**Министерство науки и высшего образования РФ**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Уфимский университет науки и технологий»**

**Кафедра** Высокопроизводительных вычислительных технологий и систем

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 100 |  |  |  |  |  |  |  |  |  |  |
| 90 |  |  |  |  |  |  |  |  |  |  |
| 80 |  |  |  |  |  |  |  |  |  |  |
| 70 |  |  |  |  |  |  |  |  |  |  |
| 60 |  |  |  |  |  |  |  |  |  |  |
| 50 |  |  |  |  |  |  |  |  |  |  |
| 40 |  |  |  |  |  |  |  |  |  |  |
| 30 |  |  |  |  |  |  |  |  |  |  |
| 20 |  |  |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |  |
| 0 |  |  |  |  |  |  |  |  |  |  |

**МИНИМИЗАЦИЯ ПОГРЕШНОСТИ ВОССТАНОВЛЕНИЯ КОЭФФИЦИЕНТОВ МАТРИЦЫ ПРОЕКЦИИ НА ОСНОВЕ ДАННЫХ С ДОРОЖНЫХ КАМЕР**

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовой работе по дисциплине

«Методы оптимизации»

**3952. 337102.000 ПЗ**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Группа  ПМ-457 | Фамилия И.О. | Подпись | Дата | Оценка |
| Студент | Акмурзин М.Э. |  |  |  |
| Консультант | Касаткин А.А. |  |  |  |
| Принял | Лукащук В.О. |  |  |  |

Уфа 2024

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Уфимский университет науки и технологий»

Кафедра Высокопроизводительных вычислительных технологий и систем

**ЗАДАНИЕ**

на курсовую работу по дисциплине

**«Методы оптимизации»**

Студент: Акмурзин Михаил Эдуардович Группа: ПМ-457

Консультант: Касаткин Алексей Александрович

1. Тема курсовой работы

Минимизация погрешности восстановления коэффициентов матрицы проекции на основе данных с дорожных камер.

2. Основное содержание

1. Изучение и реализация модели камеры обскуры для проекции мировых координат на плоскость изображения
2. Разработка и реализация метода восстановления параметров матрицы проекции, основанного на геометрических ограничениях сцены
3. Оформить пояснительную записку к курсовой работе.

3. Требования к оформлению материалов работы

3.1. Требования к оформлению пояснительной записки

Пояснительная записка к курсовой работе должна быть оформлена в соответствии с требованиями ГОСТ и содержать

• титульный лист,

• задание на курсовую работу,

• содержание,

• введение,

• заключение,

• список литературы,

• приложение, содержащее листинг разработанной программы, если таковая имеется.

|  |  |
| --- | --- |
| Дата выдачи задания  "\_\_" \_\_\_\_\_\_\_\_ 202\_ г. | Дата окончания работы  "\_\_" \_\_\_\_\_\_\_\_\_\_ 202\_ г. |

Консультант \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Касаткин А.А.

**Содержание**

[Введение 4](#_Toc191624123)

[Теоретическая часть 5](#_Toc191624124)

[1. Модель камеры обскуры 5](#_Toc191624125)

[2. Ограничения сцены 7](#_Toc191624126)

[2.1. Положения камеры 7](#_Toc191624127)

[3.1. Точки схода 9](#_Toc191624128)

[3.2. Начальное решение 10](#_Toc191624129)

[4. Функция оптимизация 10](#_Toc191624130)

[Практическая часть 12](#_Toc191624131)

[5. Демонстрация на синтетических данных 12](#_Toc191624132)

[6. Демонстрация на реальных данных 14](#_Toc191624133)

[Заключение 17](#_Toc191624134)

[Список литературы 18](#_Toc191624135)

# ВВЕДЕНИЕ

В современных системах мониторинга дорожного движения широко применяются видеокамеры, позволяющие фиксировать транспортные потоки и анализировать их характеристики. Одной из ключевых задач обработки таких данных является восстановление параметров матрицы проекции, которая определяет соответствие между координатами объектов на изображении и их реальными пространственными координатами.

Точность определения параметров матрицы проекции оказывает значительное влияние на качество реконструкции траекторий транспортных средств, оценку их скорости и других характеристик. Однако данный процесс сопровождается рядом сложностей, связанных с различными источниками погрешностей, включая искажения перспективы, геометрические особенности дорожной сцены, ошибки калибровки камеры и шумы в данных.

Целью данной работы является разработка метода минимизации погрешности при восстановлении параметров коэффициентов матрицы проекции на основе данных, полученных с дорожных камер.

В рамках курсовой работы решались следующие задачи:

1. Изучение и реализация модели камеры обскуры для проекции мировых координат на плоскость изображения.
2. Разработка и реализация метода восстановления погрешности матрицы проекции, основанного на геометрических ограничениях сцены
3. Оценка точности параметров матрицы проекции

# ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

## Модель камеры обскуры

Модель камеры-обскуры описывает математическую связь между координатами точки в трехмерном пространстве и ее проекцией на плоскость изображения идеальной камеры-обскуры, где апертура камеры описывается как точка, а линзы не используются для фокусировки света. Модель не включает, например, геометрические искажения или размытие несфокусированных объектов, вызванные линзами и апертурами конечного размера. Она также не принимает во внимание, что цифровые камеры имеют только дискретные координаты изображения. Это означает, что модель камеры-обскуры можно использовать только в качестве первого приближения преобразования 3D-сцены в 2D - изображение. Его достоверность зависит от качества камеры и, как правило, уменьшается от центра изображения к краям по мере увеличения эффектов искажения объектива.

Проективное преобразование, заданное моделью камеры-обскуры, показано ниже (1):

где – трехмерная точка в мировой системе координат,

– двумерный пиксель в плоскости изображения (используются однородные координаты),

– внутренняя матрица камеры,

и – матрица поворота и вектор перемещения, описывающие изменение координат от мира к камере,

s – произвольное масштабирование проективного преобразования, не являющееся частью модели камеры.

Внутренняя матрица камеры проецирует 3D-точки, заданные в системе координат камеры, в 2D-пиксельные координаты то есть (2):

Элементы внутренней матрицы камеры (3) включают фокусные расстояния и , выраженные в пикселях, и сдвиг центральной точки (,), которая обычно находится близко к центру изображения:

Матрица внутренних параметров не зависит от просматриваемой сцены. Таким образом, после оценки её можно использовать повторно, если фокусное расстояние фиксировано (в случае зум-объектива). Таким образом, если изображение с камеры масштабируется с коэффициентом, все эти параметры необходимо масштабировать (соответственно умножать/делить) на один и тот же коэффициент.

Совместная матрица вращения-переноса является матричным произведением проективного преобразования и однородного преобразования. Проективное преобразование 3 на 4 (4) отображает 3D-точки, представленные в координатах камеры, в 2D-точки на плоскости изображения и представленные в нормализованных координатах камеры и :

Однородное преобразование определяется внешними параметрами и и представляет собой изменение базиса с мировой системы координат на систему координат камеры . Таким образом, учитывая представление точки в мировых координатах, , мы получаем представление P в системе координат камеры, , по формуле (5):

то есть матрица однородного преобразования состоит из – матрицы вращения, и  – вектора переноса:

Получаем проективное преобразование, которое отображает 3D-точки в мировых координатах в 2D-точки на плоскости изображения и в нормированных координатах камеры (7):

где ,.

Соединяя вместе уравнения для внутренних и внешних характеристик, можно все записать в виде (8):

Если , то (8) примет вид (9):

где

На рисунке 1 показана модель камеры-обскуры.

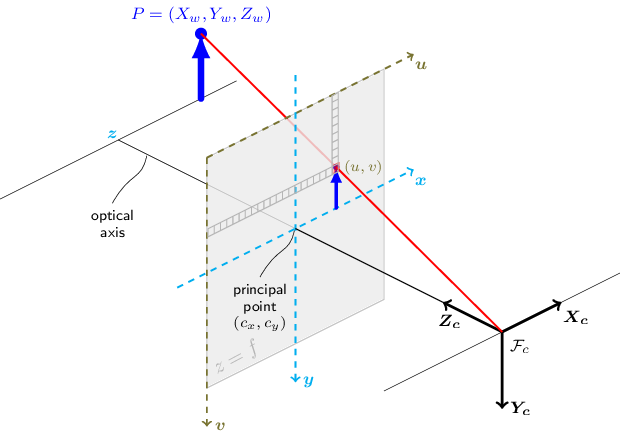


Рисунок 1 – Модель камеры-обскуры

1. **Ограничения сцены**
   1. **Положения камеры**

В качестве сцены для обзора выбирается перекрёсток или участок дороги, содержащий прямые линии и находящийся приблизительно в одной плоскости. Это допущение упрощает анализ движения транспортных средств, так как исключает сложные трехмерные структуры. Тогда внешняя матрица преобразования из мировой системы в систему камеры будет иметь вид:

где координаты относительно системы камеры, а координаты относительно мира.

Ориентация камеры в пространстве описывается матрицей вращения, которая определяется углами Тейта-Брайна. Эти углы задают последовательные повороты камеры относительно её осей.

Матрица поворота вокруг оси

Матрица поворота вокруг оси :

Матрица поворота вокруг оси :

Из (12), (13), (14) получаем, что матрица поворота будет иметь вид:

Операция перемножения матриц в (15) не коммутативна, поэтому изменение порядка перемножения матриц поворота приведет к различным результатам. Перемножая выражение (15), получим:

Внутреннюю матрицу преобразования (3) перепишем:

где – количество пикселей по вертикале, – количество пикселей по горизонтали. А (18) является отношением количества пикселей по горизонтали к количеству пикселей по вертикале:

То есть cсоединяя (17), (10), (15) получим следующую матрицу проекции:

Выразим из (19) матрицу проекции:

, где вектор параметров камеры соответственно фокусное расстояние, вращение вокруг координат и вектор переноса.

Таким образом внешние и внутренние параметры камеры задаются 7 коэффициентами. Это соответственно фокусное расстояние, углы Тейта-Брайна и вектор переноса. На рисунке 2 показано изображена мировая система координат и система координат камеры.

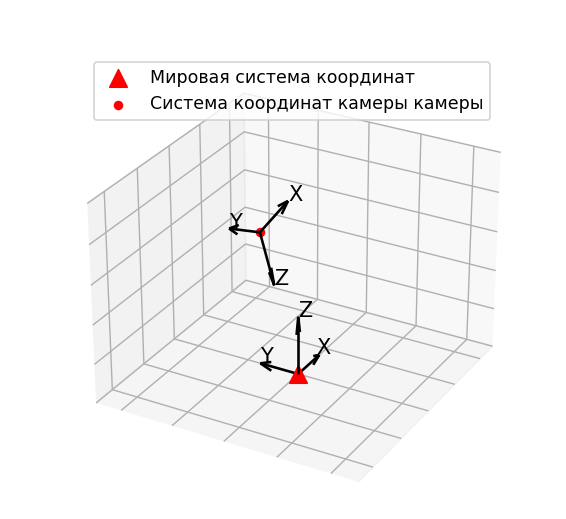


Рисунок 2 – Положение мировой системы координат и системы координат камеры

1. **Начальное решение**
   1. **Точки схода**

Точка схода — это точка на изображении, в которой сходятся параллельные в пространстве линии при их перспективной проекции. В компьютерном зрении и обработке изображений точка схода используется для анализа перспективы и определения ориентации объектов в 3D-пространстве**.** На рисунке 3 представлены две точки схода, обозначенные как R и F.

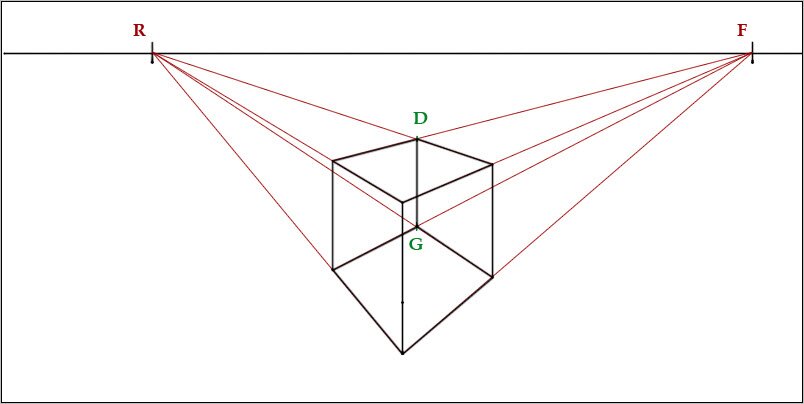


Рисунок 3 – Пример точек схода

Для вычисления точки схода согласно [1] воспользуемся минимальной суммой квадратов расстояния до всех параллельных прямых, проходящих через нее. То есть пусть будет единичной нормалью к прямой , которая проходит через конечные точки . Для точки схода ортогональное расстояние до прямой равно:

где количество параллельных прямых.

Минимизация суммы (21) эквивалентная решению линейной системы

где

* 1. **Начальное решение**

Для построения начальной матрицы поворота и фокусного расстояния вычислим две точки схода соответствующие осям мировой системы координат. После чего подставим в (23) и вычислим фокусное расстояние в соответствие [1]:

где это сдвиг центральной точки.

Составим внутреннюю матрицу камеры подставив фокусное расстояние в (17) и нормализуем точки схода в соответствие [1]:

Так как ортогональны, то может быть сформировано из векторного произведения:

Сформируем из матрицу поворота в соответствие по [2]:

Вычислим из матрицы поворота (26) начальные углы Тейта-Брайна относительно вращения :

где  вращения вокруг , вращения вокруг , вращения вокруг .

Начальный вектор переноса, следующий:

1. **Функция оптимизация**

Рассмотрим набор отрезков , где , такой что , где cсоответствуют началу отрезка, концу отрезка. Координаты являются однородными в системе координат камеры, однородные в мировой системе координат.

Основной целью является нахождение оптимальных параметров камеры

таких как фокусное расстояние, углы вращения камеры и компоненты вектора переноса для преобразования (20), которые минимизируют ошибку проекции между мировыми координатами и их проекциями на изображении.

Эти параметры должны быть определены путём минимизации функции стоимости, которая состоит из двух компонентов, каждая из которых измеряет ошибку между известными и проецируемыми отрезками.

**Первая компонента** измеряет ошибку между известным отрезком в плоскости изображения и спроецированным отрезком в плоскости изображения. Она вычисляется как сумма невязок между проекциями начальной и конечной точек отрезка:

**Вторая компонента** измеряет ошибку между углами, образованными отрезками в плоскости изображения и углами, образованными спроецированными отрезками:

Объединяя (30), (31) получим функцию стоимости:

где весовые коэффициенты.

В качестве метода оптимизации для минимизации функции стоимости (32) мы используем метод Левенберга-Маркварда, который является гибридом методов градиентного спуска и метода наименьших квадратов.

# ПРАКТИЧЕСКАЯ ЧАСТЬ

1. **Демонстрация на синтетических данных**

Проведём испытания на синтетических данных. Для этого сформируем сцену, содержащую прямые отрезки в плоскости . На рисунке 4 представлена сгенерированная сцена с прямыми линиями. Затем спроецируем эти линии на плоскость изображения, используя заданные параметры камеры. Результаты проекции отрезков на изображение показаны на рисунке 5, где отображение получено с использованием матрицы проекции камеры, основанной на синтетической сцене. Далее проведем оптимизацию параметров камеры, и в результате получим ошибку порядка 6.0361e-10. На рисунке 6 отображён график сходимости процесса оптимизации.

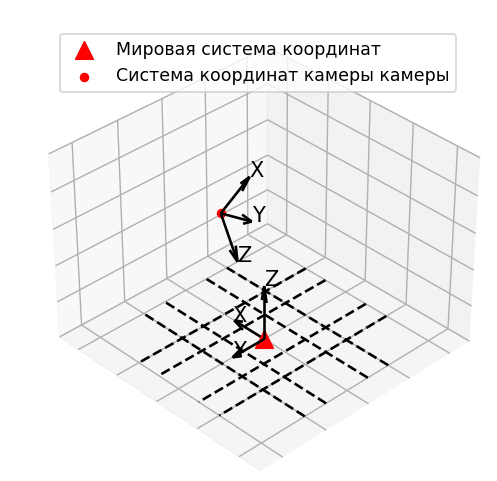


Рисунок 4 – Сгенерированная сцена с прямыми отрезками, расположенными в плоскости, и соответствующими системами координат (мировая и камера).

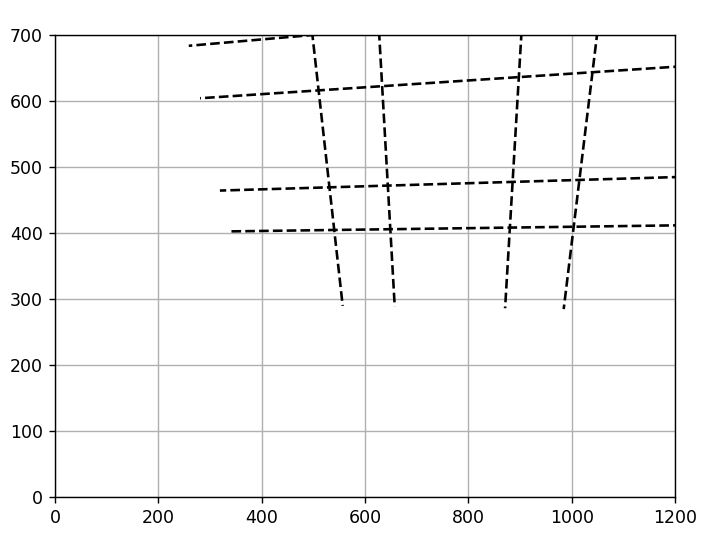


Рисунок 5 – Спроецированные прямые отрезки на плоскость изображения, полученные с использованием матрицы проекции камеры на основе синтетической сцены.

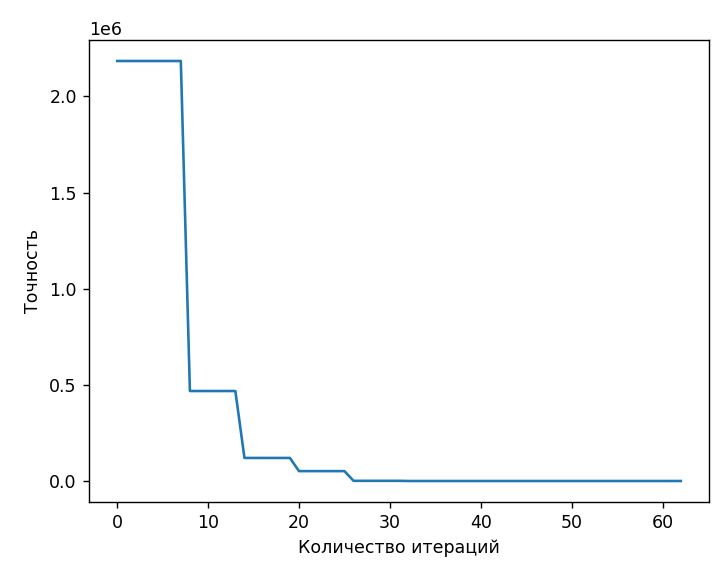


Рисунок 6 – График сходимости функции оптимизации

1. **Демонстрация на реальных данных**

Для проведения испытаний на реальных данных мы будем использовать камеру видеонаблюдения за дорожным движением. Чтобы минимизировать и подавить возможные искажения изображения, применим нейронную сеть GeoCalib [4], предназначенную для калибровки геопространственных данных. Для корректной работы с реальными данными и отображения сцены, введём мировую систему координат в плоскости . Чтобы определить параметры мировой системы координат, мы воспользуемся данными GPS, с помощью которых можно вычислить расстояния между отрезками и их длины. На рисунке 7 изображены размеченные отрезки с учетом мировой системы координат. На рисунке 8 изображены спроецированные линии из мировой системы координат в плоскость изображения. На рисунке 9 изображены и спроецированные отрезки и размеченные. На рисунке 10 показана сходимость функции ошибки.

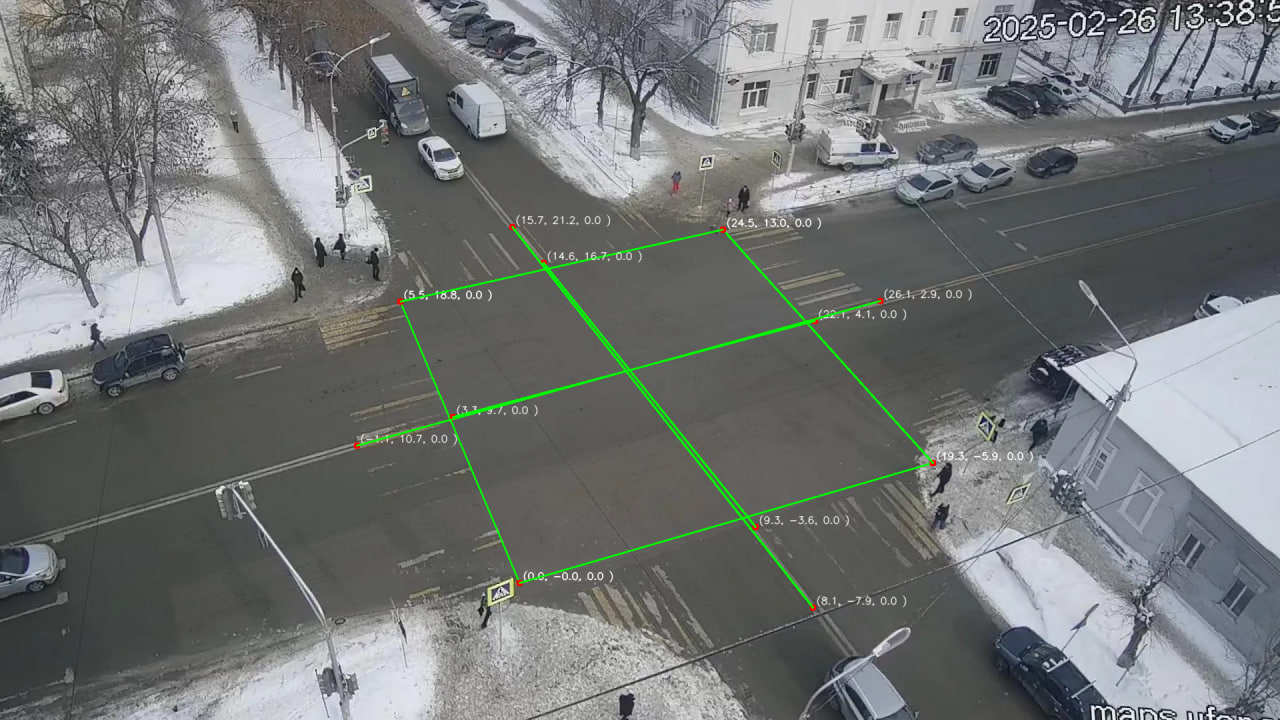


Рисунок 7 – Размеченные отрезки на перекрёстке улиц Пушкина и Аксакова, полученные с использованием камеры видеонаблюдения

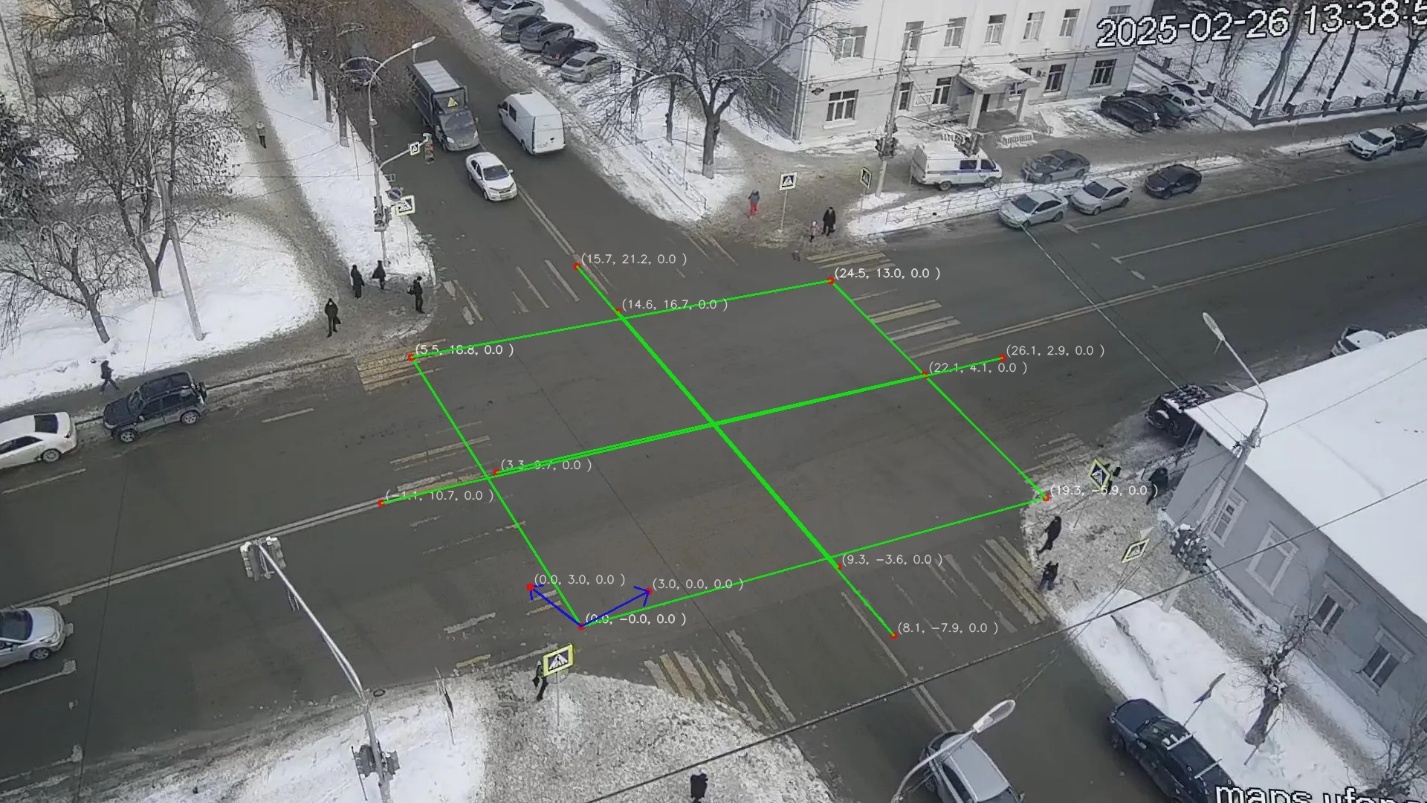


Рисунок 8 – Спроецированные прямые отрезки на перекрёстке улиц Пушкина и Аксакова, отображённые на плоскости изображения с учётом калибровки камеры и мировой системы координат.

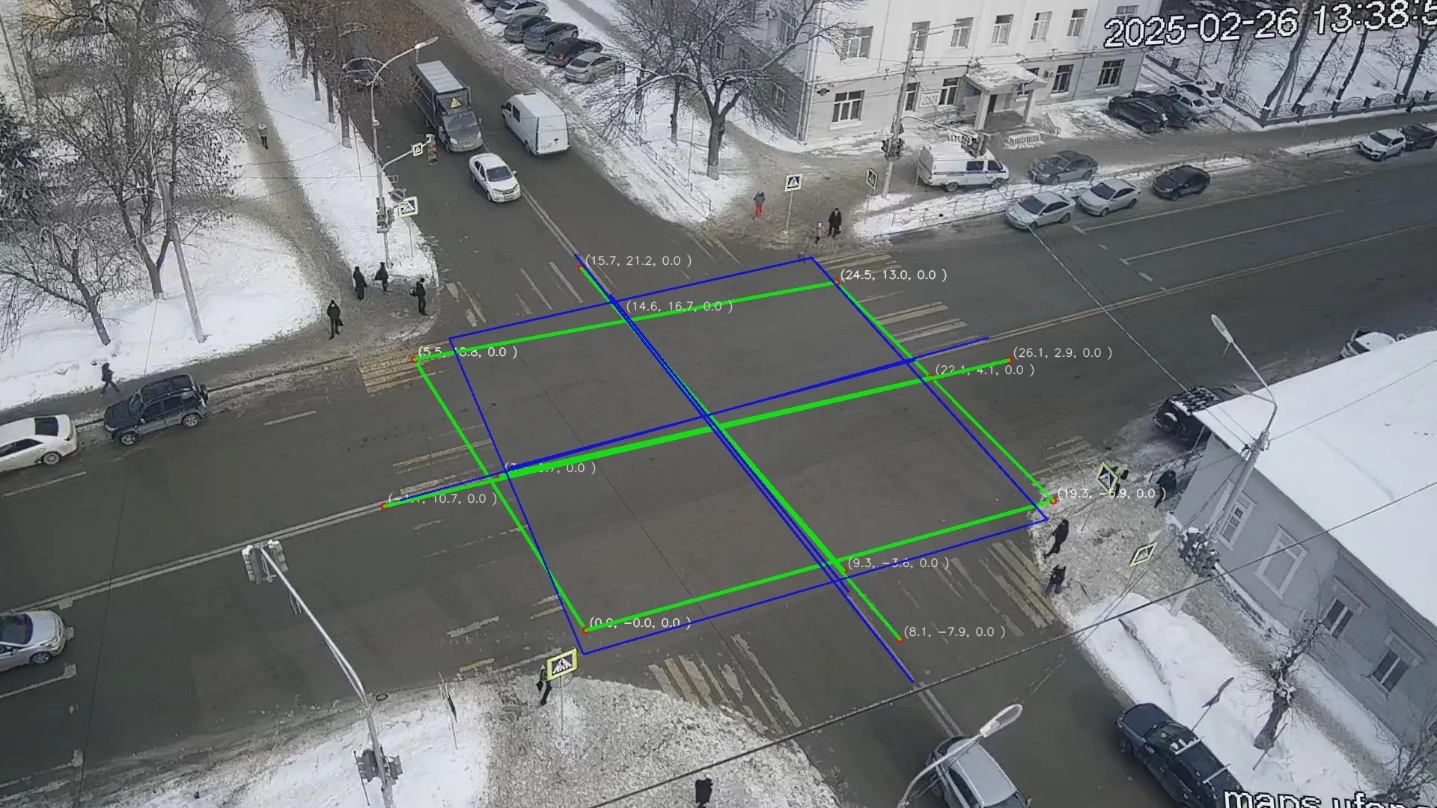


Рисунок 9 – Спроецированные прямые линии и размеченные линии

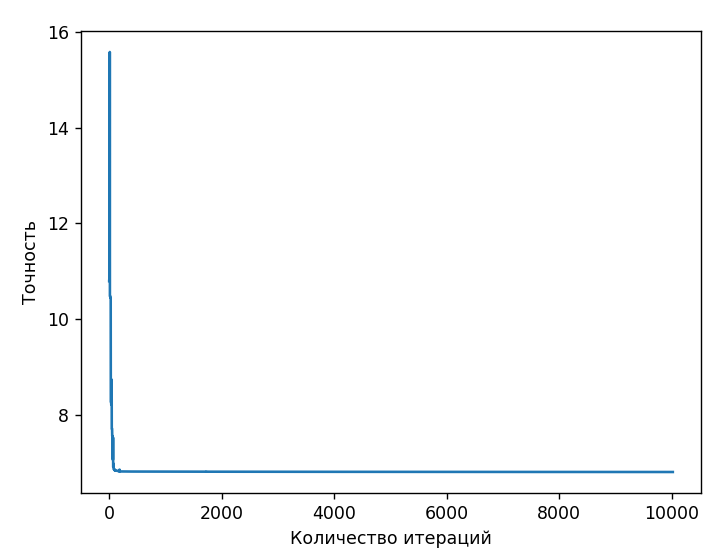


Рисунок 10 – График сходимости функции оптимизации

# ЗАКЛЮЧЕНИЕ

В рамках курсовой работы была изучена модель камеры обскуры для преобразования мировых координат в координаты на изображении, а также реализован метод восстановления погрешности матрицы проекции. При проведении испытаний были сделаны следующие выводы:

* Метод показал хорошую эффективность при обработке точных данных, обеспечивая высокую точность восстановления параметров матрицы проекции.
* Метод чувствителен к качеству исходных данных, что ограничивает его применение в реальных условиях с неоптимальными или шумными данными.

# СПИСОК ЛИТЕРАТУРЫ

1. Masoud, Osama, and Nikolaos P. Papanikolopoulos. "Using Geometric Primitives to Calibrate Traffic Scenes." Computer Vision and Image Understanding, vol. 109, no. 2, 2007, pp. 74-93.
2. He, B. W., Zhou, X. L., & Li, Y. F. (2011). A new camera calibration method from vanishing points in a vision system. Transactions of the Institute of Measurement and Control, 33(7), 806–822.
3. Тёрк, М. Компьютерное зрение. Передовые методы и глубокое обучение / М. Тёрк, Р. Дэвис ; перевод с английского В. С. Яценкова. — Москва : ДМК Пресс, 2022. — 690 с. — ISBN 978-5-93700-148-1.
4. A. Veicht, P.-E. Sarlin, P. Lindenberger, and M. Pollefeys, GeoCalib: Learning Single-image Calibration with Geometric Optimization, European Conference on Computer Vision (ECCV), 2024. Available at: <https://github.com/cvg/GeoCalib>

**ПРИЛОЖЕНИЕ А**

**(обязательное)**

**src/camera\_model.py**

import numpy as np

import cv2

from scipy.spatial.transform import Rotation

from .point2D import Point2D

from .point3D import Point3D

class Camera:

def \_\_init\_\_(self):

self.size = None

self.scene = None

self.tau = None

self.f = None

self.A = np.zeros((3, 3))

self.R = np.zeros((3, 3))

self.T = np.zeros((3, 1)).reshape(-1, 1)

def set\_params(self, params):

if len(params) == 5:

self.calc\_A(params[0])

self.calc\_R(params[1:4])

self.calc\_T(z=params[4])

elif len(params) == 7:

self.calc\_A(params[0])

self.calc\_R(params[1:4])

self.calc\_T(x=params[4], y=params[5], z=params[6])

def get\_scene(self):

return self.scene

def get\_f(self):

return self.f

def get\_tau(self):

return self.tau

def calc\_tau(self, height, width):

self.size = [height, width] # высота и ширина

self.tau = height / width

def load\_scene(self, path):

self.scene = cv2.imread(path)

height, width, channels = self.scene.shape

print(height,width)

self.calc\_tau(height, width)

# вычисление матрицы поворота

def calc\_R(self, euler\_angles):

rot = Rotation.from\_euler('zxy', euler\_angles, degrees=True)

self.R = rot.as\_matrix()

def set\_init\_R(self, p):

self.R = np.vstack(p).transpose()

def get\_R(self, angle\_output=False, output=False):

if angle\_output:

angles = Rotation.from\_matrix(self.R).as\_euler('zxy', degrees=True)

# print(angles)

return angles

if output:

print(f'Матрица поворота:\n{self.R}')

return self.R

# вычисление столбца переноса

def calc\_T(self, x=0, y=0, z=0):

self.T = np.array([x, y, z])

def get\_T(self, output=False):

if output:

print(f'Столбец переноса:\n{self.T}')

return self.T

# вычисление внутренней матрицы

def calc\_A(self, f, using\_tau=True):

self.f = f

if using\_tau:

self.A = np.array([[f, 0, self.size[1] / 2],

[0, f \* self.tau, self.size[0] / 2],

[0, 0, 1]])

# self.A = np.array([[f, 0, 0],

# [0, f \* self.tau, 0],

# [0, 0, 1]])

else:

self.A = np.array([[f, 0, self.size[1] / 2],

[0, f, self.size[0] / 2],

[0, 0, 1]])

def get\_A(self, output=False):

if output:

print(f'Внутренние параметры камеры:\n{self.A}')

return self.A

# прямое преобразование

def direct\_transform\_world(self, point\_real: Point3D, params=[]) -> Point2D:

if len(params) == 5:

self.calc\_A(params[0])

self.calc\_R(params[1:4])

self.calc\_T(z=params[4])

elif len(params) == 6:

self.calc\_A(params[0])

self.calc\_R(params[1:4])

self.calc\_T(x=params[4], z=params[4])

elif len(params) == 7:

self.calc\_A(params[0])

self.calc\_R(params[1:4])

self.calc\_T(x=params[4], y=params[5], z=params[6])

\_T1 = -self.R @ self.T

\_RT = np.hstack([self.R, \_T1[:, np.newaxis]])

\_AT = self.A @ \_RT

\_new\_point = Point2D(\_AT.dot(point\_real.get(out\_homogeneous=True)))

return \_new\_point

def direct\_transform\_camera(self, point\_real: Point3D, params=[]) -> Point2D:

if len(params) == 5:

self.calc\_A(params[0])

self.calc\_R(params[1:4])

self.calc\_T(z=params[4])

elif len(params) == 6:

self.calc\_A(params[0])

self.calc\_R(params[1:4])

self.calc\_T(x=params[4], z=params[4])

elif len(params) == 7:

self.calc\_A(params[0])

self.calc\_R(params[1:4])

self.calc\_T(x=params[4], y=params[5], z=params[6])

\_new\_point = Point2D(self.A @ point\_real.get())

return \_new\_point

def back\_transform\_world(self, point\_image: Point2D, params=[]) -> Point2D:

if len(params) == 5:

self.calc\_A(params[0])

self.calc\_R(params[1:4])

self.calc\_T(z=params[4])

elif len(params) == 7:

self.calc\_A(params[0])

self.calc\_R(params[1:4])

self.calc\_T(x=params[4], y=params[5], z=params[6])

\_T1 = -self.R @ self.T

\_RT = np.hstack([self.R, \_T1[:, np.newaxis]])

\_RT = np.delete(\_RT, 2, axis=1)

\_AT = self.A @ \_RT

\_AT\_inv = np.linalg.inv(\_AT)

# print(\_AT\_inv)

# print(point\_image.get(out\_homogeneous=True))

\_new\_point = Point2D(\_AT\_inv @ point\_image.get(out\_homogeneous=True))

return \_new\_point

**src/distance.py**

import numpy as np

from pyproj import Geod

def gps\_to\_enu(lat, lon, ref\_lat, ref\_lon):

"""

Перевод GPS (широта, долгота) в локальные координаты ENU (в метрах)

"""

geod = Geod(ellps="WGS84")

# Вычисляем расстояние и азимут до точки

azimuth, \_, distance = geod.inv(ref\_lon, ref\_lat, lon, lat)

# Преобразуем в координаты ENU

east = distance \* np.sin(np.deg2rad(azimuth))

north = distance \* np.cos(np.deg2rad(azimuth))

return east, north

**src/initsolution.py**

import numpy as np

from .camera\_model import Camera

# вычисление нормали к линии (вектора направления)

def \_normal\_vector(x1, y1, x2, y2):

dx = x2 - x1

dy = y2 - y1

normal = np.array([-dy, dx]) / np.sqrt(dx \* dx + dy \* dy)

return normal

# поиск точек схода для набора линий

def \_search\_vanishing\_point(lines):

A = []

b = []

for line in lines:

# print(line)

(x1, y1), (x2, y2) = line

n = \_normal\_vector(x1, y1, x2, y2)

A.append(n)

b.append(np.dot(n, [x1, y1]))

A = np.array(A)

b = np.array(b)

v = np.linalg.lstsq(A, b, rcond=None)[0]

return v

# поиск точек схода для нескольких осей

def \_search\_vanishing\_points(lines):

v = []

for line in lines:

\_v = \_search\_vanishing\_point(line)

v.append(\_v)

return v

# вычисление нормализованный точек схода

def \_calc\_norm\_vanishing\_points(vx, vy, camera):

px = np.linalg.inv(camera.get\_A().transpose()) @ np.transpose(np.hstack([vx, 1]))

py = np.linalg.inv(camera.get\_A().transpose()) @ np.transpose(np.hstack([vy, 1]))

pz = px \* py

return px, py, pz

# вычисление фокусного расстояния

def \_calc\_f(vx, vy, camera=None):

if camera is None:

return np.sqrt(-np.dot(vx, vy))

else:

M = np.array([[1, 0], [0, camera.tau \*\* (-2)]])

return np.sqrt(abs(vx.T @ M @ vy))

def calc\_init\_camera(path, lines) -> Camera:

camera = Camera()

camera.load\_scene(path)

v = \_search\_vanishing\_points(lines)

# print(v)

f = \_calc\_f(v[0], v[1], camera)

# print(f)

camera.calc\_A(f)

px, py, pz = \_calc\_norm\_vanishing\_points(v[0], v[1], camera)

# print(px,py,pz)

camera.set\_init\_R([pz, px, py])

# print(np.around(camera.get\_R(angle\_output=True), 2))

camera.calc\_T(z=30)

return camera

**src/optimization.py**

import numpy as np

from scipy.optimize import least\_squares

from scipy.optimize import minimize

from .camera\_model import Camera

from .point2D import Point2D

from .point3D import Point3D

class Optimizer:

def \_\_init\_\_(self, camera: Camera):

self.camera = camera

def error\_point\_to\_point(self, line\_known: tuple[Point2D, Point2D],

line\_predicted: tuple[Point2D, Point2D]) -> float:

known\_start, known\_end = line\_known

predicted\_start, predicted\_end = line\_predicted

error = np.linalg.norm(known\_start.get() - predicted\_start.get()) + \

np.linalg.norm(known\_end.get() - predicted\_end.get())

return error

def error\_shape(self, line\_known, line\_predicted):

""" Ошибка, основанная на косинусном расстоянии между векторами линий """

known\_start, known\_end = line\_known

predicted\_start, predicted\_end = line\_predicted

v1 = known\_end.get() - known\_start.get()

v2 = predicted\_end.get() - predicted\_start.get()

cos\_sim = np.dot(v1, v2) / (np.linalg.norm(v1) \* np.linalg.norm(v2))

return 1 - cos\_sim # Чем ближе к 0, тем лучше

def error\_line(self, line\_known: tuple[Point2D, Point2D],

line\_predicted: tuple[Point2D, Point2D]) -> float:

known\_start, known\_end = line\_known

predicted\_start, predicted\_end = line\_predicted

# Длина линий

known\_length = np.linalg.norm(known\_end.get() - known\_start.get())

predicted\_length = np.linalg.norm(predicted\_end.get() - predicted\_start.get())

def compute\_angle(p1, p2):

delta = p2.get() - p1.get()

return np.arctan2(delta[1], delta[0])

known\_angle = compute\_angle(known\_start, known\_end)

predicted\_angle = compute\_angle(predicted\_start, predicted\_end)

# Ошибка по длине

length\_error = abs(predicted\_length - known\_length)

# Ошибка по углу (в радианах)

angle\_error = abs(predicted\_angle - known\_angle)

# return length\_error + 10 \* angle\_error

# print(angle\_error)

return 10 \* angle\_error

def residuals\_reprojection(self, params: np.ndarray,

lines: list[tuple[tuple[Point2D, Point3D], tuple[Point2D, Point3D]]]) -> np.ndarray:

residuals = []

for known\_start, known\_end in lines:

known\_start\_2D, known\_start\_3D = known\_start

known\_end\_2D, known\_end\_3D = known\_end

predicted\_start\_2D = self.camera.direct\_transform\_world(known\_start\_3D, params)

predicted\_end\_2D = self.camera.direct\_transform\_world(known\_end\_3D, params)

error1 = self.error\_point\_to\_point((known\_start\_2D, known\_end\_2D), (predicted\_start\_2D, predicted\_end\_2D))

error2 = self.error\_line((known\_start\_2D, known\_end\_2D), (predicted\_start\_2D, predicted\_end\_2D))

error3 = self.error\_shape((known\_start\_2D, known\_end\_2D), (predicted\_start\_2D, predicted\_end\_2D))

# residuals.append(log\_error(error1) + log\_error(error2))

# residuals.append(error1 + error2)

residuals.append(0.2 \* error1 + error2)

return np.array(residuals)

def residuals\_back\_reprojection(self, params: np.ndarray,

lines: list[tuple[tuple[Point2D, Point2D], tuple[Point2D, Point2D]]]) -> np.ndarray:

residuals = []

for known\_start, known\_end in lines:

known\_start\_2D, known\_start\_3D = known\_start # первая точка в пиксялях, вторая в реальных координатах z=0

known\_end\_2D, known\_end\_3D = known\_end

predicted\_start\_3D = self.camera.back\_transform\_world(known\_start\_2D, params)

predicted\_end\_3D = self.camera.back\_transform\_world(known\_end\_2D, params)

error1 = self.error\_point\_to\_point((known\_start\_3D, known\_end\_3D), (predicted\_start\_3D, predicted\_end\_3D))

error2 = self.error\_line((known\_start\_3D, known\_end\_3D), (predicted\_start\_3D, predicted\_end\_3D))

residuals.append(error1 + error2)

return np.array(residuals)

def optimize\_init(self, lines: list[tuple[tuple[Point2D, Point3D], tuple[Point2D, Point3D]]]):

angles = self.camera.get\_R(angle\_output=True)

x0 = [self.camera.get\_f(), \*angles, 10]

result = least\_squares(self.residuals\_reprojection, x0, args=(lines,), method='trf')

return self.camera, result

def optimize\_reprojection(self, lines: list[tuple[tuple[Point2D, Point3D], tuple[Point2D, Point3D]]]):

angles = self.camera.get\_R(angle\_output=True)

# x0 = [self.camera.get\_f() , \*angles, 20]

# x0 = [900, -99.58434695, 37.91236625, -167.6947188, 31.72150605]

x0 = [930, -99.58434695, 37.91236625, -167.6947188, 1, 1, 31.72150605]

cost\_history = []

history = []

def wrapped\_residuals(params):

residuals = self.residuals\_reprojection(params, lines)

cost = 0.5 \* np.sum(residuals \*\* 2) # Вычисляем cost

cost\_history.append(cost) # Сохраняем cost

history.append(params.copy())

return residuals

# bounds = ([800, -180, -180, -180, 10], [1500, 180, 180, 180, 60])

result = least\_squares(wrapped\_residuals, x0, method='lm', verbose=2, max\_nfev=10000)

return self.camera, result, cost\_history, history

def optimize\_back\_reprojection\_LM(self, lines: list[tuple[tuple[Point2D, Point2D], tuple[Point2D, Point2D]]]):

angles = self.camera.get\_R(angle\_output=True)

# x0 = [self.camera.get\_f(), \*angles, 10]

x0 = [931.45763154, -99.58434695, 37.91236625, -167.6947188, 31.72150605]

#

result = least\_squares(self.residuals\_back\_reprojection, x0, args=(lines,), method='lm',

verbose=2, # подробно видно как сходится

loss='huber',

# max\_nfev=20000 # кол-во итераций

)

return self.camera, result

def optimize\_back\_reprojection\_NM(self, lines: list[tuple[tuple[Point2D, Point2D], tuple[Point2D, Point2D]]]):

angles = self.camera.get\_R(angle\_output=True)

# x0 = [self.camera.get\_f(), \*angles, 10]

x0 = [931.45763154, -50, 0, -150, 31.72150605]

def callback(xk):

residuals = self.residuals\_back\_reprojection(xk, lines)

loss = sum(residuals \*\* 2)

print(f"Function value at iteration: {loss}")

result = minimize(

lambda x: sum(self.residuals\_back\_reprojection(x, lines) \*\* 2), # Сумма квадратов ошибок

x0,

method='Nelder-Mead',

options={

'maxiter': 1000,

'disp': True # Показывать процесс оптимизации

},

callback=callback

)

return self.camera, result

**src/plot.py**

import cv2

import matplotlib.pyplot as plt

from .camera\_model import Camera

from .point2D import Point2D

from .point3D import Point3D

class Plot:

def \_\_init\_\_(self, camera):

self.camera = camera

self.scene\_plot = self.camera.get\_scene().copy()

# cv2.line(self.camera.get\_scene(), (828, 689), (927, 262), (0, 0, 0), 2)

# cv2.line(self.camera.get\_scene(), (828, 700), (290, 513), (0, 0, 0), 2)

# cv2.putText(self.camera.get\_scene(), 'OX', (927, 262), cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0, 0, 0), 2, cv2.LINE\_AA)

# cv2.putText(self.camera.get\_scene(), 'OY', (290, 513), cv2.FONT\_HERSHEY\_SIMPLEX, 1, (0, 0, 0), 2, cv2.LINE\_AA)

def \_draw\_point\_with\_label(self, img, point, coords):

cv2.circle(img, point, 5, (0, 0, 255), -1)

if len(coords) == 2:

text = f"({coords[0]:.1f}, {coords[1]:.1f})"

else:

text = f"({coords[0]:.1f}, {coords[1]:.1f}, {coords[2]:.1f} )"

cv2.putText(img, text, (point[0] + 5, point[1] - 5),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (255, 255, 255), 1, cv2.LINE\_AA)

def \_get\_cv2\_format(self, point: Point2D):

return tuple(map(int, point.get()))

def draw\_tranform\_line(self, lines, save=False, out\_jupyter=False, params=[]):

scene = self.camera.get\_scene().copy()

overlay = scene.copy()

for start, end in lines:

start\_trans = self.camera.direct\_transform\_world(start[1], params)

end\_trans = self.camera.direct\_transform\_world(end[1], params)

start\_plot = self.\_get\_cv2\_format(start\_trans)

self.\_draw\_point\_with\_label(overlay, start\_plot, start[1].get())

end\_plot = self.\_get\_cv2\_format(end\_trans)

self.\_draw\_point\_with\_label(overlay, end\_plot, end[1].get())

cv2.line(overlay, start\_plot,

end\_plot, (0, 255, 0), 3)

for start, end in lines:

# start\_trans = self.camera.direct\_transform\_world(start[1], params)

# end\_trans = self.camera.direct\_transform\_world(end[1], params)

start\_plot = self.\_get\_cv2\_format(start[0])

# self.\_draw\_point\_with\_label(overlay, start\_plot, start[0].get())

end\_plot = self.\_get\_cv2\_format(end[0])

# self.\_draw\_point\_with\_label(overlay, end\_plot, end[1].get())

cv2.line(overlay, start\_plot,

end\_plot, (255, 0, 0), 2)

# start, end = Point3D([0, 0, 0]), Point3D([0, 3, 0])

# start\_trans, end\_trans = self.camera.direct\_transform\_world(start, params), self.camera.direct\_transform\_world(

# end, params)

# start\_plot, end\_plot = self.\_get\_cv2\_format(start\_trans), self.\_get\_cv2\_format(end\_trans)

# cv2.arrowedLine(overlay, start\_plot, end\_plot, (255, 0, 0), 2, tipLength=0.2)

# end\_plot\_point = self.\_get\_cv2\_format(end\_trans)

# self.\_draw\_point\_with\_label(overlay, end\_plot\_point, end.get())

# start, end = Point3D([0, 0, 0]), Point3D([3, 0, 0])

# start\_trans, end\_trans = self.camera.direct\_transform\_world(start, params), self.camera.direct\_transform\_world(

# end, params)

# start\_plot, end\_plot = self.\_get\_cv2\_format(start\_trans), self.\_get\_cv2\_format(end\_trans)

# end\_plot\_point = self.\_get\_cv2\_format(end\_trans)

# self.\_draw\_point\_with\_label(overlay, end\_plot\_point, end.get())

# cv2.arrowedLine(overlay, start\_plot, end\_plot, (255, 0, 0), 2, tipLength=0.2)

alpha = 0.8

cv2.addWeighted(overlay, alpha, scene, 1 - alpha, 0, scene)

if not save and not out\_jupyter:

cv2.imshow('Вид сцены калибровочный', scene)

cv2.waitKey(0)

cv2.destroyAllWindows()

elif out\_jupyter:

scene\_rgb = cv2.cvtColor(scene, cv2.COLOR\_BGR2RGB)

plt.figure(figsize=(10, 8))

plt.imshow(scene\_rgb)

plt.axis('off')

plt.show()

else:

cv2.imwrite('evalution\_scene.png', scene)

def draw\_transform\_point(self, points, save=False, out\_jupyter=False, params=[]):

scene = self.camera.get\_scene().copy()

overlay = scene.copy()

for point in points:

point\_trans = self.camera.direct\_transform\_world(point, params)

start\_plot = self.\_get\_cv2\_format(point\_trans)

self.\_draw\_point\_with\_label(overlay, start\_plot, point\_trans.get())

alpha = 0.8

cv2.addWeighted(overlay, alpha, scene, 1 - alpha, 0, scene)

if not save and not out\_jupyter:

scene\_resized = cv2.resize(scene, (600, 400)) # Масштабируем изображение

cv2.imshow('Вид сцены калибровочный', scene\_resized)

cv2.waitKey(0)

cv2.destroyAllWindows()

elif out\_jupyter:

scene\_rgb = cv2.cvtColor(scene, cv2.COLOR\_BGR2RGB)

plt.figure(figsize=(10, 8))

plt.imshow(scene\_rgb)

plt.axis('off')

plt.show()

else:

cv2.imwrite('../data/crossroads\_karls\_marks/evalution\_scene.png', scene)

def draw\_calibration\_line(self, lines: list[tuple[tuple[Point2D, Point3D], tuple[Point2D, Point3D]]], save=False,

out\_jupyter=False):

scene = self.camera.get\_scene()

for start, end in lines:

start\_plot = self.\_get\_cv2\_format(start[0])

end\_plot = self.\_get\_cv2\_format(end[0])

self.\_draw\_point\_with\_label(scene, start\_plot, start[1].get())

self.\_draw\_point\_with\_label(scene, end\_plot, end[1].get())

cv2.line(scene, start\_plot,

end\_plot, (0, 255, 0), 2)

if not save and not out\_jupyter:

cv2.imshow('Вид сцены калибровочный', self.camera.get\_scene())

cv2.waitKey(0)

cv2.destroyAllWindows()

elif out\_jupyter:

scene\_rgb = cv2.cvtColor(scene, cv2.COLOR\_BGR2RGB)

plt.figure(figsize=(10, 8))

plt.imshow(scene\_rgb)

plt.axis('off')

plt.show()

else:

cv2.imwrite('calibration\_line.png', scene)

def draw\_tranform\_net(self, lines, save=False, out\_jupyter=False, params=[]):

scene = self.camera.get\_scene().copy()

overlay = scene.copy()

for start, end in lines:

\_start = start[1]

\_end = end[1]

start\_trans = self.camera.direct\_transform\_world(\_start, params)

end\_trans = self.camera.direct\_transform\_world(\_end, params)

start\_plot = self.\_get\_cv2\_format(start\_trans)

# self.\_draw\_point\_with\_label(overlay, start\_plot, start[1].get())

end\_plot = self.\_get\_cv2\_format(end\_trans)

# self.\_draw\_point\_with\_label(overlay, end\_plot, end[1].get())

cv2.line(overlay, start\_plot,

end\_plot, (0, 255, 0), 2)

\_start = \_start.set\_Z(3)

print(\_start.get())

start\_trans = self.camera.direct\_transform\_world(\_start, params)

# end\_trans = self.camera.direct\_transform\_world(\_end.set\_Z(3), params)

start\_plot = self.\_get\_cv2\_format(start\_trans)

end\_plot = self.\_get\_cv2\_format(end\_trans)

alpha = 0.8

cv2.addWeighted(overlay, alpha, scene, 1 - alpha, 0, scene)

if not save and not out\_jupyter:

cv2.imshow('Вид сцены калибровочный', scene)

cv2.waitKey(0)

cv2.destroyAllWindows()

elif out\_jupyter:

scene\_rgb = cv2.cvtColor(scene, cv2.COLOR\_BGR2RGB)

plt.figure(figsize=(10, 8))

plt.imshow(scene\_rgb)

plt.axis('off')

plt.show()

else:

cv2.imwrite('evalution\_scene\_net.png', scene) **src/point2D.py**

import numpy as np

# по умолчанию грузим гомогенные координаты

class Point2D:

def \_\_init\_\_(self, coord):

if len(coord) == 2:

coord = np.append(coord, 1)

self.coord = coord

elif len(coord) == 3:

self.coord = coord

def set(self, coord):

self.coord = coord

def get(self, out\_homogeneous=False):

if out\_homogeneous:

return self.coord

else:

return self.coord[:-1] / self.coord[-1]

**src/point3D.py**

import numpy as np

class Point3D:

def \_\_init\_\_(self, coord):

if len(coord) == 3:

coord = np.append(coord, 1)

self.coord = coord

elif len(coord) == 4:

self.coord = coord

def set(self, coord):

self.coord = coord

def set\_Z(self, z):

self.coord[2] = z

# по умолчанию неоднородные координаты

def get(self, out\_homogeneous=False):

if out\_homogeneous:

return self.coord

else:

return np.array(self.coord[:-1]) / self.coord[-1]

**example\_direct.py**

from src.camera\_model import Camera

from src.optimizetion import Optimizer

from src.initsolution import calc\_init\_camera

from src.plot import Plot

from src.point3D import Point3D

from src.point2D import Point2D

from src.distance import gps\_to\_enu

import numpy as np

Line\_Y = [[[297, 521], [1365, 272]], [[378, 555], [1462, 301]], [[417, 702], [1398, 430]], [[843, 894], [1343, 720]],

[[1197, 283], [1396, 244]]]

Line\_X = [[[755, 810], [601, 453]], [[1258, 962], [745, 315]], [[1388, 653], [1096, 345]], [[949, 268], [852, 179]]]

camera = calc\_init\_camera('../../data/crossroads\_pushkin\_aksakov/crossroads\_not\_dist.jpg', [Line\_X, Line\_Y])

# Опорная точка (центр локальной системы)

ref\_lat, ref\_lon = 54.723767, 55.933369

LINE\_CALIB = [

[[54.723767, 55.933369, 779, 874], [54.723936, 55.933454, 600, 452]],

[[54.723767, 55.933369, 779, 874], [54.723714, 55.933668, 1399, 694]],

[[54.723714, 55.933668, 1399, 694], [54.723884, 55.933750, 1084, 344]],

[[54.723884, 55.933750, 1084, 344], [54.723936, 55.933454, 600, 452]],

[[54.723854, 55.933420, 679, 625], [54.723804, 55.933712, 1222, 481]],

[[54.723735, 55.933514, 1133, 790], [54.723917, 55.933596, 815, 394]],

[[54.723863, 55.933352, 535, 668], [54.723793, 55.933774, 1320, 451]],

[[54.723696, 55.933495, 1219, 911], [54.723957, 55.933613, 768, 340]],

# [[54.723889, 55.933191, 95, 803], [54.723761, 55.933949, 1565, 392]],

# [[54.723764, 55.933953, 1558, 386], [54.723847, 55.933996, 1395, 268]],

]

LINE\_CALIB\_NEW = []

# Переводим координаты первой линии в ENU

for line in LINE\_CALIB:

(lat1, lon1, x1, y1), (lat2, lon2, x2, y2) = line

e1, n1 = gps\_to\_enu(lat1, lon1, ref\_lat, ref\_lon)

e2, n2 = gps\_to\_enu(lat2, lon2, ref\_lat, ref\_lon)

LINE\_CALIB\_NEW.append([[x1, y1, float(e1), float(n1), 0], [x2, y2, float(e2), float(n2), 0]])

LINE\_PREP = []

for line in LINE\_CALIB\_NEW:

start, end = line

start2D, start3D = Point2D(start[0:2]), Point3D(start[2:6])

end2D, end3D = Point2D(end[0:2]), Point3D(end[2:6])

LINE\_PREP.append([(start2D, start3D), (end2D, end3D)])

print(LINE\_CALIB\_NEW)

camera.set\_params([929.67, -141.65, 17.12, -186.47, 5.31, 3.68, 27.73])

optimize = Optimizer(camera)

camera, info, cost\_history, history = optimize.optimize\_reprojection(LINE\_PREP)

print("Финальная ошибка:", info.cost)

print("Финальные параметры:", np.around(info.x, 2))

plot = Plot(camera)

plot.draw\_tranform\_line(LINE\_PREP, save=True)

# plot.draw\_calibration\_line(LINE\_PREP, save=True)

#

import matplotlib.pyplot as plt

plt.plot(np.arange(0, len(cost\_history)), np.log(cost\_history))

plt.ylabel('Точность')

plt.xlabel('Количество итераций')

plt.show()

**draw.py**

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

import numpy as np

from scipy.spatial.transform import Rotation

from src.camera\_model import Camera

from src.point3D import Point3D

from src.point2D import Point2D

from src.optimizetion import Optimizer

def init(h):

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

ax.zaxis.line.set\_color((1.0, 1.0, 1.0, 0.0)) # Ось Z

ax.xaxis.line.set\_color((1.0, 1.0, 1.0, 0.0)) # Ось X

ax.yaxis.line.set\_color((1.0, 1.0, 1.0, 0.0)) # Ось Y

ax.xaxis.set\_tick\_params(labelleft=False, labelbottom=False) # Убираем метки для оси X

ax.yaxis.set\_tick\_params(labelleft=False, labelbottom=False) # Убираем метки для оси Y

ax.zaxis.set\_tick\_params(labelleft=False, labelbottom=False) # Убираем метки для оси Z

ax.xaxis.set\_ticks\_position('none') # Убираем засечки для оси X

ax.yaxis.set\_ticks\_position('none') # Убираем засечки для оси Y

ax.zaxis.set\_ticks\_position('none') # Убираем засечки для оси Z

# Настройка углов обзора

ax.view\_init(elev=20, azim=30)

ax.set\_proj\_type('persp')

ax.set\_zlim(0, h + 10)

ax.set\_xlim(-50, 50)

ax.set\_ylim(-50, 50)

return ax

def plot\_axies(position, angles=[]):

if not angles:

ax.quiver(\*position, 15, 0, 0, color='black')

ax.quiver(\*position, 0, 15, 0, color='black')

ax.quiver(\*position, 0, 0, 15, color='black')

ax.scatter(0, 0, 0, marker='^', s=100, color='red', label='Мировая система координат')

text\_size = 12

ax.text(position[0] + 15, position[1] + 1, position[2], 'X', color='black', fontsize=text\_size)

ax.text(position[0], position[1] + 15, position[2], 'Y', color='black', fontsize=text\_size)

ax.text(position[0], position[1], position[2] + 15, 'Z', color='black', fontsize=text\_size)

else:

rot = Rotation.from\_euler('zxy', angles, degrees=True).as\_matrix()

transform = np.eye(4)

transform[:3, :3] = rot

transform[:3, 3] = -rot @ position

x\_position = transform @ np.array([15, 0, 0, 1])

y\_position = transform @ np.array([0, 15, 0, 1])

z\_position = transform @ np.array([0, 0, 15, 1])

origin = transform @ np.array([0, 0, 0, 1])

# print(f'Положение камеры:\nx: {x\_position[:-1]}\ny: {y\_position[:-1]}\nz: {z\_position[:-1]}')

distances = np.linalg.norm(transform[:3, 3])

ax.scatter(\*transform[:3, 3], color='red', label='Система координат камеры камеры')

# label=f'{np.around(transform[:3, 3], 2)}, расстонияние до центра {round(distances, 2)}')

ax.quiver(\*origin[:-1], \*(x\_position[:-1] - origin[:-1]), color='black')

ax.quiver(\*origin[:-1], \*(y\_position[:-1] - origin[:-1]), color='black')

ax.quiver(\*origin[:-1], \*(z\_position[:-1] - origin[:-1]), color='black')

text\_size = 12

ax.text(\*x\_position[:-1], 'X', color='black', fontsize=text\_size)

ax.text(\*y\_position[:-1], 'Y', color='black', fontsize=text\_size)

ax.text(\*z\_position[:-1], 'Z', color='black', fontsize=text\_size)

ax.legend(loc='upper center')

def world\_to\_image(params):

points3D = [[Point3D(start), Point3D(end)] for start, end in POINTS]

camera = Camera()

camera.calc\_tau(height, width)

camera.set\_params(params)

points2D = [[camera.direct\_transform\_world(start), camera.direct\_transform\_world(end)] for start, end

in points3D]

\_points = [[start.get(), end.get()] for start, end in points2D]

\_points = np.array(\_points)

return \_points

def create\_dataset(params):

camera = Camera()

camera.calc\_tau(height, width)

camera.set\_params(params)

points\_dataset = [

[(camera.direct\_transform\_world(Point3D(start)), Point3D(start)),

(camera.direct\_transform\_world(Point3D(end)), Point3D(end))]

for start, end in POINTS]

return points\_dataset

POINTS = np.array([

[[-10, -20, 0, 1], [-10, 20, 0, 1]],

[[-5, -20, 0, 1], [-5, 20, 0, 1]],

[[5, -20, 0, 1], [5, 20, 0, 1]],

[[10, -20, 0, 1], [10, 20, 0, 1]],

[[-20, 10, 0, 1], [20, 10, 0, 1]],

[[-20, -10, 0, 1], [20, -10, 0, 1]],

[[-20, -5, 0, 1], [20, -5, 0, 1]],

[[-20, 5, 0, 1], [20, 5, 0, 1]],

]) \* 2

def plot\_lines\_world():

for start, end in POINTS:

plt.plot([start[0], end[0]], [start[1], end[1]], ls='--', color='black')

plt.show()

def plot\_lines\_image(params):

\_points = world\_to\_image(params)

for start, end in \_points:

plt.plot([start[0], end[0]], [start[1], end[1]], ls='--', color='black')

plt.xlim(0, width)

plt.ylim(0, height)

plt.grid()

# эталонные значения

height, width = 700, 1200

h = 40

angles = [-90, 20, -170]

f = 920

ax = init(h)

plot\_axies([0, 0, 0])

plot\_axies([0, 0, h], angles)

plot\_lines\_world()

plot\_lines\_image([f, \*angles, h])

plt.show()

camera = Camera()

camera.calc\_tau(height, width)

camera.set\_params([f, \*angles, h])

optimize = Optimizer(camera)

dataset = create\_dataset([f, \*angles, h])

camera, info, cost\_history, history = optimize.optimize\_reprojection(dataset)

print(np.around(info.x))

import matplotlib.pyplot as plt

# plt.plot(np.arange(0, len(cost\_history)), np.log(cost\_history))

plt.plot(np.arange(0, len(cost\_history)), cost\_history)

plt.ylabel('Точность')

plt.xlabel('Количество итераций')

plt.show()

**ПЛАН-ГРАФИК**

**выполнения курсовой работы**

обучающегося Акмурзина М.Э.

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование этапа работ | Трудоемкость выполнения, час. | Процент к общей трудоемкости выполнения | Срок предъявления консультанту |
| Получение и согласование задания | 0,3 | 0,8 | 4 неделя |
| Знакомство с литературой по теме курсовой работы | 2,7 | 7,5 | 8 неделя |
| Формирование модели камеры | 10 | 27,7 | 9 неделя |
| Реализация целевого функционала | 10 | 27,7 | 10 неделя |
| Проведение испытаний | 10 | 27,7 | 12 неделя |
| Составление и оформление пояснительной записки и подготовка к защите | 2,7 | 7,5 | 13 неделя |
| Защита | 0,3 | 0,8 | 14 неделя |
| Итого | 36 | 100 | - |