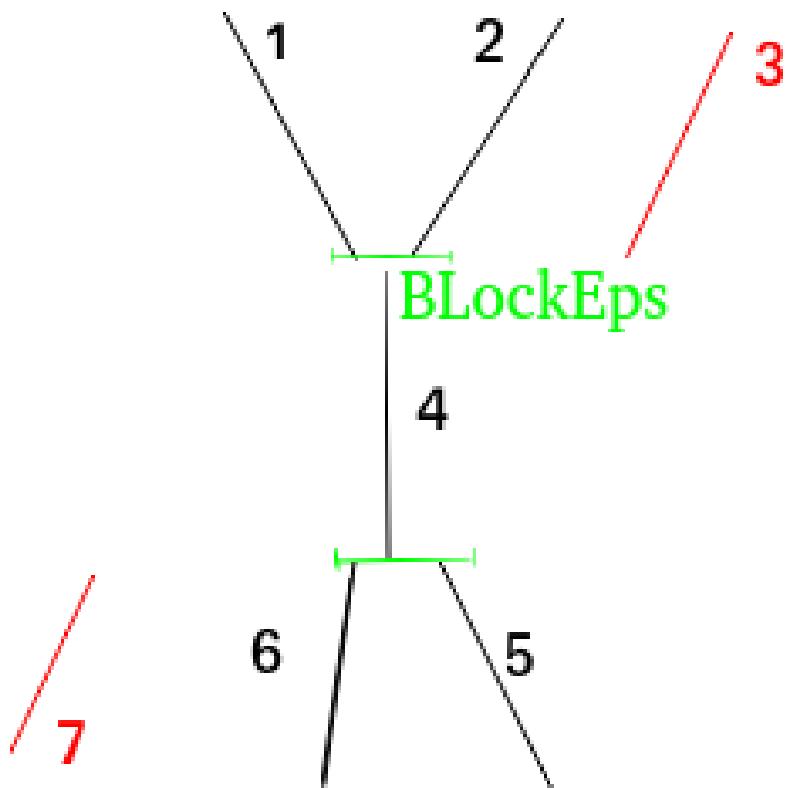


Рис.2.9

**Поиск стрелок в графе** В этом параграфе будет рассмотрен алгоритм поиска стрелок в построенном на предыдущем шаге графе.

Алгоритм поиска пересекающихся вершин в графе:

В цикле для каждой вершины графа производим следующие действия:

- A. Если вершина не имеет параллельного нижнего соседа - переходим к следующей вершине
- B. Для каждой пары верхних соседей текущей вершины проверяем, пересекаются ли они(2.1.6), если да - верхнюю точку текущей вершины добавляем в итоговый результат найденных стрелок. Таким образом, будут найдены Y и X пересечения.

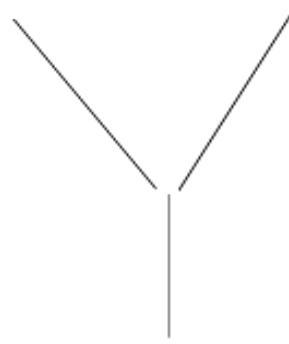


Рис.2.10

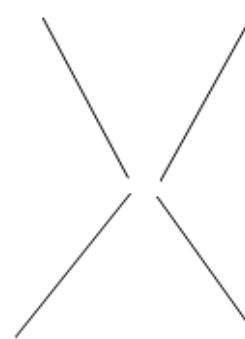


Рис.2.11

- C. Пункт 2) повторяется для нижних соседей текущей вершины. Таким образом, будут найдены Y - обратное и X образные пересечения.

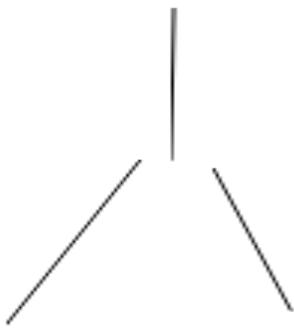


Рис.2.12

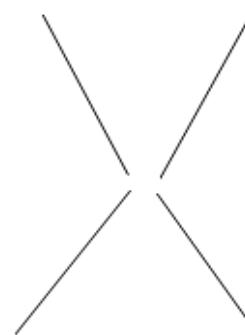


Рис.2.13

**Алгоритм проверки параллельности прямых** Рассмотрим функцию проверки параллельности прямых:

A. Вычисляем косинусы между прямыми и осью абсцисс.

```
0x = (1, 0)
cos1 = cos(line1, 0x)
cos2 = cos(line2, 0x)
```

B. Вычисляем разность косинусов

```
cos_diff = abs(cos1, cos2)
```

C. Вычисляем насколько близко должны быть прямые, чтобы считать их параллельными. Тут в очередной раз применяется линейная интерполяция. Так как чем ближе рельсы к камере, тем меньше угол должен быть между ними, чтобы посчитать их параллельными. Параметры max\_parallel\_cos и min\_parallel\_cos были получены экспериментально, рассматривая результаты работы алгоритма для крайних случаев реальных параллельных прямых на разных уровнях высоты.

```
max_parallel_cos = 0.4
min_parallel_cos = 0.05
parallel_cos_eps = max_parallel_cos - (
    max_parallel_cos -
    min_parallel_cos) / image_height * current_y
```

D. Если разность косинусов, вычисленная ранее, меньше parallel\_cos\_eps, прямые считаются параллельными.

```
return cos_diff < parallel_cos_eps
```

**Алгоритм проверки пересечения вершин** На вход поступают v1, v2 -вершины, для которых нужно определить пересекаются ли они. А также параметры: глубина проверки пересечений(intersection\_depth) и глубина проверки соседей(check\_neighs\_depth).

Алгоритм:

A. Если глубина проверки пересечений достигла 0 - возвращаем True - вершины пересекаются.

```
if intersection_depth == 0:
    return True
```

B. Если прямые, соответствующие вершинам НЕ параллельны - выбираем у каждой прямой параллельного верхнего соседа. Если параллельные соседи

существуют, рекурсивно вызываем функцию `is_intersection` с уменьшенной на единицу глубиной поиска пересечений.

```
5   if not is_parallel(v1, v2):
        parallel_1 = v1.getParallelNeigh()
        parallel_2 = v2.getParallelNeigh()
        if parallel_1 and parallel_2:
            return is_intersection(parallel_1, parallel_2,
                                   intersection_depth - 1, check_neighs_depth)
```

- C. Если глубина проверки соседей ещё не достигла нуля И у текущих вершин  $v_1$ ,  $v_2$  существуют параллельные им верхние соседи, то рекурсивно вызываем `is_intersection` для найденных параллельных верхних соседей с уменьшенной на единицу глубиной проверки соседей.

```
5   if check_neighs_depth > 0:
        parallel_1 = v1.getParallelNeigh()
        parallel_2 = v2.getParallelNeigh()
        if parallel_1 and parallel_2:
            return is_intersection(parallel_1, parallel_2,
                                   intersection_depth, check_neighs_depth - 1)
```

- D. Если не было выполнено ни одно из первых трех условий - вершины не являются пересекающимися - вернем `False`.

Описанный выше алгоритм применяется как для проверки пересечения между верхними соседями, так и между нижними соседями.

Рассмотрим алгоритм на примере(синими линиями изображены границы горизонтальных блоков):

В данном примере рассматривается прямая 1.

На вход функции `is_intersection` поступают соседние прямые 2 и 3, и параметры `intersection_depth = 2`, `check_neighs_depth = 1`.

Если прямые 2 и 3 НЕ параллельны, то запускаем `is_intersection` для прямых 4 и 5 и `intersection_depth = 1`, `check_neighs_depth = 1`.

Если прямые 4 и 5 НЕ параллельны, то запускаем `is_intersection` для прямых 6 и 7 и `intersection_depth = 0`, `check_neighs_depth = 1`. Теперь функция возвращает `True` и верхняя точка прямой 1 добавляется, как стрелка в результат.

В случае пересечений между нижними соседями пример будет зеркально отражен по оси Y.

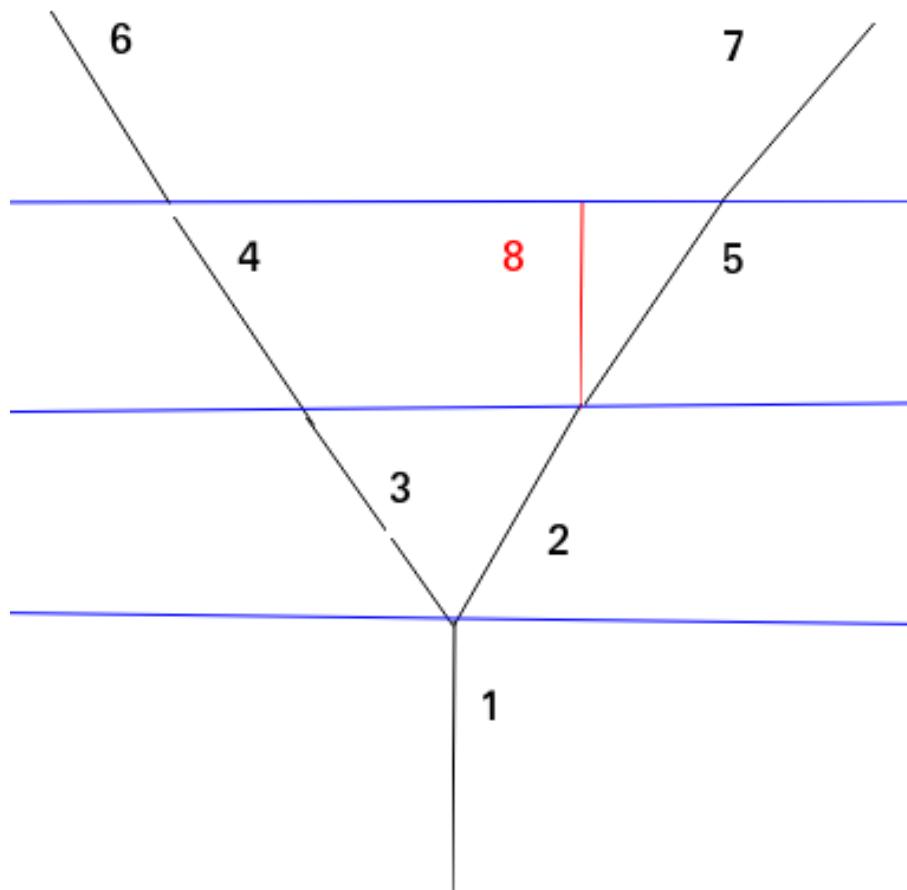


Рис.2.14

Параметр `check_neighs_depth` нужен в случае, когда сами соседи являются параллельными(прямые 2 и 3), но их продолжения(прямые 4 и 5) уже НЕ являются параллельными. Такая ситуация характерна для Y пересечений, которые находятся близко к камере.

**Аппроксимация близких вершин** На последнем этапе близкие по расстоянию стрелки аппроксимируются их центрами масс. Эта процедура повторяется рекурсивно, пока в результирующем массиве все стрелки не станут полностью отделимыми друг от друга.

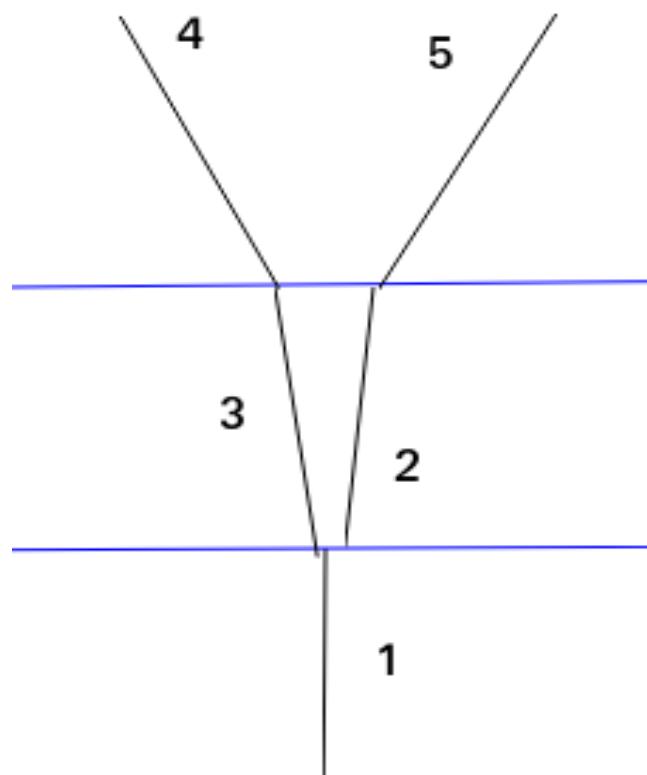


Рис.2.15

### ***2.1.7. Описание данных для тестирования***

Для тестирования алгоритма было размечено 204 фотографии ж/д путей, сделанных с локомотива. Часть фотографий была сделана при дневном свете 2.16, а часть при искусственном 2.18. Также имеются изображения, сделанные в летний 2.16 и в зимний период 2.19.

#### **Параметры датасета**

- A. Всего стрелок размечено: 1308
- B. Среднее количество стрелок на одном изображении: 6.3
- C. Около 70% изображений сделаны в летний период. Около 30% в зимний.

### 2.1.8. Результаты

**Примеры работы алгоритма** Рассмотрим результаты работы алгоритма на различных изображениях.

Результаты работы на изображении сделанном при дневном:



Рис.2.16



Рис.2.17

Истинное положение стрелок (маркеры) на рисунках 2.16, 2.17, 2.18, 2.19 - не обозначены, так как их положения очевидны

Видно, что есть большие стрелки - те, которые находятся достаточно близко к камере, алгоритм распознает очень хорошо. С маленькими стрелками есть небольшие проблемы, например на изображении 2.17 самая дальняя маленькая стрелка слева не распознавалась.

Также видно, что достаточно много false positive стрелок получившихся от столбов и заборов. Один из способов их подавления - использовать алгоритм Region growing up([5]) для сегментации изображения снизу вверх.

Теперь рассмотрим результат работы алгоритма при других погодных условиях и искусственном свете. Видно, что алгоритм хорошо справляется и с такими изображениями.

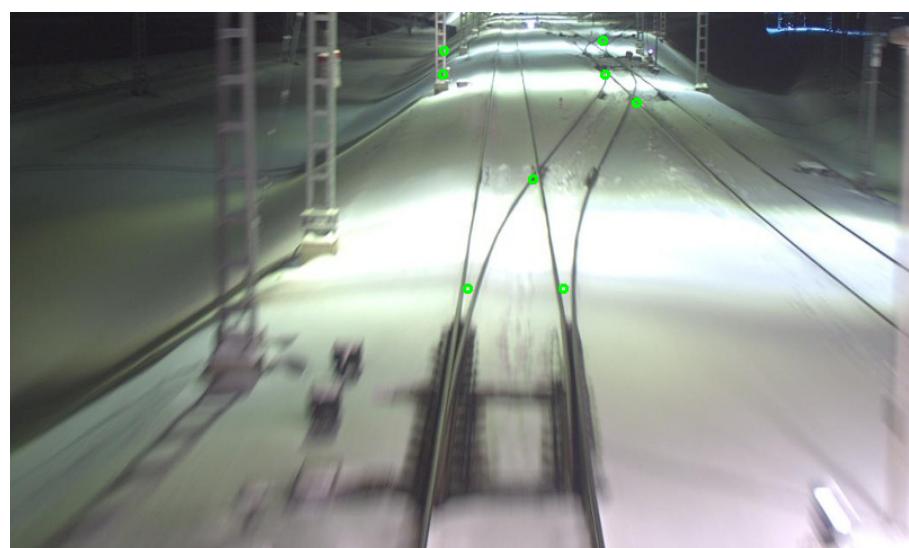


Рис.2.18

Теперь посмотрим на пример изображения, на котором алгоритм выдал очень плохой результат: Видно, что на изображении есть ярко освещенные солнцем



Рис.2.19

участки, и темная тень от поезда, в связи с этим алгоритм Отцу [3] выдал достаточно высокое значение пороговых фильтров и рельсы, находящиеся в тени не были восприняты как рёбра алгоритмом Canny[2]. В следствии чего соответствующие рельсам линии не были найдены алгоритмом Хафа [1].

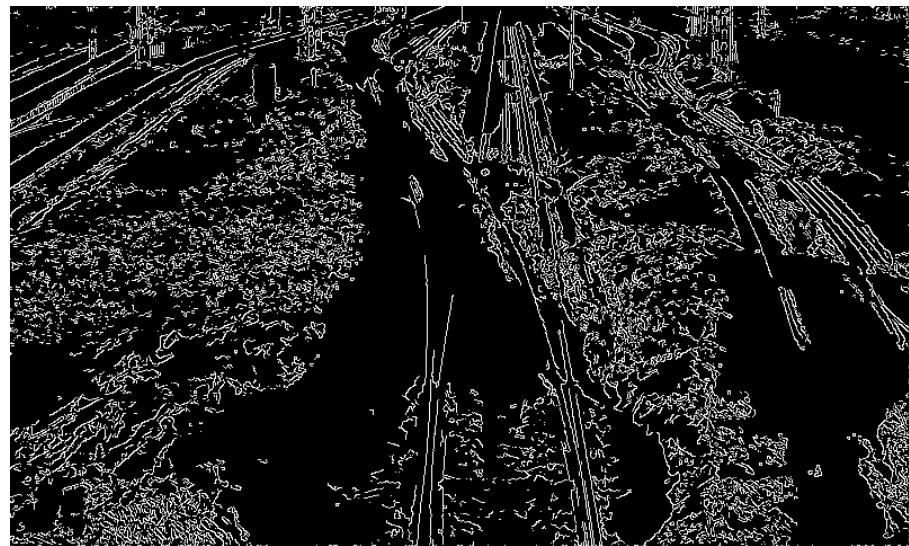


Рис.2.20

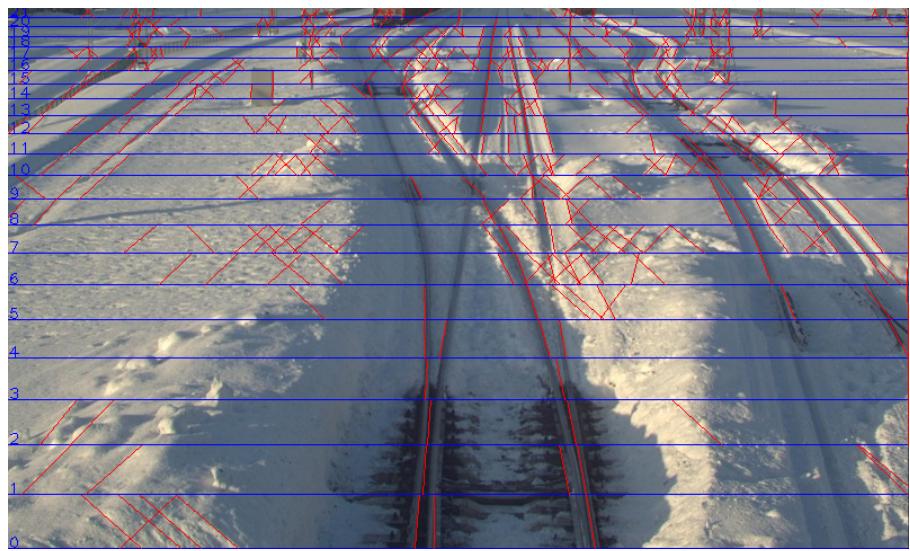


Рис.2.21

**Статистические результаты** Вспомним о том, что мы не знаем параметры камеры и поэтому ширину горизонтальных блоков выбираем с помощью линейной интерполяции 2.1.6. Интерполяция происходит между значениями `maxBlockSize` и `minBlockSize`. Рассмотрим зависимость результатов обнаружения от выбора этих параметров:

<code>minBlockEps</code>	<code>maxBlockEps</code>	<code>time(s)</code>	<code>Precision</code>	<code>Recall</code>
7	45	0.8	0.27	0.92
7	30	5.5	0.22	0.97
20	45	0.08	0.4	0.44

$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$  - процент корректных предсказаний.

$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$  - мера, определяющая как хорошо алгоритм находит позитивные примеры(стрелки)

Видно, что при уменьшении максимального размера блока растёт время работы алгоритма. Уменьшается процент корректных предсказаний. И увеличивается количество найденных положительных примеров.

Также и в обратную сторону, если увеличивать минимальный размер блока, то время работы уменьшается, НО количество найденных положительных примеров и процент корректных предсказаний уменьшаются.

При большой нижней границе  $\text{minBlockEps}$  далекие стрелки вообще не находятся, зато хорошо ищутся близкие стрелки, вот пример для  $\text{minBlockEps} = 20\text{pix}$ :

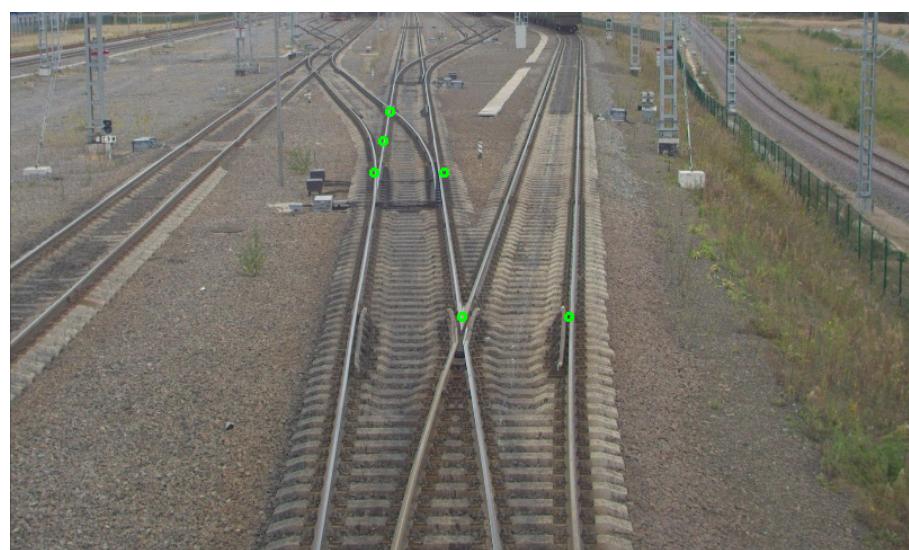


Рис.2.22

## ЛИТЕРАТУРА

- [1] **Преобразование Хафа.** Википедия. [Электронный ресурс] URL - [https://en.wikipedia.org/wiki/Hough\\_transform](https://en.wikipedia.org/wiki/Hough_transform)
- [2] **Алгоритм Canny.** [Электронный ресурс] URL - <https://habr.com/ru/post/114589/>
- [3] **Алгоритм Отцу.** [Электронный ресурс] URL - <https://docs.opencv.org/2.4/doc/tutorials/imgproc/threshold/threshold.html>
- [4] **Перспективная проекция.** [Электронный ресурс] URL - <http://stratum.ac.ru/education/textbooks/kgrafic/lection04.html>
- [5] **Efficient railway tracks detection and turnouts recognition method using HOG features.** [Электронный ресурс] URL - <https://link.springer.com/article/10.1007/s00521-012-0846-0>
- [6] **Vision based rail track and switch recognition for self-localization of trains in a rail network.** [Электронный ресурс] URL - <https://ieeexplore.ieee.org/document/5940466?denied=>
- [7] **Histogram of oriented gradients.** Википедия. [Электронный ресурс] URL - [https://en.wikipedia.org/wiki/Histogram\\_of\\_oriented\\_gradients](https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients)
- [8] **Scale-invariant feature transform.** [Электронный ресурс] URL - [https://en.wikipedia.org/wiki/Scale-invariant\\_feature\\_transform](https://en.wikipedia.org/wiki/Scale-invariant_feature_transform)
- [9] **Using Partial Edge Contour Matches for Efficient Object Category Localization**, Hayko Riemenschneider, Michael Donoser and Horst Bischof Proceedings of European Conference on Computer Vision. [Электронный ресурс] URL - <http://www.icg.tugraz.at/Members/hayko/partial-contour-efficient-matching>
- [10] **Speeded up robust features.** [Электронный ресурс] URL - [https://en.wikipedia.org/wiki/Speeded\\_up\\_robust\\_features](https://en.wikipedia.org/wiki/Speeded_up_robust_features)
- [11] **Support Vector Machine.** [Электронный ресурс] URL - [https://en.wikipedia.org/wiki/Support\\_vector\\_machine](https://en.wikipedia.org/wiki/Support_vector_machine)

- [12] **Decision tree.** [Электронный ресурс] URL -  
[https://en.wikipedia.org/wiki/Decision\\_tree](https://en.wikipedia.org/wiki/Decision_tree)
- [13] **K-nearest neighbors algorithm.** [Электронный ресурс] URL -  
[https://en.wikipedia.org/wiki/K-nearest\\_neighbors\\_algorithm](https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm)
- [14] **Sliding window.** [Электронный ресурс] URL -  
<https://www.pyimagesearch.com/2015/03/23/sliding-windows-for-object-detection-with-p>