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

2.1 Изучение и реализация модели камеры обскуры для проекции мировых координат на плоскость изображения

2.2 Разработка и реализация метода восстановления параметров матрицы проекции, основанного на геометрических ограничениях сцены

2.3 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	Дата окончания работы
"__" _____ 202_ г.	"__" _____ 202_ г.

Консультант _____ Касаткин А.А.

СОДЕРЖАНИЕ

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

1. 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):

$$s\,p = A\,[R|t]P_w, \tag{1}$$

где P_w – трехмерная точка в мировой системе координат,
$p = [u, v, 1]^T$ – двумерный пиксель в плоскости изображения (используются однородные координаты),
A – внутренняя матрица камеры,
R и t – 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 A проецирует 3D-точки, заданные в системе координат камеры, в 2D-пиксельные координаты то есть (2):

$$p = AP_c s \tag{2}$$

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

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

Матрица внутренних параметров A 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 $[R|t]$ 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 $x' = X_c/Z_c$ и $y' = Y_c/Z_c$:

$$Z_c \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}. \quad (4)$$

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

$$P_c = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} P_w, \quad (5)$$

то есть матрица однородного преобразования состоит из $R \in \mathbb{R}^{3\times3}$ – матрицы вращения, и $t \in \mathbb{R}^{3\times1}$ – вектора переноса:

$$\begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \quad (6)$$

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

$$Z_c \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = [R|t] \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}, \quad (7)$$

где $x' = X_C/Z_c, y' = Y_c/Z_c$.

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

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}. \quad (8)$$

Если $Z_c \neq 0$, то (8) примет вид (9):

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \frac{f_x X_c}{Z_c} + c_c \\ \frac{f_y Y_c}{Z_c} + c_y \end{bmatrix}, \quad (9)$$

где $\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = [R|t] \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$.

На рисунке 1 показана модель камеры-обскуры.

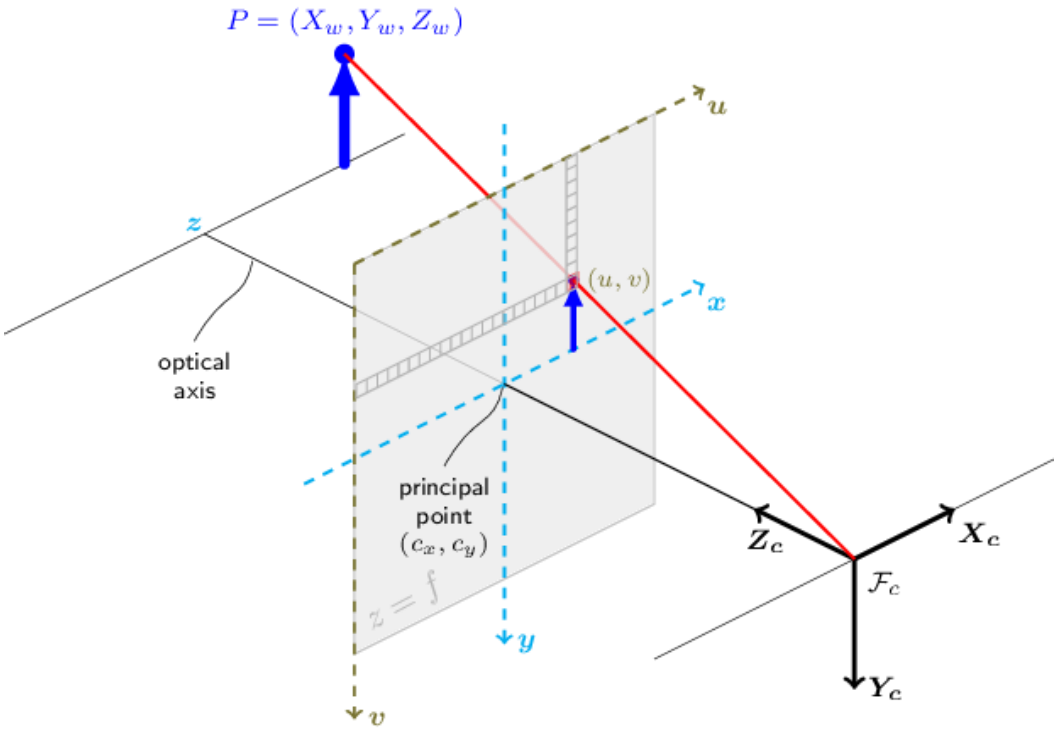

Рисунок 1 – Модель камеры-обскуры

2. Ограничения сцены

2.1. 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:

$$P_c = \begin{bmatrix} R & -Rt \\ 0 & 1 \end{bmatrix} P_w, \tag{10}$$

где P_c координаты относительно системы камеры, а P_w 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 X:

$$M_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix} \tag{12}$$

Матрица поворота вокруг оси Y:

$$M_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \tag{13}$$

Матрица поворота вокруг оси Z:

$$M_z(\gamma) = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{14}$$

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

$$R = M_z(\gamma)M_x(\beta)M_y(\alpha) \tag{15}$$

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

$$R = \begin{bmatrix} \cos(\gamma)\cos(\beta) - \sin(\gamma)\sin(\alpha)\sin(\beta) & -\sin(\gamma)\cos(\alpha) & \cos(\gamma)\sin(\beta) + \sin(\gamma)\sin(\alpha)\cos(\beta) \\ \sin(\gamma)\cos(\beta) + \cos(\gamma)\sin(\alpha)\sin(\beta) & \cos(\gamma)\cos(\alpha) & \sin(\gamma)\sin(\beta) - \cos(\gamma)\sin(\alpha)\cos(\beta) \\ -\cos(\alpha)\sin(\beta) & \sin(\alpha) & \cos(\alpha)\cos(\beta) \end{bmatrix} \tag{16}$$

Внутреннюю матрицу преобразования (3) перепишем:

$$A = \begin{bmatrix} f & 0 & \frac{W}{2} \\ 0 & f\,\tau & \frac{H}{2} \\ 0 & 0 & 1 \end{bmatrix}, \tag{17}$$

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

$$\tau = \frac{H}{W} \tag{18}$$

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

$$s\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & \frac{W}{2} \\ 0 & f\,\tau & \frac{H}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R & -Rt \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{19}$$

Выразим из (19) матрицу проекции:

$$P(Y) = \begin{bmatrix} f & 0 & \frac{W}{2} \\ 0 & f\,\tau & \frac{H}{2} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R & -Rt \\ 0 & 1 \end{bmatrix}, \tag{20}$$

, где $Y = [f, \gamma, \alpha, \beta, t_x, t_y, t_z]$ вектор параметров камеры соответственно фокусное расстояние, вращение вокруг координат z, x, y 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.

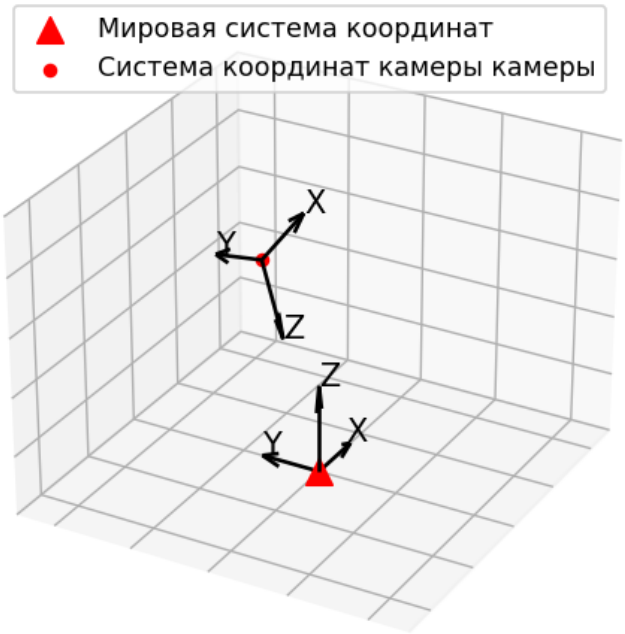

Рисунок 2 – Положение мировой системы координат и системы координат камеры

3. Начальное решение

3.1. 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.

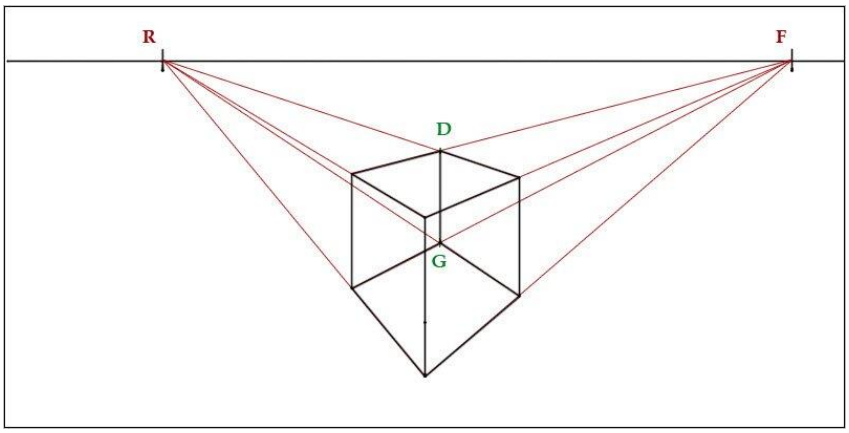

Рисунок 3 – Пример точек схода

Для вычисления точки схода согласно [1] воспользуемся минимальной суммой квадратов расстояния до всех параллельных прямых, проходящих через нее. То есть пусть $u_i \in R^2$ будет единичной нормалью к прямой L_i, которая проходит через конечные точки $a_i, b_i \in R^2$. Для точки схода p ортогональное расстояние до прямой L_i равно:

$$\sum_{i=1}^{n} (u_i p - u_i a_i)^2 , \qquad (21)$$

где $n \in N$ количество параллельных прямых.

Минимизация суммы (21) эквивалентная решению линейной системы

$$Ap = r \tag{22}$$

где $A = [u_1, u_2, \dots, u_n]^T, r = [u_1 a_1, u_2 a_2, \dots, u_n a_n]^T$.

3.2.Начальное решение

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

$$f = \sqrt{|(v_y - D)(v_x - D)|}, \tag{23}$$

где $D = [u_0, v_0]^T$ это сдвиг центральной точки.

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

$$p_y = A^{-1}\left[u_y^T, 1\right]^T, p_x = A^{-1}[u_x^T, 1]^T \tag{24}$$

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

$$p_z = p_x \times p_y \tag{25}$$

Сформируем из p_y, p_x, p_z матрицу поворота в соответствие по [2]:

$$R = \left[p_x^T, p_y^T, p_z^T\right]^T \tag{26}$$

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

$$\begin{gathered} \alpha = \arcsin(-R_{23}), \\ \beta = \arctan2(R_{21}, R_{11}), \\ \gamma = \arctan2(R_{31}, R_{33}), \end{gathered} \tag{27}$$

где α вращения вокруг OX, β вращения вокруг OY, γ вращения вокруг OZ.

Начальный вектор переноса, следующий:

$$t = [1,1,10]^T \tag{28}$$

4. Функция оптимизация

Рассмотрим набор отрезков L_i, где $i = \overline{1, N}$, такой что $L_i = (x_i', X_i', x_i'', X_i'')$, где $x_i' \in R^3, X_i' \in R^4$ ссоответствуют началу отрезка, $x_i'' \in R^3, X_i'' \in R^4$ концу отрезка. Координаты x_i', x_i'' являются однородными в системе координат камеры, X_i', X_i'' однородные в мировой системе координат.

Основной целью является нахождение оптимальных параметров камеры

$$Y = \left[f, \gamma, \alpha, \beta, t_x, t_y, t_z\right], \tag{29}$$

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:

$$f_1(Y, L_i) = \left||x_i' - PX_i'|\right| + \left||x_i'' - PX_i''|\right| \quad (30)$$

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

$$f_2(Y, L_i) = |\arctan2(x_i', x_i'') - \arctan2(PX_i', PX_i'')\,| \quad (31)$$

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

$$F(Y) = \sum_{i=1}^{N} C_1 f_1(Y, L_i) + C_2 f_2(Y, L_i)\,, \quad (32)$$

где C_1, C_2 – весовые коэффициенты.

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

ПРАКТИЧЕСКАЯ ЧАСТЬ

5. Демонстрация на синтетических данных

Проведём испытания на синтетических данных. Для этого сформируем сцену, содержащую прямые отрезки в плоскости $Z = 0$. 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.

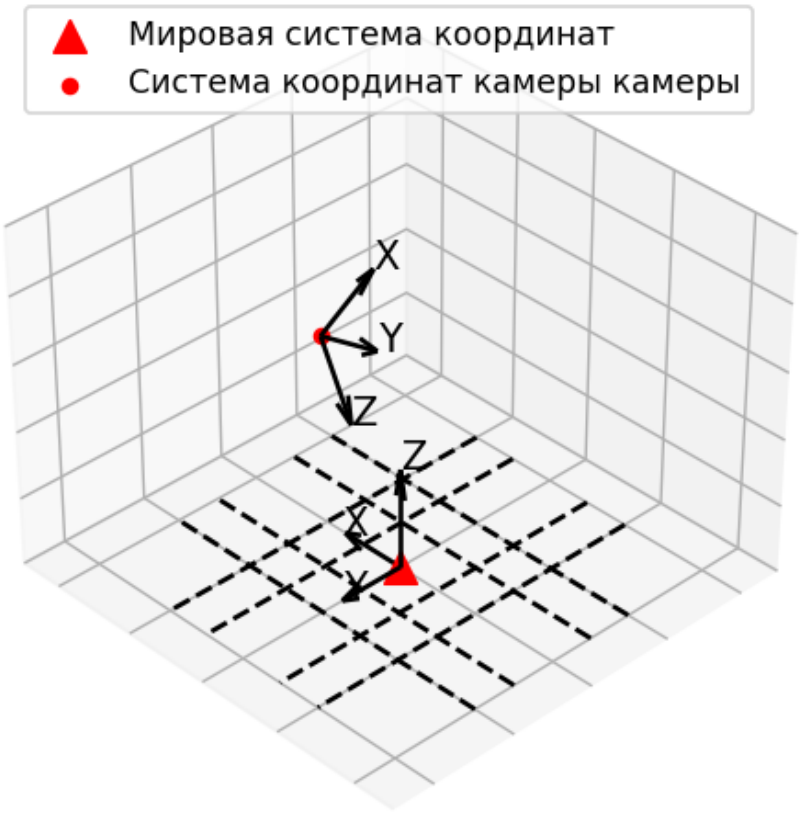

Рисунок 4 – Сгенерированная сцена с прямыми отрезками, расположенными в плоскости $Z = 0$, и соответствующими системами координат (мировая и камера).

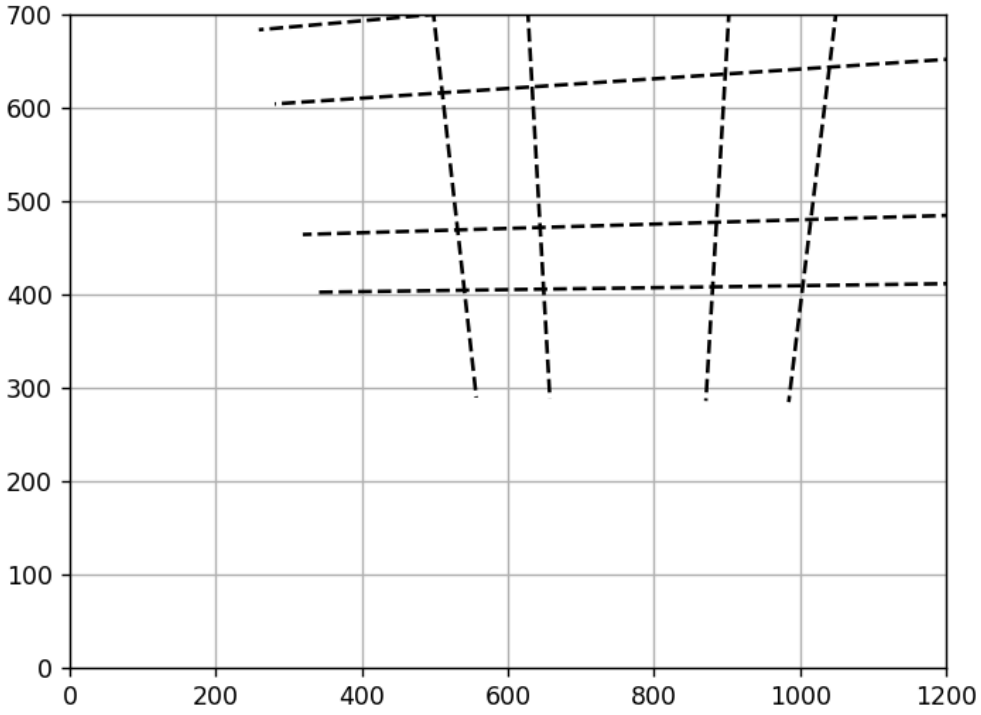

Рисунок 5 – Спроецированные прямые отрезки на плоскость изображения, полученные с использованием матрицы проекции камеры на основе синтетической сцены.

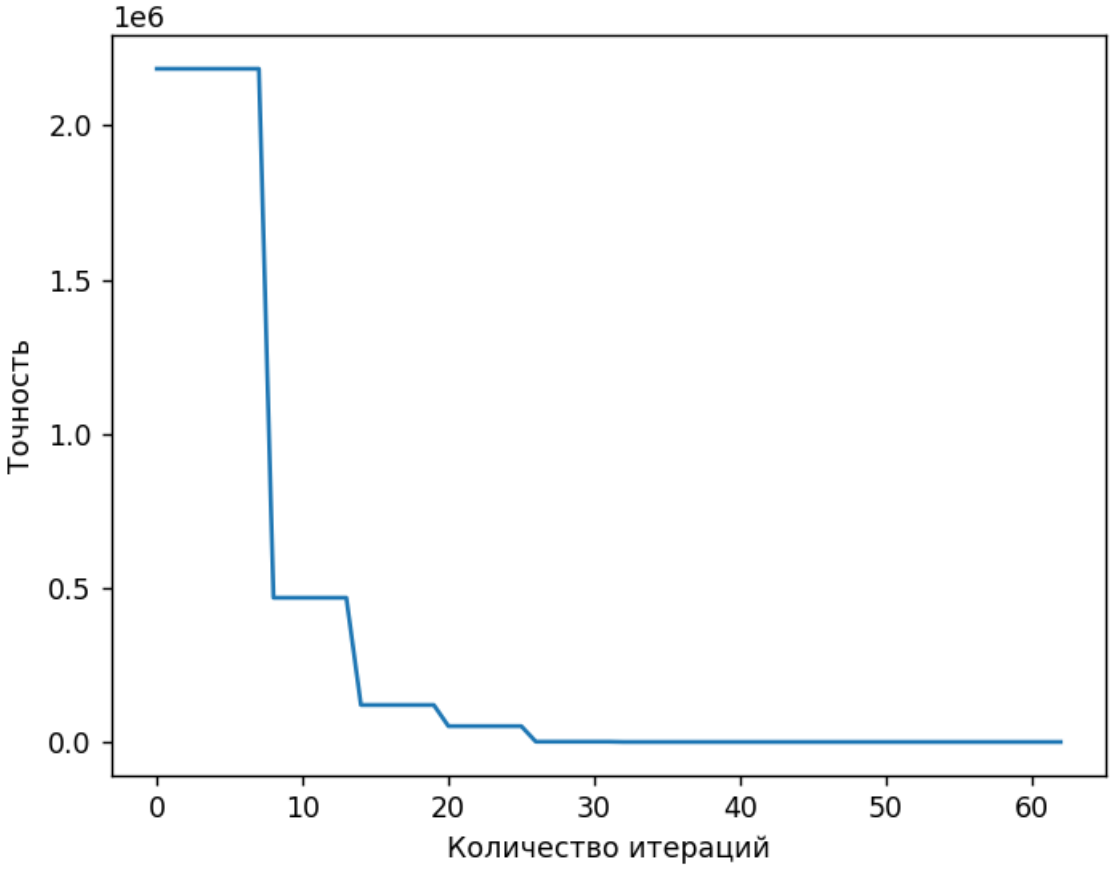

Рисунок 6 – График сходимости функции оптимизации

6. 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 $Z = 0$. 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.

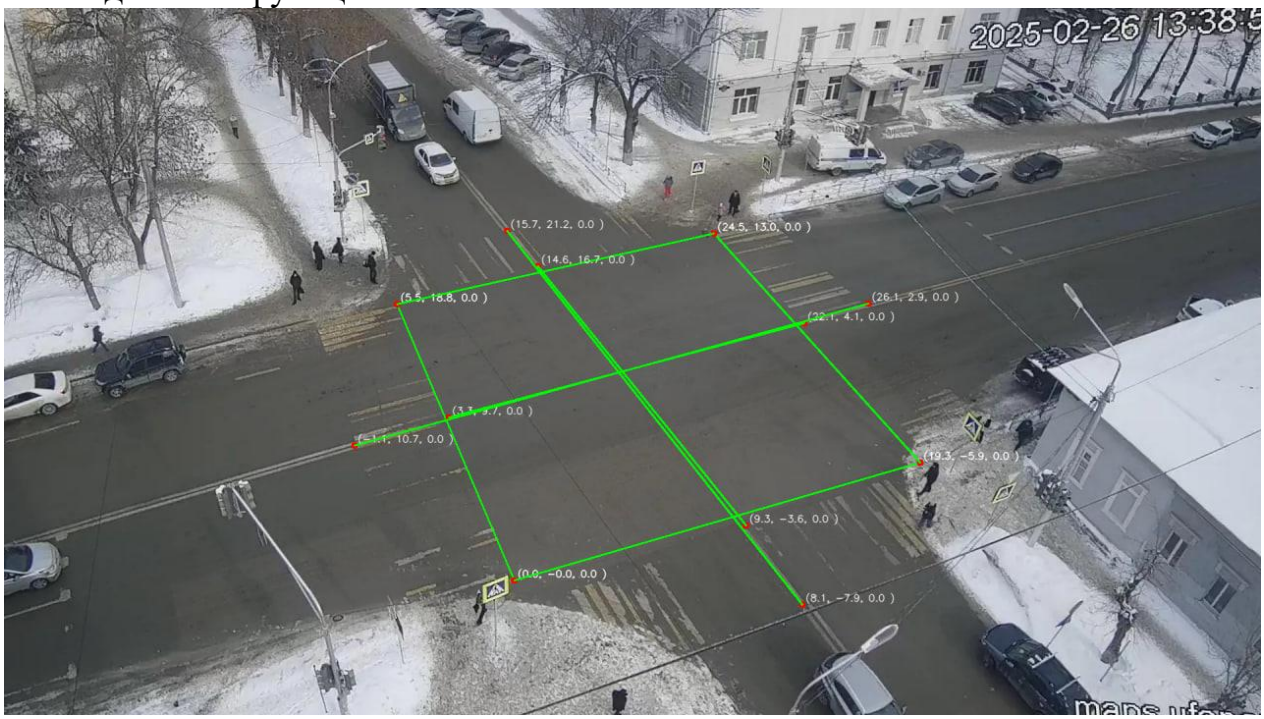

Рисунок 7 – Размеченные отрезки на перекрёстке улиц Пушкина и Аксакова, полученные с использованием камеры видеонаблюдения

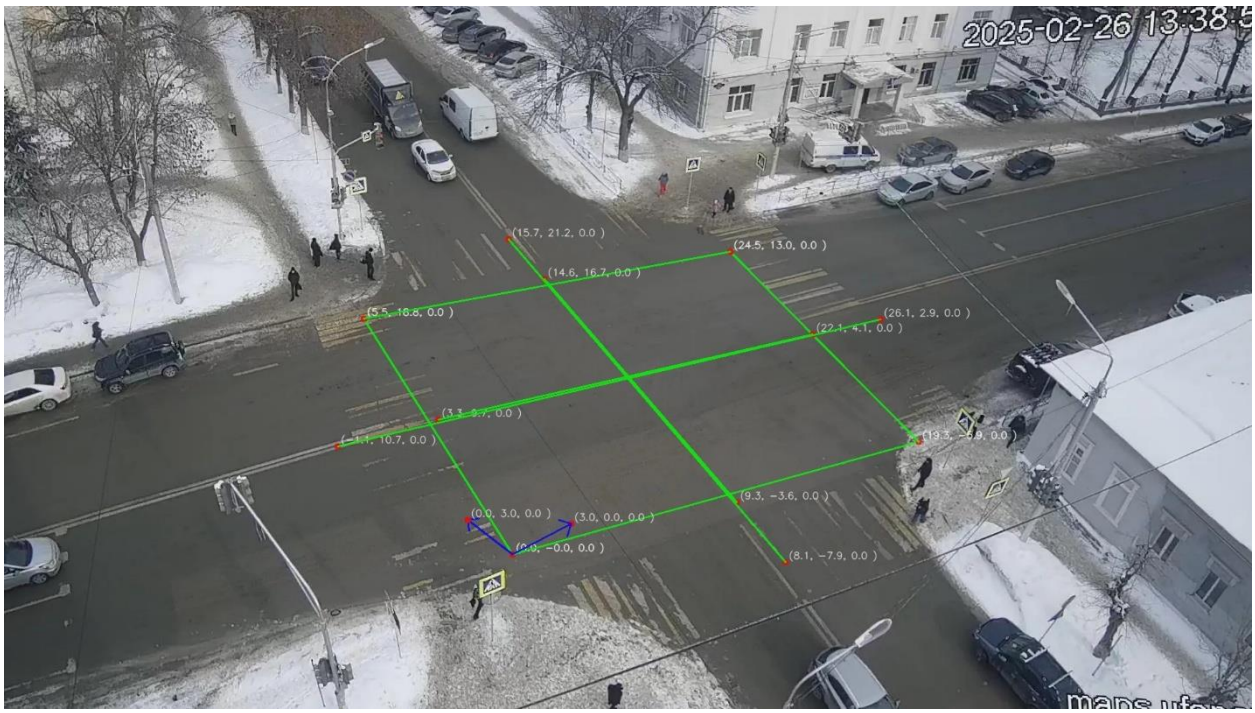

Рисунок 8 – Спроецированные прямые отрезки на перекрёстке улиц Пушкина и Аксакова, отображённые на плоскости изображения с учётом калибровки камеры и мировой системы координат.

Рисунок 9 – Спроецированные прямые линии и размеченные линии

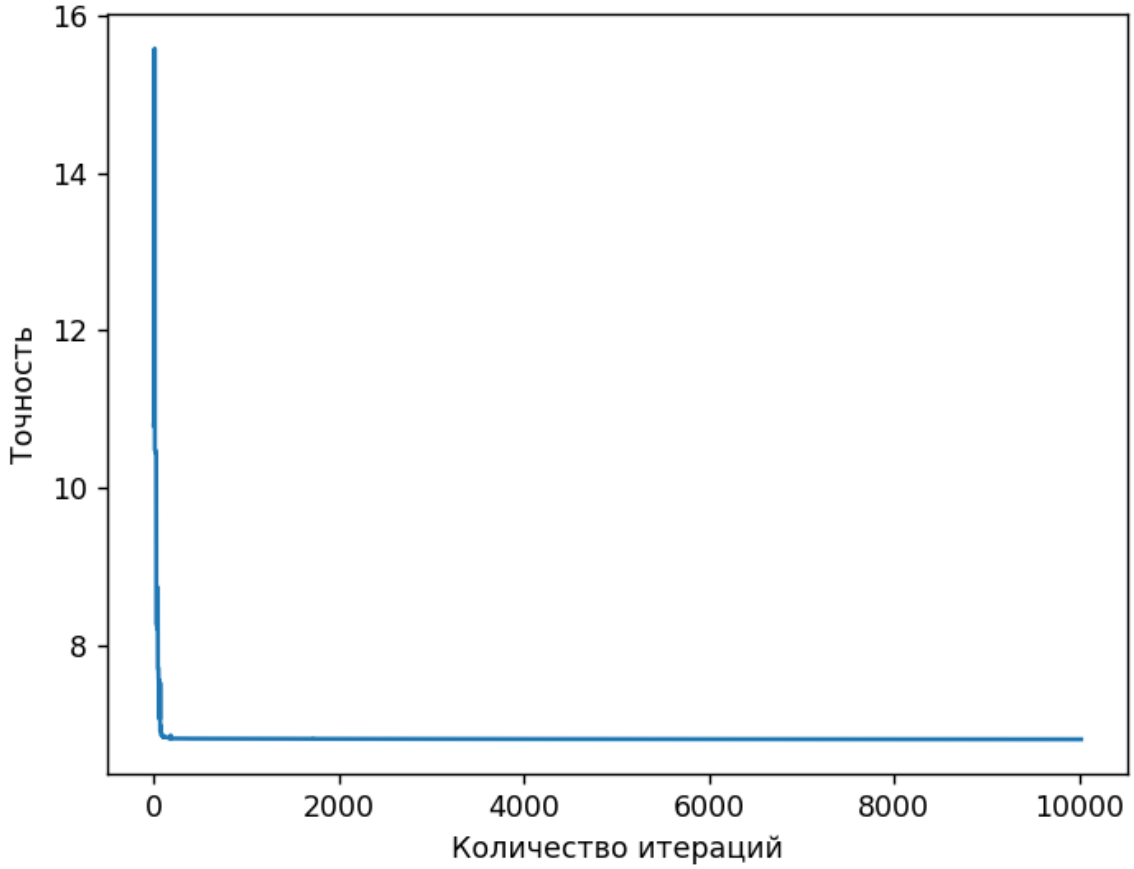

Рисунок 10 – График сходимости функции оптимизации

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	-