Оглавление

[**Описание метода** 2](#_Toc52411839)

[**Описание программного кода** 5](#_Toc52411840)

[**Шаг 1. Считывание изображений с файла** 5](#_Toc52411841)

[**Шаг 2. Поиск особых точек** 6](#_Toc52411842)

[**Шаг 3. Поиск оптического потока методом Лукаса-Канаде** 8](#_Toc52411843)

[**Шаг 4. Отрисовка результатов** 10](#_Toc52411844)

[**5. Описание вспомогательных функций** 11](#_Toc52411845)

[**5.1** **Создание результирующего изображения** 11](#_Toc52411846)

[**5.2** **Отображение изображение с помощью MatPlotLib** 14](#_Toc52411847)

[**5.1** **Вычисление смещения** 16](#_Toc52411848)

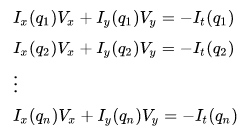
[**Результаты** 17](#_Toc52411849)

[**Исходный код** 19](#_Toc52411850)

# **Описание метода**

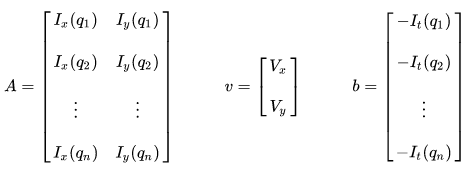
В компьютерном зрении метод **Лукаса – Канаде** - широко используемый дифференциальный метод оценки оптического потока, разработанный Брюсом Д. Лукасом и Такео Канаде. Он предполагает, что поток по существу постоянен в локальной окрестности рассматриваемого пикселя, и решает основные уравнения оптического потока для всех пикселей в этой окрестности по критерию наименьших квадратов.

Метод Лукаса – Канаде предполагает, что смещение содержимого изображения между двумя соседними моментами (кадрами) мало и приблизительно постоянно в пределах окрестности рассматриваемой точки *p*. Таким образом, можно предположить, что уравнение оптического потока выполняется для всех пикселей в окне с центром в точке *p*. А именно, вектор потока (скорости) локального изображения должен удовлетворять



где - пиксели внутри окна, а - частные производные изображения по позиции *x*, *y* и времени *t* , вычисленные в точке и в текущий момент времени. 

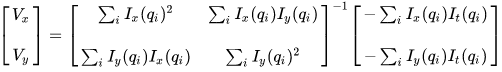
Эти уравнения можно записать в матричной форме, где 



В этой системе больше уравнений, чем неизвестных, и поэтому она обычно переоценивается. Метод Лукаса – Канаде дает компромиссное решение по принципу наименьших квадратов. А именно, он решает систему 2 × 2



где является транспонированной матрицы. То есть вычисляет 



где центральная матрица в уравнении является обратной матрицей. Суммы идут от *i* = 1 до *n*.

Матрицу часто называют тензором структуры изображения в точке *p* 

Методы вычисления оптического потока в настоящее время становятся все популярнее, а области его применения с каждым годом расширяются (сжатие видеофайлов, создание видео со спецэффектами, компьютерное зрение и т. д). Можно выделить основные применения оптического потока: определение направления движения объектов; плавная трансформация между кадрами, снятыми последовательно либо соседними камерами в киноиндустрии; определение расстояния между объектами с помощью анализа оптического потока кадров, полученных с двух камер (стереозрение).

Исследования оптического потока ведутся в областях сжатия видеофайлов и анализа движений. Алгоритмы оптического потока не только определяют поле потока, но и используют поток при анализе трехмерной сущности и структуры сцены, а также трехмерного движения объектов и наблюдателя относительно сцены. Оптический поток применяют в робототехнике при распознавании объектов и слежении за ними, определении движения и при навигации робота, для изучения структуры объектов. Поскольку определение движения и создание карт структуры окружающей среды являются неотъемлемой частью животного (человеческого) зрения, то реализация этой врожденной способности средствами компьютера является неотъемлемой частью компьютерного зрения.

# **Описание программного кода**

## **Шаг 1. Считывание изображений с файла**

image1\_name = **'images/1.jpg'**image2\_name = **'images/2.jpg'**img1, cap1 = loadImage(image1\_name)  
img2, cap2 = loadImage(image2\_name)

*# переводим изображение в нужный для работы методов формат*ret, old\_frame = cap1.read()

*# и переводим изображение в grayscale (оттенки серого)*old\_gray = cv2.cvtColor(old\_frame, cv2.COLOR\_BGR2GRAY)

## **Шаг 2. Поиск особых точек**

*# находит N сильнейших углов на изображении*p0 = cv2.goodFeaturesToTrack(old\_gray, mask=None, \*\*feature\_params)

goodFeaturesToTrack –метод находит maxCorners сильнейших углов изображения с помощью метода Ши-Томази(или Детектора Харриса). Передаваемое изображение должно быть в оттенках серого.

* old\_gray – изображение на котором будет производиться поиск
* mask – необязательный параметр, интересующий регион для поиска.
* feature\_params – параметры поиска

*# параметры углового детектора Ши-Томаси, который  
# ищет особые точки по которым далее будет рассчитываться оптичискей поток Лукаса-Канаде*feature\_params = dict(maxCorners=100,  
 qualityLevel=0.3,  
 minDistance=7,  
 blockSize=7)

* maxCorners – Максимальное количество углов которое получив в результате работы детектора. Если углов будет больше, то из списка всех углов выберутся maxCorners сильнейших.
* qualityLevel – уровень качества, который представляет собой значение от 0 до 1, он обозначает минимальное качество угла, ниже которого угол отбрасывается
* minDistance – минимальное Евклидово расстояние между обнаруженными углами
* blockSize – размер среднего блока для вычисления производного матрица ковариации по каждой окрестности пикселя.

**Шаг 3. Поиск оптического потока методом Лукаса-Канаде**

С помощью алгоритма Лукаса-Канаде вычисляется смещение объектов на изображении.

*# Расчитываем оптический поток методом Лукаса-Канаде*p1, st, err = cv2.calcOpticalFlowPyrLK(prevImg, nextImg, prevPts, nextPts, \*\*lk\_params)

**Input params:**

* prevImg – исходное изображение
* nextImg – второе изображение, того же размера и того же типа что и первое
* prevPts – вектор 2D точек, для которых необходимо найти поток; координаты точки должны быть числами с плавающей запятой одинарной точности.
* nextPts **–** выходной вектор двумерных точек, содержащий вычисленные новые положения входных объектов во втором изображении.
* lk\_params – параметры поиска

**Return params:**

* p1 – координаты смещенных точек на втором изображении
* st – вектор состояния. Каждый элемент вектора устанавливается в 1 если поток для соответствующей точки был найден, иначе в 0.
* Err – выходной вектор ошибок. Каждый элемент вектора устанавливается на ошибку для соответствующего признака.

*# Параметры алгоритма Лукааса - Канаде*lk\_params = dict(winSize=(15, 15),  
 maxLevel=2,  
 criteria=(cv2.TERM\_CRITERIA\_EPS | cv2.TERM\_CRITERIA\_COUNT, 10, 0.03))

* winSize – размер окна поиска на каждом уровне пирамиды
* maxLevel – максимальное количество пирамид. Если 0, то пирамиды не строятся, используется только исходное изображение.
* criteria – критерий завершения алгоритма.
  1. TERM\_CRITERIA\_EPS – когда окно поиска перемещается меньше чем на эпсилон.
  2. TERM\_CRITERIA\_COUNT – когда достигнуто максимальное количество итераций

## **Шаг 4. Отрисовка результатов**

*# Выбираем координаты особых точек на первом изображении*good\_old = p0[st == 1]  
  
*# Выбираем координаты особых точек на втором изображении*good\_new = p1[st == 1]  
  
Далее, в цикле, смотрим для каких точек исходного изображения были найдены точки на втором изображении без ошибок, запоминая пары координат для отрисовки результата:

*# отображаем смещение изображения*draw\_points = [] *# точки по которым рисуем*for i, (new, old) in enumerate(zip(good\_new, good\_old)):  
 x1, y1 = new.ravel()  
 x2, y2 = old.ravel()  
 draw\_points.append([[x1, y1], [x2, y2]])

Вызываем функцию отрисовки результата и подсчета смещения.

*#отрисовываем и сохраняем совмещенное итоговое изображение*createImages(img1, img2, draw\_points, **"results/1\_2.jpg"**)

**5. Описание вспомогательных функций**

* 1. **Создание результирующего изображения**

**Параметры функции:**

def createImages(img1, img2, coordinates, resultPathName)

* Img1- первое изображение
* Img2 – второе изображение
* Coordinates – координаты соответствующих точек на первом и втором изображениях
* resultPathName – имя файла, куда будет выведен результат ( в формате jpg)

**Код функции:**

*# отрисовываем и сохраняем совмещенное итоговое изображение*def createImages(img1, img2, coordinates, resultPathName):  
 key\_points\_1 = []  
 key\_points\_2 = []  
 allignment\_points = []  
  
 i = 0

for coor in coordinates:  
 x1 = coor[0][0]  
 y1 = coor[0][1]  
 x2 = coor[1][0]  
 y2 = coor[1][1]  
 key\_points\_1.append(cv2.KeyPoint(x1, y1, 0))  
 key\_points\_2.append(cv2.KeyPoint(x2, y2, 0))  
 allignment\_points.append([cv2.DMatch(i, i, 0, 57)])  
 i += 1  
  
 *# рисуем результат, совмещаем точки первого и второго изображения* img\_result = cv2.drawMatchesKnn(img1, key\_points\_1, img2, key\_points\_2, allignment\_points, None,  
 flags=cv2.DrawMatchesFlags\_NOT\_DRAW\_SINGLE\_POINTS)  
  
 *# считаем смещение* bias\_x\_middle, bias\_y\_middle = calculate\_bias(coordinates)  
  
 *# заголовок результирующего изображения* title = **"X: "** + str(round(bias\_x\_middle, 5)) + \  
 **" Y: "** + str(round(bias\_y\_middle, 5))  
  
 *# рисуем и сохраняем результат* plot\_create\_image(img\_result, title, resultPathName)

**Используемые в функции объекты и другие функции:**

* KeyPoint – объект opencv, Структура данных для детекторов характерных точек.
* DMatch – класс для сопоставления дескрипторов ключевых точек.
* Отрисовка совпадающих точек на двух изображениях

outImg = cv2.drawMatchesKnn(img1, keypoints1, img2, keypoints2, matches1to2, outImg[, matchColor[, singlePointColor[, matchesMask[, flags]]]])

**Параметры функции**

|  |  |
| --- | --- |
| **img1** | Первое изображение. |
| **keypoints1** | Точки с первого изображения |
| **img2** | Второе изображение |
| **keypoints2** | Точки второго изображение |
| **matches1to2** | Соответствие от первого изображения ко второму. keypoints1[i] соответствует точке в keypoints2[matches[i]] . |
| **outImg** | Результирующее изображение. |
| **matchColor** | Цвета линий соединяющих особые точки первого и второго изображения. |
| **singlePointColor** | Цвета особых точек изображения |
| **matchesMask** | Маска, определяющая, какие совпадения отрисовывать. Если маска пуста, отображаются все совпадения. |

**5.2 Отображение изображение с помощью MatPlotLib**

*# рисуем и сохраняем результат* plot\_create\_image(img\_result, title, resultPathName)

**Параметры функции:**

* img\_result - полученное изображение на прошлых шагах
* title – заголовок изображения, где отображаем смещение по координатам
* resultPathName – имя сохраняемого файла

**Код функции:**

*# рисуем результирующее изображение, на котором отмечаем смещение*def plot\_create\_image(data, title, fileName):  
 fig, ax1 = plt.subplots(1, 1)  
  
 im = ax1.imshow(data, interpolation=**'nearest'**)  
  
 fg\_color = **'black'** bg\_color = **'white'** *# IMSHOW  
 # устанавилваем цвет заголовка* ax1.set\_title(title, color=fg\_color)  
  
 ax1.patch.set\_facecolor(bg\_color)  
  
 *# установить цвет меток* im.axes.tick\_params(color=fg\_color, labelcolor=fg\_color)  
  
 *# set imshow outline* for spine in im.axes.spines.values():  
 spine.set\_edgecolor(fg\_color)  
  
 fig.patch.set\_facecolor(bg\_color)  
 plt.tight\_layout()  
 plt.savefig(fileName, dpi=300)  
  
 plt.show()  
 *# plt.savefig('save/to/pic.png', dpi=200, facecolor=bg\_color)*

* 1. **Вычисление смещения**

*# считаем среднее смещение*def calculate\_bias(coordinates):

**Параметры функции:**

* coordinates – координаты точек на первом и втором изображениях

**Код функции:**

def calculate\_bias(coordinates):  
 bias = []  
  
 i = 0  
 for coor in coordinates:  
 x1 = coor[0][0]  
 y1 = coor[0][1]  
 x2 = coor[1][0]  
 y2 = coor[1][1]  
 bias.append([x2 - x1, y2 - y1])  
  
 bias\_x\_avg, bias\_y\_avg = calculate\_avg(bias)  
  
 return bias\_x\_avg, bias\_y\_avg

**Результаты**



Рис 1 - Изображение 1



Рис 2 - Изображение 2

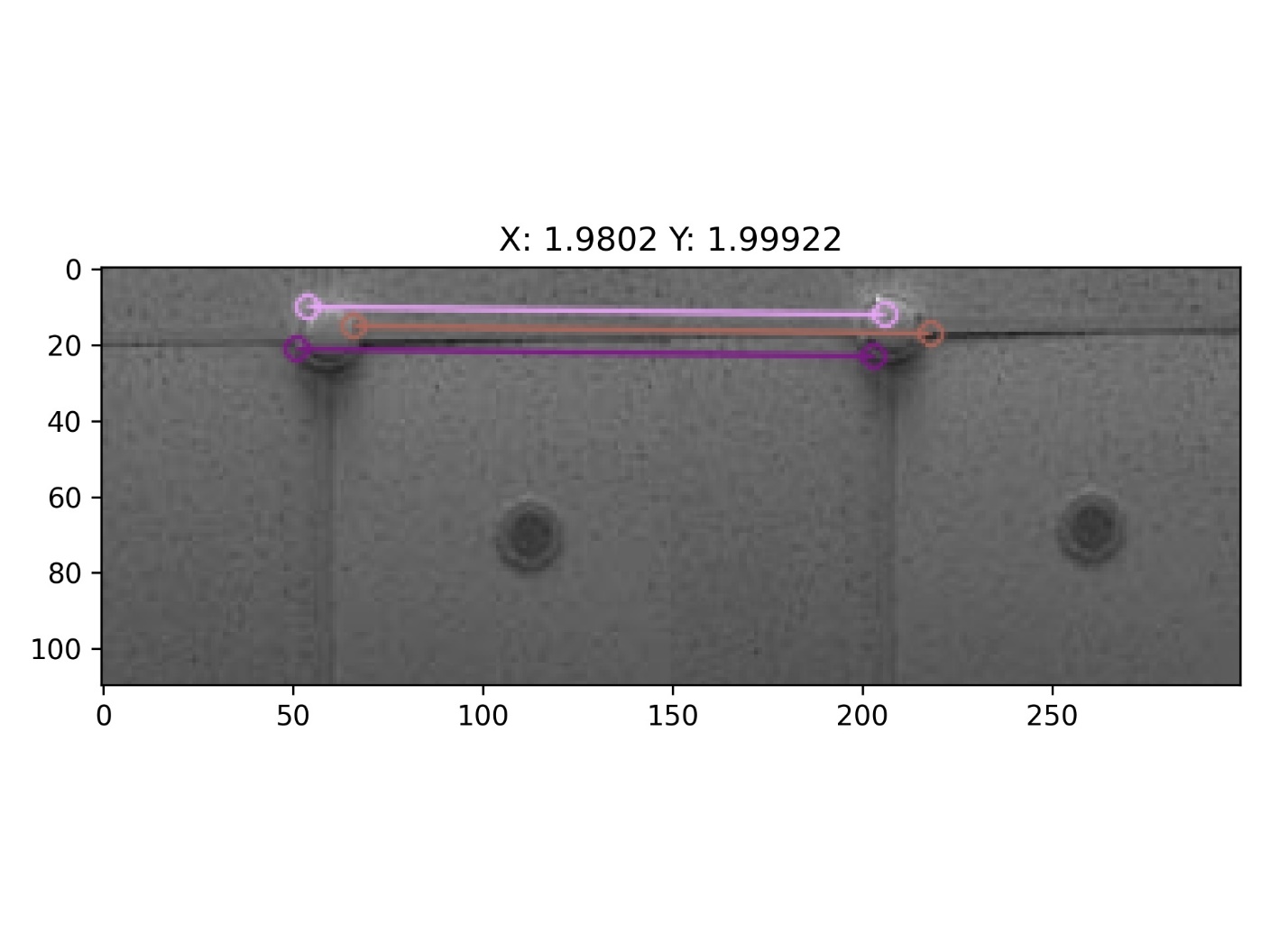


Рис 3 - Результат

**Исходный код**

import numpy as np  
import cv2  
import matplotlib.pyplot as plt  
  
*# Считаем среднее смещение*def calculate\_avg(bias):  
 bias\_x\_avg = 0  
 bias\_y\_avg = 0  
  
 for b in bias:  
 bias\_x\_avg += b[0]  
 bias\_y\_avg += b[1]  
  
 bias\_x\_avg /= len(bias)  
 bias\_y\_avg /= len(bias)  
 return bias\_x\_avg, bias\_y\_avg  
  
  
*# считаем среднее смещение*def calculate\_bias(coordinates):  
 bias = []  
  
 i = 0  
 for coor in coordinates:  
 x1 = coor[0][0]  
 y1 = coor[0][1]  
 x2 = coor[1][0]  
 y2 = coor[1][1]  
 bias.append([x2 - x1, y2 - y1])  
  
 bias\_x\_avg, bias\_y\_avg = calculate\_avg(bias)  
  
 return bias\_x\_avg, bias\_y\_avg  
  
*# рисуем результирующее изображение, на котором отмечаем смещение*def plot\_create\_image(data, title, fileName):  
 fig, ax1 = plt.subplots(1, 1)  
  
 im = ax1.imshow(data, interpolation=**'nearest'**)  
  
 fg\_color = **'black'** bg\_color = **'white'** *# IMSHOW  
 # устанавилваем цвет заголовка* ax1.set\_title(title, color=fg\_color)  
  
 ax1.patch.set\_facecolor(bg\_color)  
  
 *# установить цвет меток* im.axes.tick\_params(color=fg\_color, labelcolor=fg\_color)  
  
 *# set imshow outline* for spine in im.axes.spines.values():  
 spine.set\_edgecolor(fg\_color)  
  
 fig.patch.set\_facecolor(bg\_color)  
 plt.tight\_layout()  
 plt.savefig(fileName, dpi=300)  
  
 plt.show()  
 *# plt.savefig('save/to/pic.png', dpi=200, facecolor=bg\_color)  
  
  
# отрисовываем и сохраняем совмещенное итоговое изображение*def createImages(img1, img2, coordinates, resultPathName):  
 key\_points\_1 = []  
 key\_points\_2 = []  
 allignment\_points = []  
  
 i = 0  
 for coor in coordinates:  
 x1 = coor[0][0]  
 y1 = coor[0][1]  
 x2 = coor[1][0]  
 y2 = coor[1][1]  
 key\_points\_1.append(cv2.KeyPoint(x1, y1, 0))  
 key\_points\_2.append(cv2.KeyPoint(x2, y2, 0))  
 allignment\_points.append([cv2.DMatch(i, i, 0, 57)])  
 i += 1  
  
 *# рисуем результат, совмещаем точки первого и второго изображения* img\_result = cv2.drawMatchesKnn(img1, key\_points\_1, img2, key\_points\_2, allignment\_points, None,  
 flags=cv2.DrawMatchesFlags\_NOT\_DRAW\_SINGLE\_POINTS)  
  
 *# считаем смещение* bias\_x\_middle, bias\_y\_middle = calculate\_bias(coordinates)  
  
 *# заголовок результирующего изображения* title = **"X: "** + str(round(bias\_x\_middle, 5)) + \  
 **" Y: "** + str(round(bias\_y\_middle, 5))  
  
 *# рисуем и сохраняем результат* plot\_create\_image(img\_result, title, resultPathName)  
  
def loadImage(image\_name):  
 img1 = cv2.imread(image\_name, cv2.IMREAD\_GRAYSCALE) *# first image* cap1 = cv2.VideoCapture(image\_name)  
 return img1, cap1  
  
  
def main():  
  
 image1\_name = **'images/1.jpg'** image2\_name = **'images/2.jpg'** img1, cap1 = loadImage(image1\_name)  
 img2, cap2 = loadImage(image2\_name)  
  
  
 *# считываем первого изображение* ret, old\_frame = cap1.read()  
 *# переводим изображение в грейСкейл (черно-белое)* old\_gray = cv2.cvtColor(old\_frame, cv2.COLOR\_BGR2GRAY)  
  
 *# считываем второе изображение* ret, frame = cap2.read()  
 *# переводим изображение в грейСкейл (черно-белое)* frame\_gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)  
  
 *# параметры углового детектора Ши-Томаси, который  
 # ищет особые точки по которым далее будет рассчитываться оптичискей поток Лукаса-Канаде* feature\_params = dict(maxCorners=100,  
 qualityLevel=0.3,  
 minDistance=7,  
 blockSize=7)  
 *# находит N сильнейших углов на изображении* p0 = cv2.goodFeaturesToTrack(old\_gray, mask=None, \*\*feature\_params)  
  
  
 *# Параметры алгоритма Лукааса - Канаде* lk\_params = dict(winSize=(15, 15),  
 maxLevel=2,  
 criteria=(cv2.TERM\_CRITERIA\_EPS | cv2.TERM\_CRITERIA\_COUNT, 10, 0.03))  
 *# Расчитываем оптический поток методом Лукаса-Канаде* p1, st, err = cv2.calcOpticalFlowPyrLK(old\_gray, frame\_gray, p0, None, \*\*lk\_params)  
  
 *# Выбираем координаты хороших точек на первом изображении* good\_old = p0[st == 1]  
  
 *# Выбираем координаты хороших точек на втором изображении* good\_new = p1[st == 1]  
  
 *# отображаем смещение изображения* draw\_points = [] *# точки по которым рисуем* for i, (new, old) in enumerate(zip(good\_new, good\_old)):  
 x1, y1 = new.ravel()  
 x2, y2 = old.ravel()  
 draw\_points.append([[x1, y1], [x2, y2]])  
 print(**'('** + str(x1) + **' ; '** + str(y1) + **')'** + **' - '** + **'('** + str(x2) + **' ; '** + str(y2) + **')'**)  
  
 *#отрисовываем и сохраняем совмещенное итоговое изображение* createImages(img1, img2, draw\_points, **"results/s1\_2.jpg"**)  
  
  
  
if \_\_name\_\_ == **"\_\_main\_\_"**:  
 main()