

Санкт-Петербургский политехнический университет Петра Великого  
Институт прикладной математики и механики  
Высшая школа прикладной математики и вычислительной физики

---

**ОТЧЁТ**  
**по преддипломной практике**

**Подготовка выпускной квалификационной работы бакалавра  
на тему: *Поиска пересечений линий на изображении***

по направлению 01.03.02 «Прикладная математика и информатика»  
(профиль Системное программирование)

Студент гр. 360102/60201 ..... Туников Д.А.

Оценка научного руководителя ВКР ..... Шубников В.Г.

Санкт-Петербург  
2020

# Содержание

<b>1. Введение</b>	<b>3</b>
1.1. Актуальность задачи . . . . .	3
1.2. Обзор литературы . . . . .	3
1.2.1. Поиск объекта по его геометрическим свойствам . .	4
1.3. Постановка задачи . . . . .	11
<b>2. Основная часть</b>	<b>12</b>
2.1. Метод, основанный на преобразовании Хафа . . . . .	12
2.1.1. Преобразование Хафа . . . . .	12
2.1.2. Алгоритм метода . . . . .	13
2.1.3. Поиск границ на изображении . . . . .	13
2.1.4. Разбиение изображения на горизонтальные блоки .	16
2.1.5. Поиск рельсов . . . . .	17
2.1.6. Поиск стрелок . . . . .	17
2.1.7. Описание данных для тестирования . . . . .	24
2.1.8. Результаты . . . . .	25
<b>Список литературы</b>	<b>30</b>

# 1. Введение

## 1.1. Актуальность задачи

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

- 1) Автоматическое определение полей во время спортивных трансляций
- 2) Детектирование ориентиров при автоматическом управлении роботом
- 3) Детектирование железнодорожных стрелок на изображении, сделанном с локомотива поезда

В данной статье 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.

## 1.2. Обзор литературы

Существует несколько глобальных подходов к поиску объектов на изображении:

- 1) Поиск объекта по его геометрическим свойствам. В случае ж/д стрелок это может быть поиск пересекающихся линий.
- 2) Построение классификатора, обученного на размеченных изображениях.

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:

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

Пример применения алгоритма Canny к входному изображению:

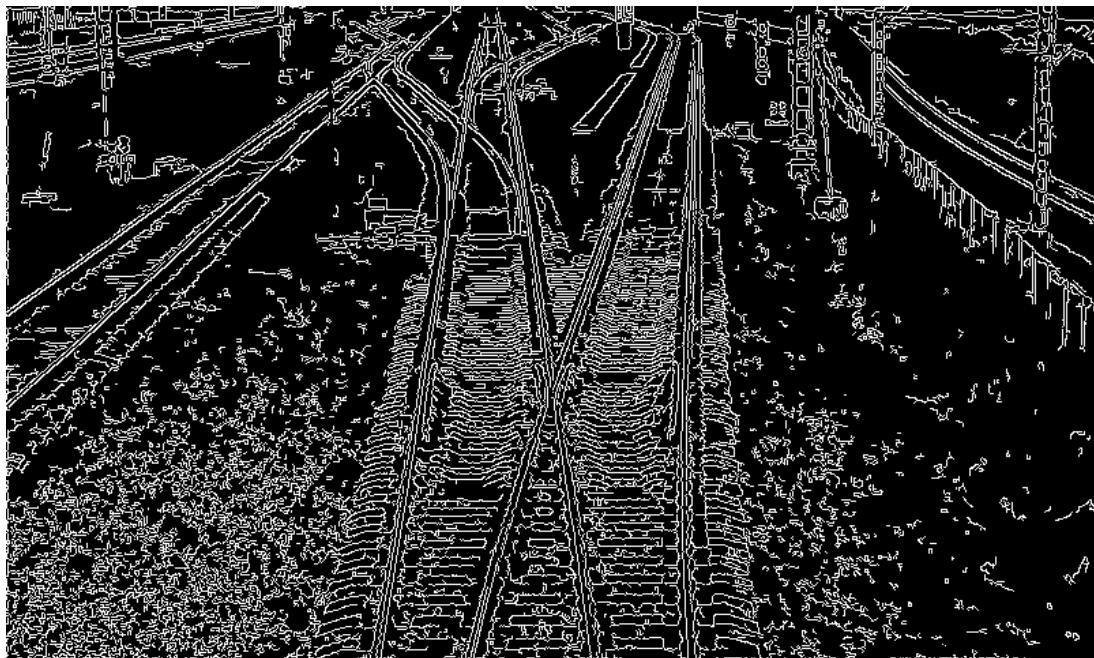


Рис. 1.

- 2) 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):

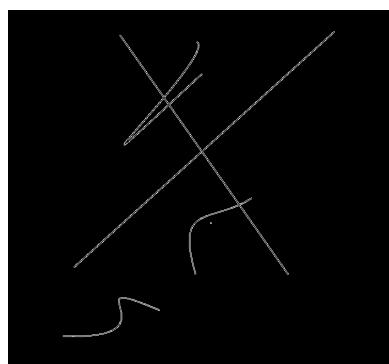


Рис. 2.

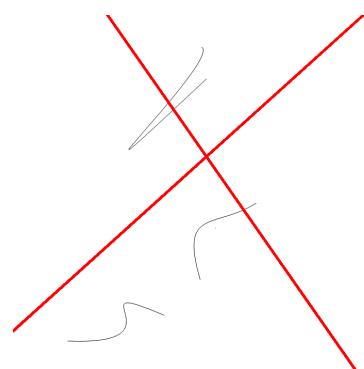


Рис. 3.

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

- 1) Поиск границ [2]
- 2) Поиск линий 2.1.1
- 3) 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**

**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. 4.

2. К выделенному участку применяется алгоритм поиска ребер:
  - 1) Рассчитывается вектор градиента в каждой точке изображения

- 2) Далее рассматриваются окрестности по 3х3 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 3х3.

$$S = \begin{pmatrix} Cov_{xx} & Cov_{xy} \\ Cov_{yx} & Cov_{yy} \end{pmatrix}, Cov_{ab} = \frac{1}{n} \sum_1^N (g_a, g_b)$$

- 3) Вычисляются собственные числа этой матрицы, и если выполнены неравенства:  $\begin{cases} \lambda_1 \geq t_l \\ \frac{\lambda_2}{\lambda_1} \geq m_l \end{cases}$ , где  $t_l, m_l$  вычисленные эмперическим 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. 5.

### 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 ОУ. 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.

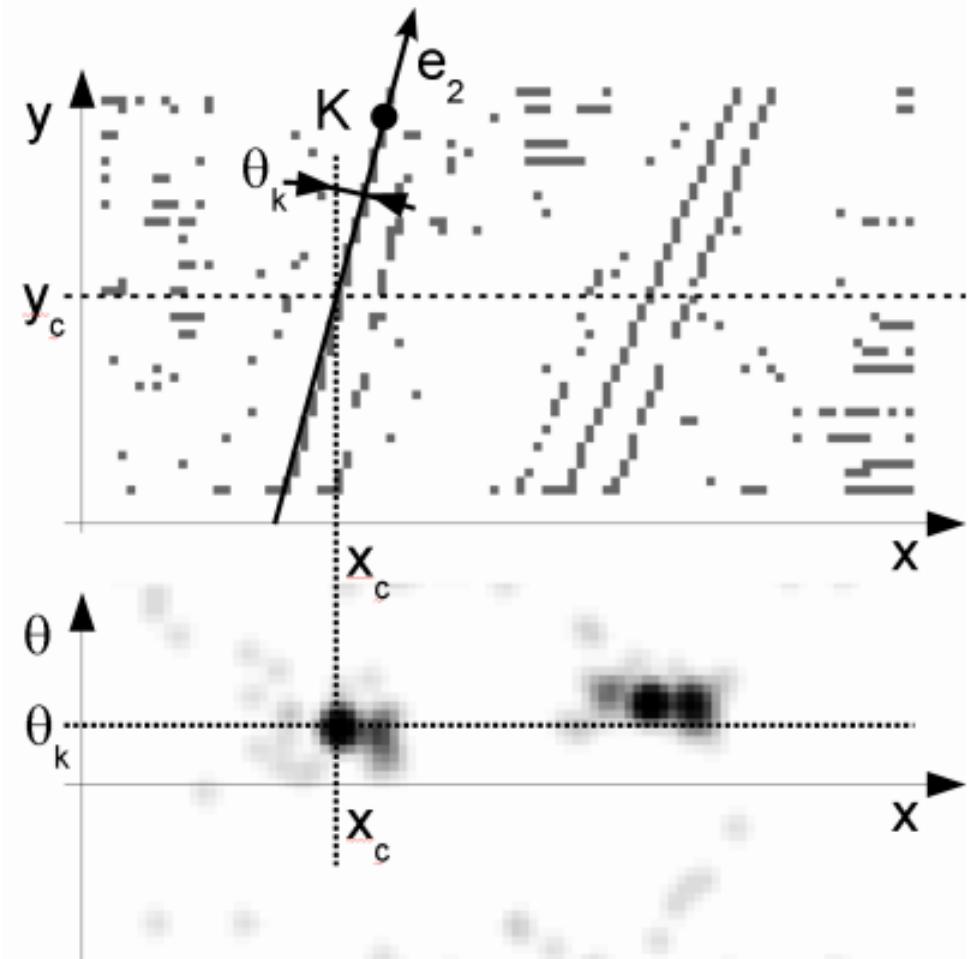


Рис. 6.

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. 7.

### 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 наблюдаемых нами колея всегда имеет боковое отклонение от центра поезда принадлежащее интервалу  $[-s_1, s_1]$ . Если центр колеи становится  $< -s_1$ , то это означается, 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).

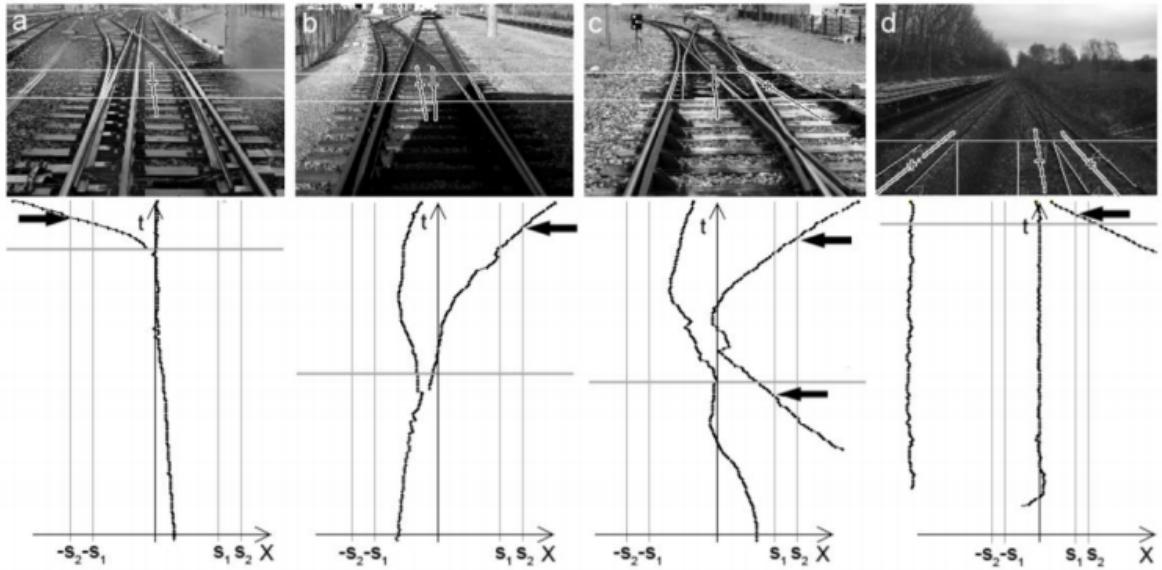


Рис. 8.

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 и их пересечений 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. 9.

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).

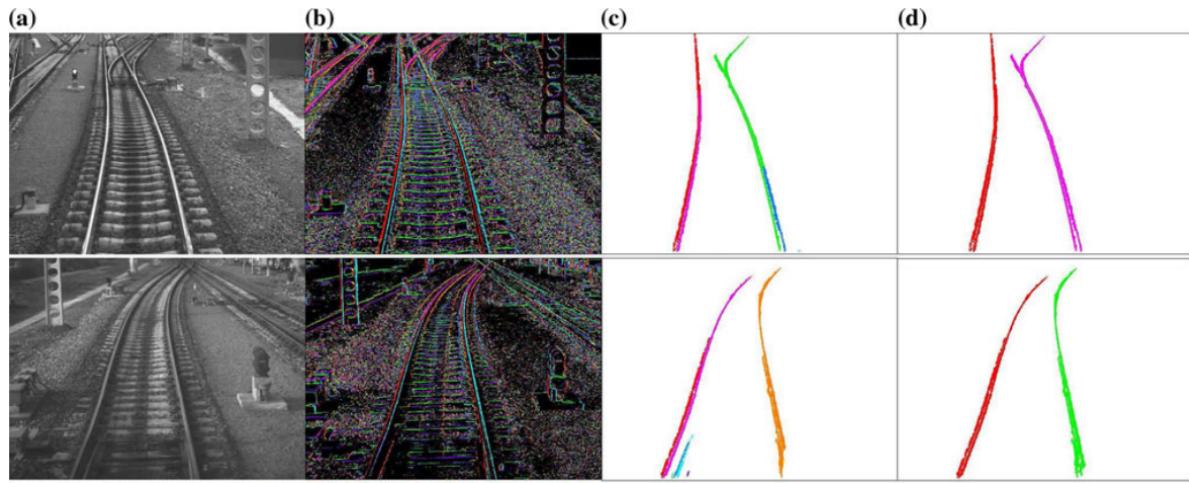


Рис. 10.

### 1.3. Постановка задачи

На вход подается изображение — фотография сделанная с головы поезда. Необходимо найти на входном изображении места пересечений железнодорожных 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. 11. Пример входного изображения

Замечания:

- 1) Точность поиска стрелок - окрестность 30 на 30 пикселей
- 2) 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).

## 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  $\theta \in [0, \pi]$ . 12

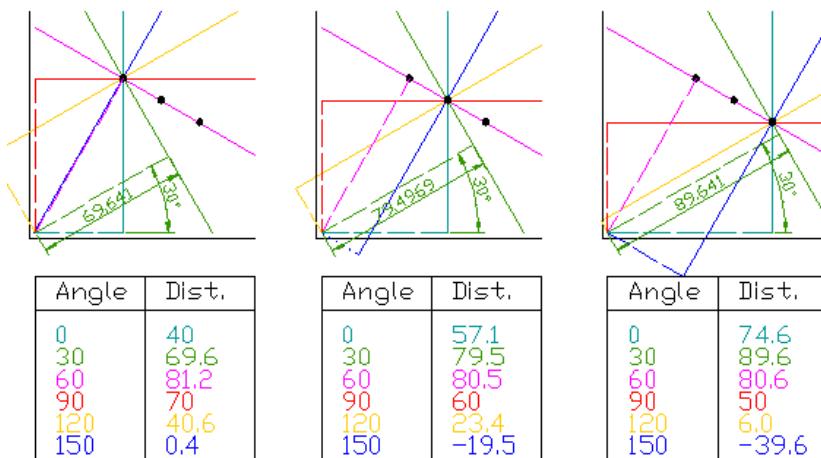


Рис. 12.

Каждая прямая голосует за свои параметры  $(\rho, \theta)$ , то есть значение значения аккумулятора  $A[\rho, \theta]$  увеличивается на единицу. После голо-

сования 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:

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

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

- 2) 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:
- 3) Подавление не-максимумов. Среди всех максимумов оставляем только те, что являются локальными максимумами в окрестности.
- 4) 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 14, здесь только зелёные точки будут выбраны граничными. В нашем случае значение верхнего порога вычисляется с помощью бинаризации 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.

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

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) (Ay,x · MGxy,x)
                           |     GY ← ∑(y=0..2) ∑(x=0..2) (Ay,x · MGyy,x)
                           |     G ← √(GX2 + GY2)
                           |     θ ← round(atan2(GX, GY) / (π/4)) · π/4 - π/2 if G ≠ 0
                           |     θ ← ErrCode otherwise
                           |     SobMtrixiY, iX ← G
                           |     SobMtrixiY, iX+1+cols(Matrix) ← θ
                           |
                           | return SobMtrix
```

Рис. 13.

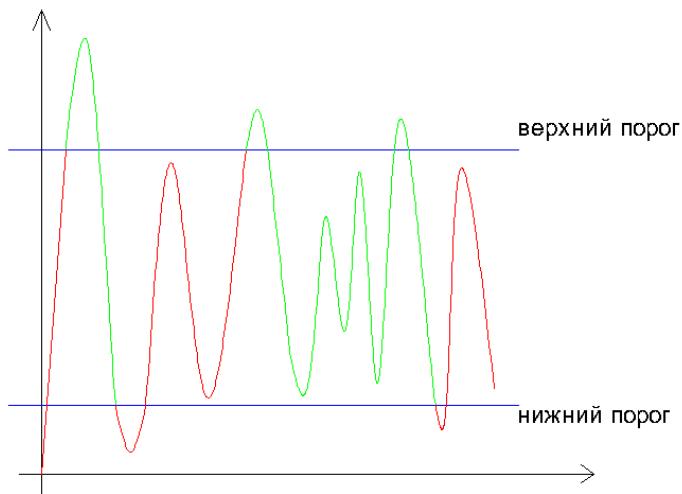


Рис. 14.

- 5) 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:

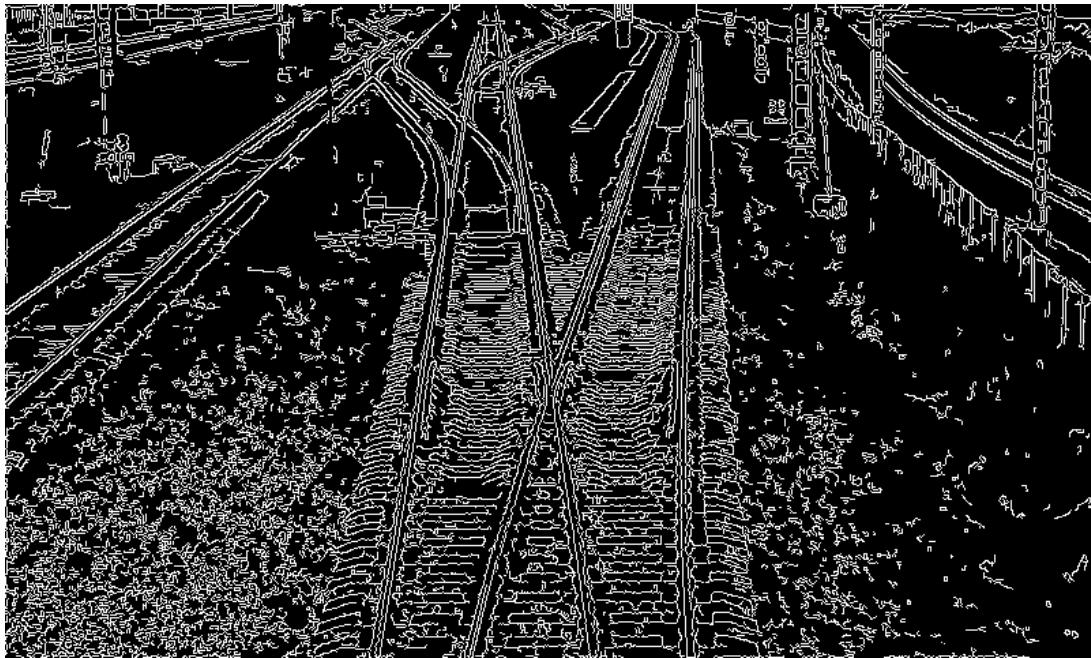


Рис. 15.

#### 2.1.4. Разбиение изображения на горизонтальные блоки

Сначала разбиваем изображение на горизонтальные блоки 16. 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:

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

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. 16.

## 2.1.5. Поиск рельсов

Теперь рассмотрим процесс поиска рельсов на изображении, полученном после применения алгоритма Canny15.

После разбиения 16 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):

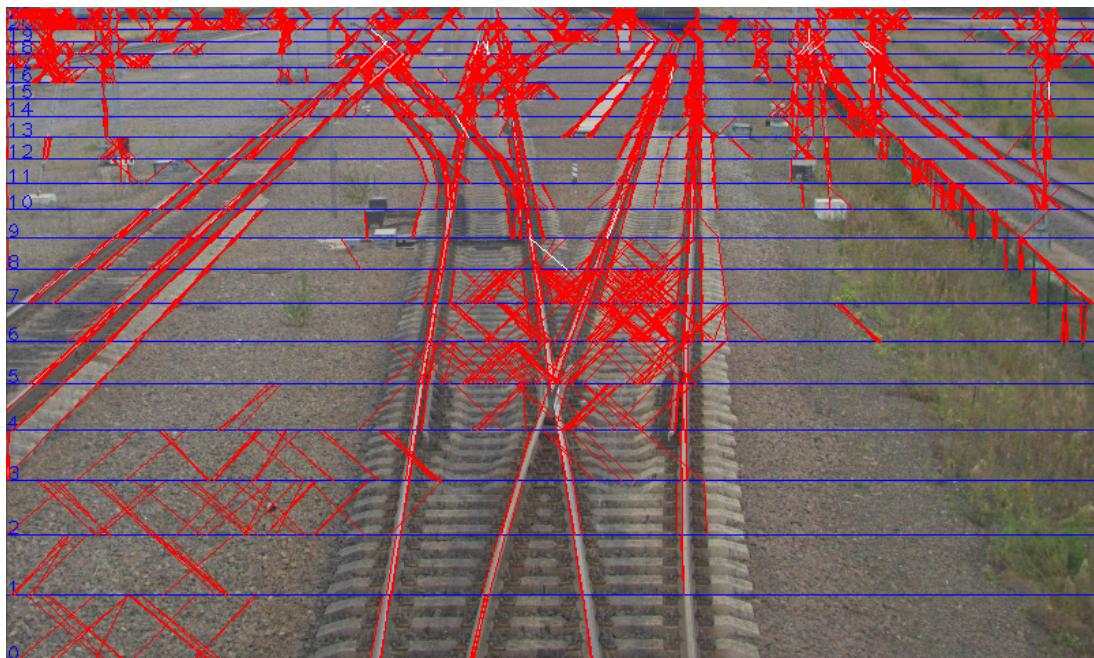


Рис. 17.

## 2.1.6. Поиск стрелок

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

$(p1, p2, high_{neighs}, low_{neighs})$ , где  $p1$  - нижняя точка прямой,  $p2$  - верхняя точка прямой,  $high_{neighs}$  - верхние соседи(прямые, которые выходят из прямой, соответствующей текущей вершине),  $low_{neighs}$  - 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:

$$|nextLine.p1.x - curLine.p2.x| < blockEps, \text{ где} \\ blockEps = minNeighsEps + \frac{maxNeighsEps - minNeighsEps}{imageHeight - maxNeighsEps}y,$$

где  $minNeighsEps = 3$ ,  $maxNeighsEps = 10$ . То есть  $blockEps$  линейно увеличивается в зависимости от координаты Y.

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 18:

- 1) прямые 1, 2 являются верхними соседями для прямой 4. А прямая 4 является нижним соседом для прямых 1 и 2.
- 2) Прямая 3 лежит дальше, чем  $blockEps$  от прямой 4, поэтому между ними нет связи в графе.
- 3) 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.

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

Алгоритм поиска пересекающихся вершин в графике:

В цикле для каждой вершины графа производим следующие действия:

- 1) Если вершина не имеет параллельного нижнего соседа - переходим к следующей вершине
- 2) 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.
- 3) Пункт 2) повторяется для нижних соседей текущей вершины. Таким образом, будут найдены Y - обратное и X образные пересечения.

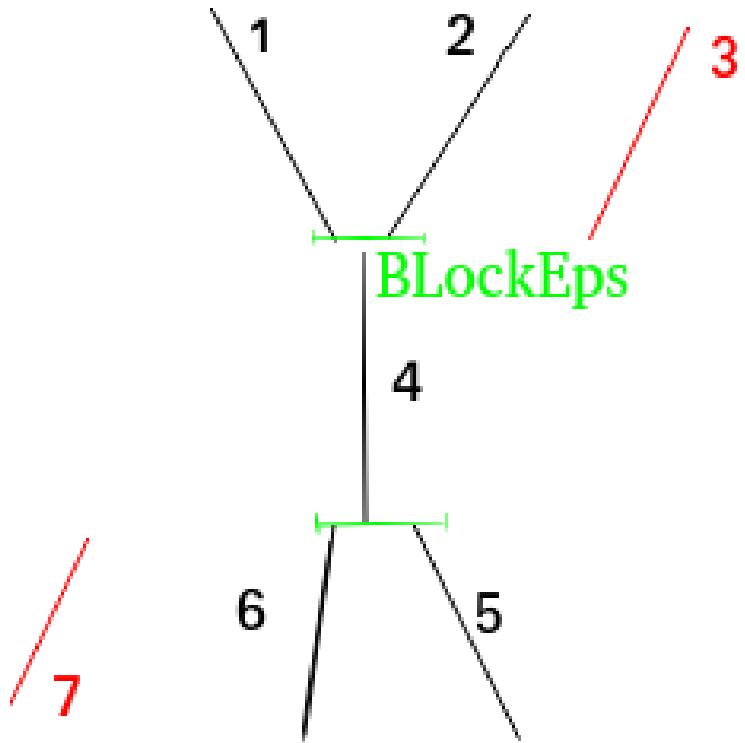


Рис. 18.

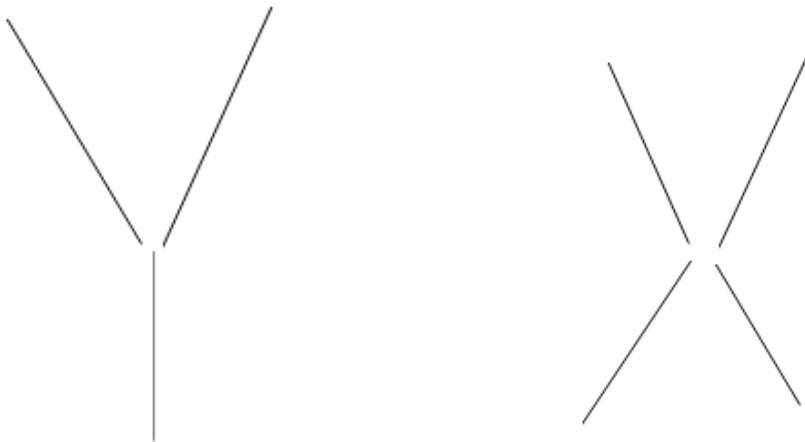


Рис. 20.

Рис. 19.

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

- 1) Вычисляем косинусы между прямыми и осью абсцисс.

$$Ox = (1, 0)$$

$$\cos 1 = \cos(\text{line1}, Ox)$$

$$\cos 2 = \cos(\text{line2}, Ox)$$

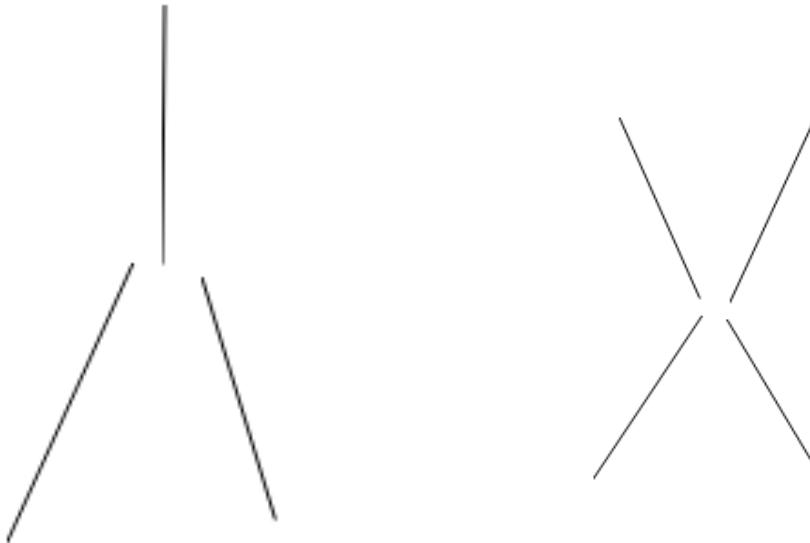


Рис. 22.

Рис. 21.

2) Вычисляем разность косинусов

$$\cos\_diff = \text{abs}(\cos1, \cos2)$$

3) 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.

$$\text{max_parallel_cos} = 0.4$$

$$\text{min_parallel_cos} = 0.05$$

$$\text{parallel_cos_eps} = \text{max_parallel_cos} - (\text{max_parallel_cos} - \text{min_parallel_cos}) / \text{image_height} * \text{current_y}$$

4) Если разность косинусов, вычисленная ранее, меньше `parallel_cos_eps`, прямые считаются параллельными.

$$\text{return } \cos\_diff < \text{parallel\_cos\_eps}$$

**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:

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

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

- 2) 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.

```
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)
```

- 3) Если глубина проверки соседей ещё не достигла нуля И у текущих вершин `v1`, `v2` существуют параллельные им верхние соседи, то рекурсивно вызываем `is_intersection` для найденных параллельных верхних соседей с уменьшенной на единицу глубиной проверки соседей.

```
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)
```

- 4) 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 граници 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.

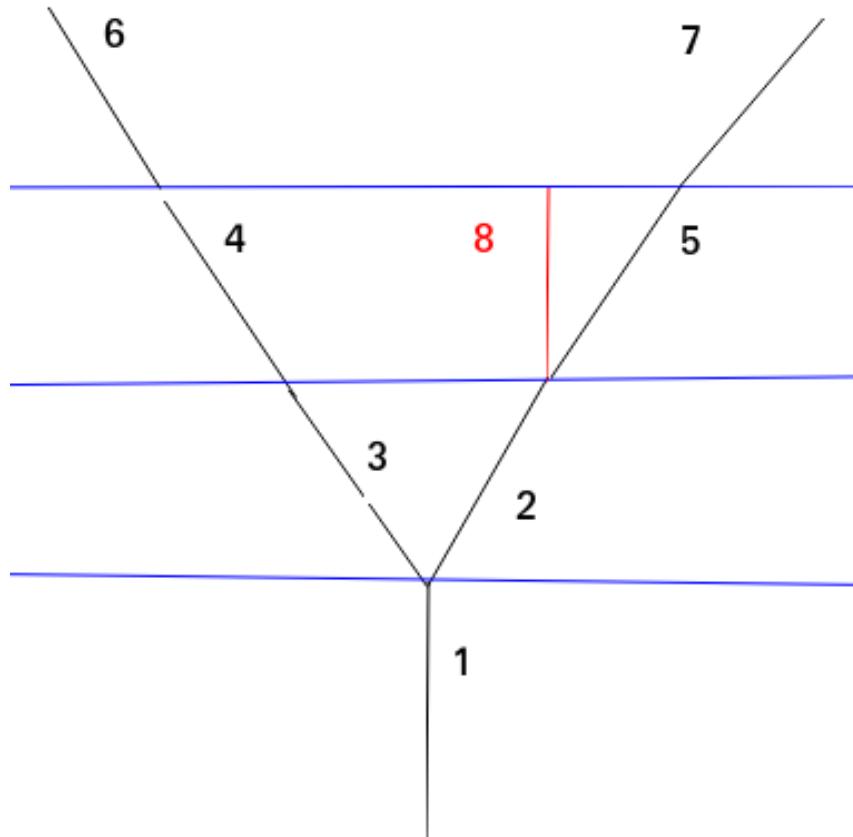


Рис. 23.

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

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.

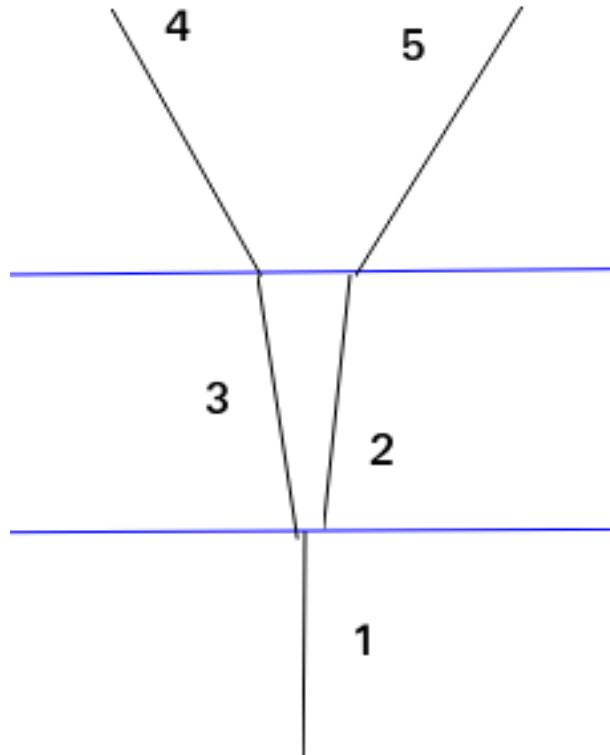


Рис. 24.

**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.

## **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 25, а часть при искусственном 27. Также имеются изображения, сделанные в летний 25 и в зимний период 28.

### **Параметры датасета**

- 1) Всего стрелок размечено: 1308
- 2) Среднее количество стрелок на одном изображении: 6.3
- 3) 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. 25.

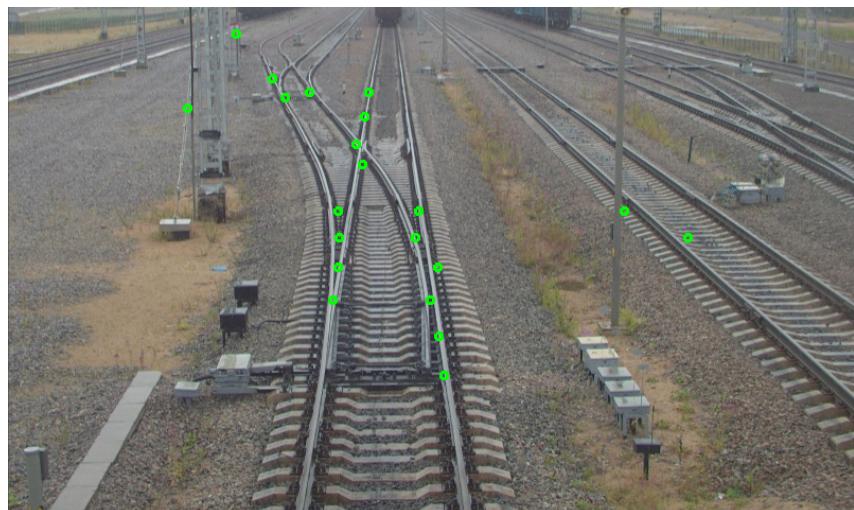


Рис. 26.

Истинное положение стрелок (маркеры) на рисунках 25, 26, 27, 28 - 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 26 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.

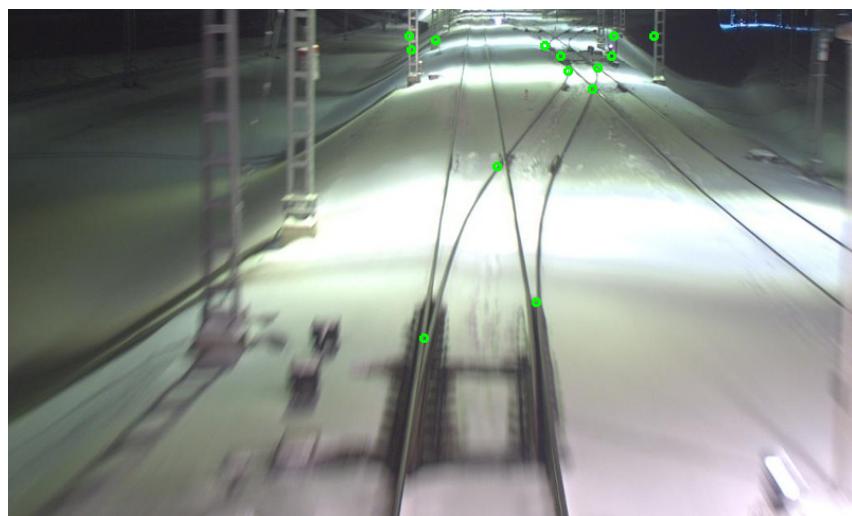


Рис. 27.

Теперь посмотрим на пример изображения, на котором алгоритм выдал очень плохой результат:

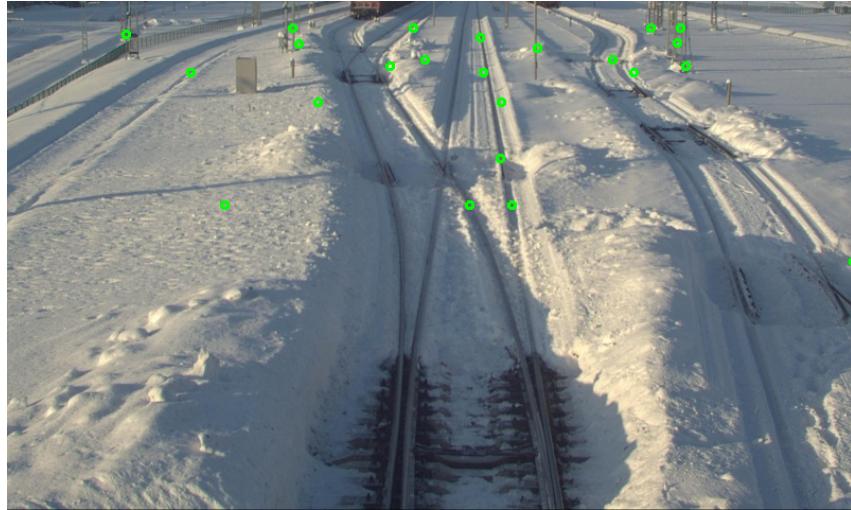


Рис. 28.

Видно, что на изображении есть ярко освещенные солнцем 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].

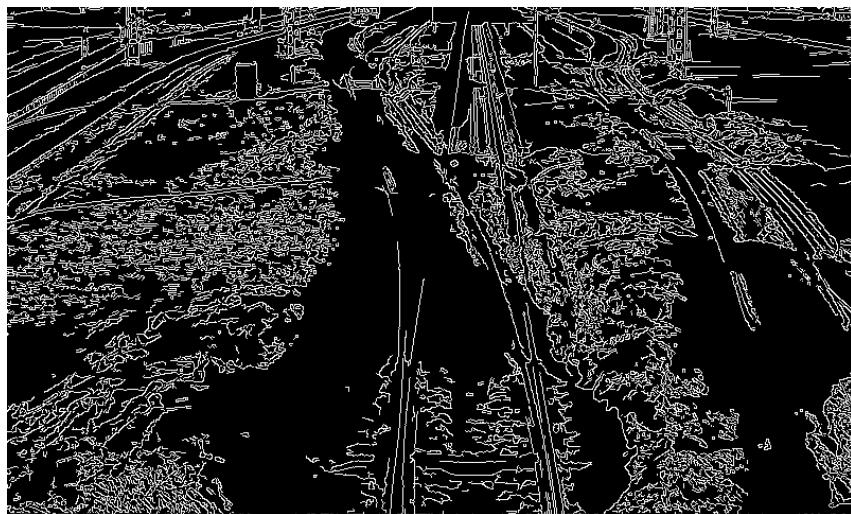


Рис. 29.

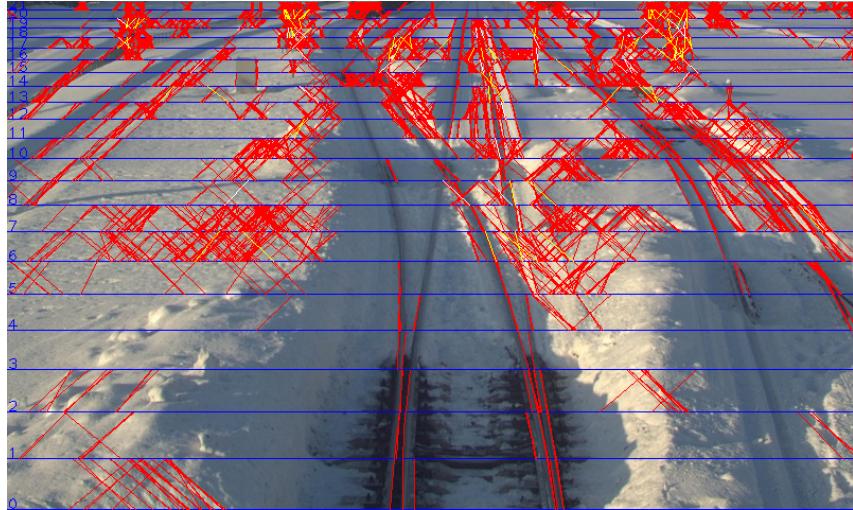


Рис. 30.

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



Рис. 31.

## Список литературы

- [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=1>
- [7] **Histogram of oriented gradients.** [Электронный ресурс] URL - [https://en.wikipedia.org/wiki/Histogram\\_of\\_oriented\\_gradients](https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients)