Федеральное государственное автономное образовательное учреждение высшего образования

«Национальный исследовательский университет ИТМО»

Отчет по лабораторной работе №2
«Геометрические преобразования изображений»
по дисциплине «Техническое зрение»

Выполнил: студент гр. R3338,
Кирбаба Д.Д.
Преподаватель: Шаветов С.В.,
канд. техн. наук, доцент ФСУ и Р

Санкт-Петербург, 2022

Цель работы

Освоение основных видов отображений и использование геометрических преобразований для решения задач пространственной коррекции изображений.

Теоретическое обоснование применяемых методов

В прошлой лабораторной работе были изучены различные яркостные преобразования изображений, а в этой будем работать с геометрическими преобразованиями, то есть с координатами пикселей (x, y). Для этого необходимо использовать однородные координаты (обладают свойством, что точка остается неизменно при умножении на ненулевые число), запись точки (x, y) в которой будет выглядеть как (x', y', ω), где ω – скалярный произвольный множитель: $x = \frac{x'}{\omega}, y = \frac{y'}{\omega}$.

Таким образом, любое линейное преобразование изображения можно описать с помощью матричного соотношения: $\begin{bmatrix} x' \\ y' \\ \omega' \end{bmatrix} = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \begin{bmatrix} x \\ y \\ \omega \end{bmatrix}$.

Первая группа изучаемых линейных преобразований — это конформные отображения. При данных отображениях сохраняются формы бесконечно малых фигур и углы между кривыми в точках их пересечения. Сдвиг, отражение, однородное масштабирование и поворот – преобразования, относящиеся к этой группе.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & C \\ 0 & 1 & F \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} - \text{сдвиг на } \mathrm{C}(x), F(y).$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} - \text{отражение относительно } oX.$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} - \text{однородное масштабирование}, \quad \alpha > 0.$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\alpha & sin\alpha & 0 \\ -sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} - \text{поворот на } \alpha \text{ вокруг точки } (0, 0).$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\alpha & sin\alpha & 0 \\ -sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} - \text{поворот на } \alpha \text{ вокруг точки } (a, b).$$

Рассмотрим также аффинные преобразования – это отображения, при которых параллельные, пересекающиеся и скрещивающиеся прямые переходят соответственно в параллельные, пересекающиеся и скрещивающиеся, сохраняются отношения площадей фигур и длин отрезков, лежащих на одной или параллельных прямых.

Основными элементами группы аффинных преобразований являются конформные отображения, а также скос и неоднородное масштабирование.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ s & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} - \text{скос.}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} - \text{неоднородное масштабирование}, \qquad \alpha, \beta > 0, \alpha \neq \beta.$$

Кусочно-линейные отображения – отображения, при которых сначала происходит разбиение на отдельные части изображения, а затем к разным областям применяются различные линейные преобразования.

Стоит отметить, что при формировании изображения неизбежно появляются различные нелинейные искажения и естественно, что для их устранения необходимы уже нелинейные преобразования геометрических характеристик картинки.

Самой обширной группой рассматриваемых в этой работе преобразований является проекционные отображения. Это такие отображения, при которых прямые линии остаются прямыми линиями, однако параллельность линий может быть нарушена.

При проективном отображении точки трехмерной сцены на двумерную плоскость, данное отображение не является линейным в общем случае:

$$\begin{cases} x' = \dfrac{Ax + By + C}{Gx + Hy + I} \\ y' = \dfrac{Dx + Ey + F}{Gx + Hy + I} \end{cases} \Rightarrow T = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & 1 \end{bmatrix}$$

Также нелинейные преобразования можно описывать с помощью полиномов определенных степеней. Например, в случае полиномиального преобразования первой степени получится следующая система уравнений:

$$\begin{cases} x' = a_1 x + a_2 y + a_3 xy + a_4 \\ y' = b_1 x + b_2 y + b_3 xy + b_4 \end{cases}$$

В случае, когда не известны коэффициенты системы $\{a_1, a_2, \dots, b_4\}$ и доступны исходное и преобразованное изображения, можно найти соответствующие одинаковые точки на обоих изображениях ($t_{min} = \frac{(n+1)(n+2)}{2}$ – минимальное количество точек для решения система), составить систему уравнений и найти коэффициенты.

Еще одно нелинейное преобразование – синусоидальное:

$$\begin{cases} x' = x + \text{asin}(by) \\ y' = y \end{cases}$$

Стоит отметить, то при различных преобразованиях появляются так называемые неопределенные пиксели (пиксели, значения которых не попали в новую вычисленную сетку), в таких случаях можно пользоваться либо прямым (присваивание неопределенным пикселям яркости ближайших соседей), либо обратным (обратное преобразование и также присваивание яркости ближайших соседей).

Часто при формировании изображения может возникать искажение – дисторсия. Это зависит из-за свойств используемой оптической системы. Она бывает двух типов: положительная

(подушкообразная дисторсия) и отрицательная (бочкообразная дисторсия). Для коррекции дисторсии необходимо иметь два изображения (исходное и искаженное) чтобы сопоставив одинаковые точки на соответствующих изображениях найти коэффициенты для корректирующего преобразования (метод был описан выше).

Используя вышеописанную теорию, можно производить склейку изображений. Несмотря на разные системы координат двух изображений (из-за разного освещения, ракурса, движения объекта и т. д.) достаточно отметить одинаковые точки на соответствующих изображениях (самостоятельно или программно) и затем привести какое-либо изображение в систему координат другого.

Ход выполнения работы

1. Простейшие геометрические преобразования

Figure 1: исходное изображение.

1.1. Сдвиг изображения

Произведем сдвиг на 60 пикселей вниз и 100 пикселей влево.

Listing 1. Сдвиг изображения на Python с помощью cv.warpAffine.

```
# Set the parameters
x_shift = -100
y_shift = 60

# Form transformation matrix and apply transform
```

```
T = np.float32([[1, 0, x_shift], [0, 1, y_shift]])
shifted_img = cv.warpAffine(img, T, (n_cols, n_rows))
```

Figure 2: Сдвинутое изображение.

Сдвиг - самое простое геометрическое преобразование изображения, необходимо задать параметры и вычислить новые координаты пикселей по данным формулам:

$$\begin{cases} x' = x + a \\ y' = y + b \end{cases}$$

Можно либо передать матрицу преобразования во встроенную функцию *cv.warpAffine*, либо самостоятельно произвести сдвиг элементов в массиве изображения, уделяя внимание границам массива.

Listing 2. Сдвиг изображения на Python без использования cv.warpAffine.

```
# Set the parameters
x_shift = -100
y_shift = 60

shifted_img = np.zeros_like(img)

# define the range of indices
if x_shift > 0:
    x_range = range(0, shifted_img.shape[1] - x_shift)
else:
    x_range = range(-x_shift, shifted_img.shape[1])
if y_shift > 0:
    y_range = range(0, shifted_img.shape[0] - y_shift)
else:
    y_range = range(-y_shift, shifted_img.shape[0])
```

```
# shifting
for i in y_range:
    for j in x_range:
        shifted_img[i + y_shift, j + x_shift] = img[i, j]
```

Результаты выполнения обоих программ одинаковы.

1.2. Отражение изображения

Для успешного отражения изображения необходимо преобразовать координаты таким образом:

$$\begin{cases} x' = x \\ y' = -y \end{cases} - \text{отражение относительно оси } oX$$

Однако также необходимо применить сдвиг изображения по оси oX на *(n_rows - 1)* вниз для того, чтобы избавиться от отрицательных координат пикселей.

Listing 3. Отражение изображения относительно оси oX на Python.

```
# Form the transformation matrix with reflect and shift
T = np.float32([[1, 0, 0], [0, -1, n_rows - 1]])
# Apply transform
reflected_img = cv.warpAffine(img, T, (n_cols, n_rows))
```

Figure 3: отраженное по оси oX изображение.

Также можно использовать встроенную функцию *cv.flip()* для отражения.

Listing 4. Отражение с помощью cv.flip() на Python.

```
# Using flip
reflected_img = cv.flip(img, 0)
```

Результаты работы обоих программ идентичны.

1.3. Однородное масштабирование изображения

Для выполнения данного преобразования необходимо изменить координаты следующим образом:

$$\begin{cases} x' = \alpha x \\ y' = \alpha y \end{cases}, \qquad \alpha > 0$$

Также необходимо изменить соответственно размеры результирующего изображения $(n_{cols} * \alpha, n_{rows} * \alpha)$ и применить интерполяцию к неопределенным пикселям.

Listing 5. Однородное масштабирование на Python.

```
# Set the parameters
x_scale = 2.3
y_scale = 2.3

# Form the transformation matrix
T = np.float32([[x_scale, 0, 0], [0, y_scale, 0]])

# Apply transformation
resized_img = cv.warpAffine(img, T, (int(n_cols * x_scale), int(n_rows
* y_scale)), flags=cv.INTER_CUBIC)
```

Figure 4: изображение после применения однородного масштабирования $\alpha = 2.3$.

Естественно, при увеличении размеров изображения, хотя мы и применяет интерполяцию неопределенных пикселей будет ухудшение качества. Метод интерполяции *CV.INTER_CUBIC* - бикубическая интерполяция. При изменении размера и интерполяции новых пикселей этот метод действует на соседние пиксели изображения размером 4×4. Затем берется среднее значение веса 16 пикселей для создания нового интерполированного пикселя.

Также для масштабирования можно использовать встроенную в библиотеку OpenCV функцию *resize*.

Listing 6. Однородное масштабирование с помощью cv.resize на Python.

```
# Using cv.resize func
resized_img = cv.resize(img, (int(n_cols * x_scale), int(n_rows *
y_scale)), interpolation=cv.INTER_CUBIC)
```

Результаты применения первого и второго методов одинаковы.

1.4. Поворот изображения

Для совершения поворота вокруг заданной точки необходимо произвести сдвиг изображения так, чтобы заданная точка пришла в координату $(0, 0)$, затем совершить непосредственно поворот (используя вычисленную матрицу) и в конце произвести обратный сдвиг изображения.

Listing 7. Поворот изображения вокруг точки $(50,50)$ *на* $30°$ *на Python.*

```
# Set the parameters
rot_dot = [50, 50]
angle_grad = 30
angle_rad = angle_grad * math.pi / 180

# Calculate the transformation matrix
T1 = np.float32([[1, 0, -rot_dot[0]], [0, 1, -rot_dot[1]], [0, 0, 1]])
T2 = np.float32([[math.cos(angle_rad), math.sin(angle_rad), 0], [-
math.sin(angle_rad), math.cos(angle_rad), 0], [0, 0, 1]])
T3 = np.float32([[1, 0, rot_dot[0]], [0, 1, rot_dot[1]], [0, 0, 1]])
T = np.matmul(T3, np.matmul(T2, T1))


# Apply transformation
rotated_img = cv.warpAffine(img, T[0:2, :], (n_cols, n_rows),
flags=cv.INTER_CUBIC)
```

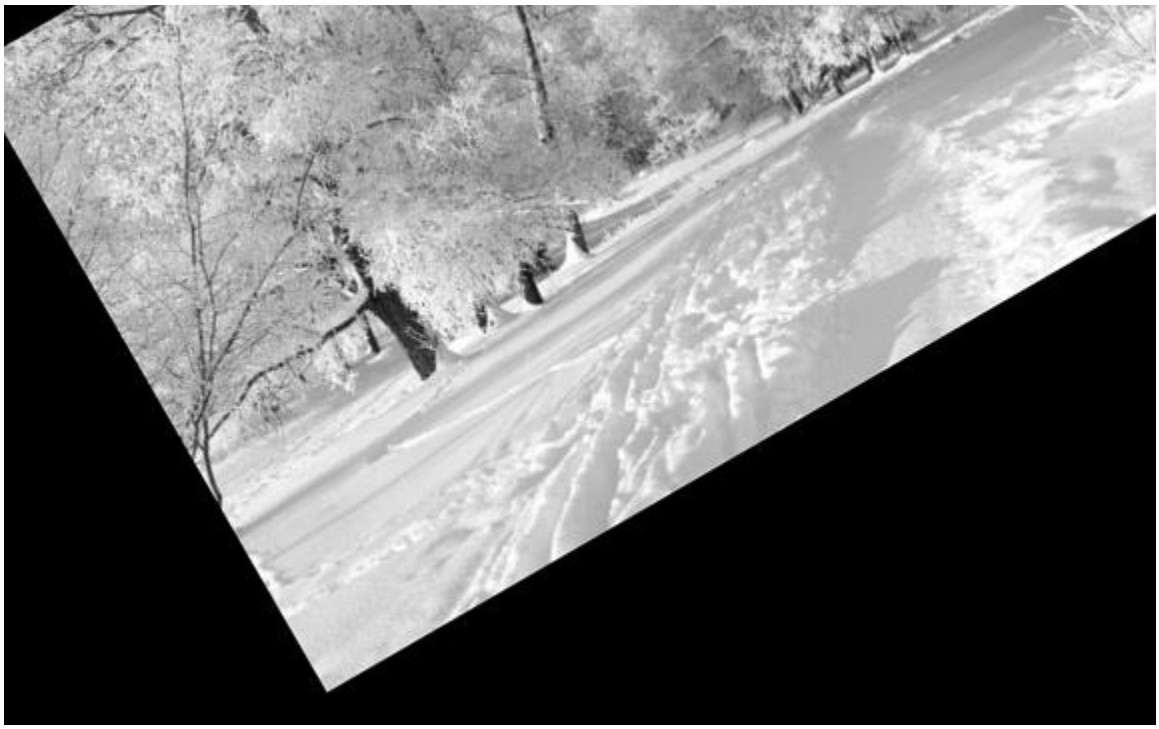

Figure 5: повернутое изображение вокруг точки $(50, 50)$ *на* $30°$.

При данной трансформации также неизбежно появление неопределенных пикселей, для уточнения их интенсивностей используется интерполяция.

Также можно производить рассчитывать матрицу поворота с помощью встроенной функции *getRotationMatrix2D*.

Listing 8. Расчет матрицы поворота с помощью встроенной функции OpenCV на Python.

```
# Get the transformation matrix using getRotationMatrix func
T = cv.getRotationMatrix2D(rot_dot, angle_grad, scale=1)
```

Результаты расчета матриц обоими способами совпадают.

1.5. Произвольное аффинное преобразование

Произвольное аффинное преобразование можно вычислить и применить при помощи функции библиотеки OpenCV *getAffineTransform()*, для этого необходимо задать координаты исходных точек координаты точек после преобразования.

Например, неоднородное масштабирование с коэффициентами $x_{scale} = 1, y_{scale} = 0.5$ можно задать следующими образом:

$$(0,0),(10,0),(5,5) \rightarrow (0,0),(10,0),(5,2.5)$$

Listing 9. Произвольное аффинное преобразование 1.

```
# Set points
pts_src = np.float32([[0, 0], [10, 0], [5, 5]])
pts_dst = np.float32([[0, 0], [10, 0], [5, 2.5]])

# Get the transformation matrix
T = cv.getAffineTransform(pts_src, pts_dst)

# Apply transformation
new_img = cv.warpAffine(img, T, (n_cols, int(n_rows / 2)),
flags=cv.INTER_CUBIC)
```

Figure 6: произвольное аффинное преобразование 1.

Теперь, сделаем «более произвольное» аффинное преобразование:

$$(0,0),(10,0),(5,5) \rightarrow (0.4,1.4),(7,3),(4,6)$$

Listing 10. Произвольное аффинное преобразование 2.

```
# Set points
pts_src = np.float32([[0, 0], [10, 0], [5, 5]])
pts_dst = np.float32([[0.4, 1.4], [7, 3], [4, 6]])

# Get the transformation matrix
T = cv.getAffineTransform(pts_src, pts_dst)

# Apply transformation
new_img = cv.warpAffine(img, T, (n_cols, n_rows),
flags=cv.INTER_CUBIC)
```

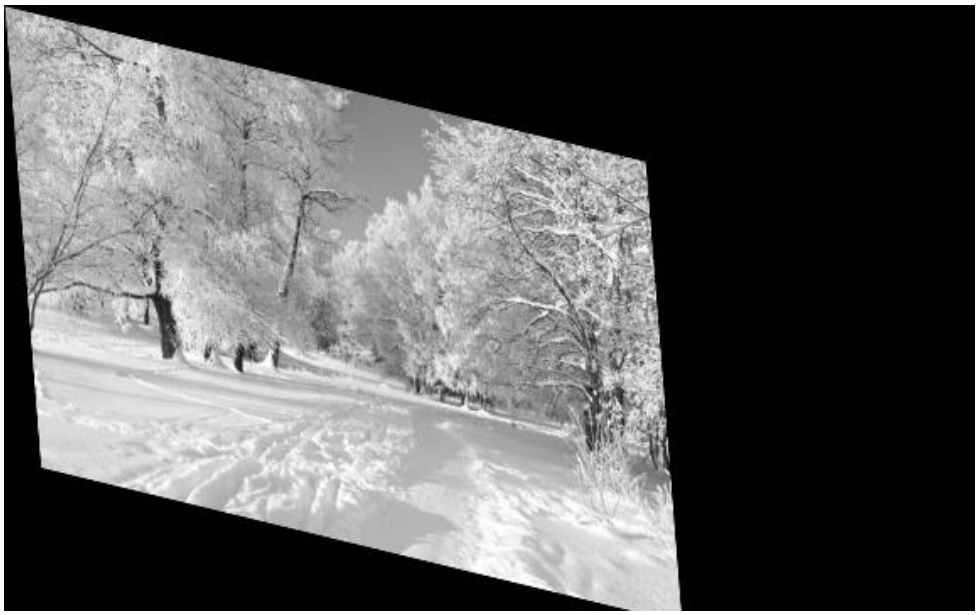

Figure 7: произвольное аффинное преобразование 2.

Данное изображение действительно является результатом аффинного преобразования, так как параллельные прямые остались параллельными, пересекающиеся – пересекающимися, скрещивающиеся – скрещивающимися, а также сохранились отношения фигур и длин отрезков лежащих на одной прямой.

1.6. Скос изображения

Данному преобразованию соответствует следующая система уравнений:

$$\begin{cases} x' = x + s_x y \\ y' = s_y x + y \end{cases}$$

Listing 11. Скос изображения с коэффициентами $s_x = 0.3, s_y = -0.1$ *на Python.*

```
# Set the parameters
s_x = 0.3
s_y = -0.1

# Make the transformation matrix
T = np.float32([[1, s_x, 0], [s_y, 1, 0]])

# Apply transformation
bevelled_img = cv.warpAffine(img, T, (n_cols, n_rows),
flags=cv.INTER_CUBIC)
```

Figure 8: скошенное изображение.

1.7. Неоднородное масштабирование

Неоднородное масштабирование – масштабирование с различными коэффициентами по разным осям:

$$\begin{cases} x' = \alpha x \\ y' = \beta y' \end{cases} \qquad \alpha \neq \beta, \alpha > 0, \beta > 0$$

Реализация данного преобразования аналогична однородному преобразованию. Однако для общности, приведем пример.

Listing 12. Неоднородное преобразование с коэффициентами $x_{scale} = 0.5, y_{scale} = 2.2$ *на Python.*

```
# Set the parameters
x_scale = 0.5
y_scale = 2.2

# Form the transformation matrix
T = np.float32([[x_scale, 0, 0], [0, y_scale, 0]])


# Using cv.resize func
resized_img = cv.resize(img, (int(n_cols * x_scale), int(n_rows *
y_scale)), interpolation=cv.INTER_CUBIC)
```

Figure 9: неоднородное масштабирование изображения.

1.8. Кусочно-линейное отображение

Применим кусочно-линейное отображение, поделив изображение на 4 равных области и в трех из них из областей сделаем линейное преобразование.

Listing 13. Кусочно-линейное отображение на Python.

```
# Construct the new image array
new_img = img.copy()

# Define the areas
top_left_x, top_left_y = np.meshgrid(np.arange(0, int(n_cols / 2)),
np.arange(0, int(n_rows / 2)))
top_right_x, top_right_y = np.meshgrid(np.arange(int(n_cols / 2),
n_cols), np.arange(0, int(n_rows / 2)))
bot_left_x, bot_left_y = np.meshgrid(np.arange(0, int(n_cols / 2)),
np.arange(int(n_rows / 2), n_rows))
bot_right_x, bot_right_y = np.meshgrid(np.arange(int(n_cols / 2),
n_cols), np.arange(int(n_rows / 2), n_rows))

# Stretch by oX the top left part of the image
x_scale = 1.5
T = np.float32([[x_scale, 0, 0], [0, 1, 0]])
new_img[top_left_y, top_left_x] = cv.warpAffine(new_img[top_left_y,
```

```
top_left_x], T, (top_left_x.shape[1], top_left_x.shape[0]),
flags=cv.INTER_CUBIC)

# Stretch by oY the top right part of the image
y_scale = 1.8
T = np.float32([[1, 0, 0], [0, y_scale, 0]])
new_img[top_right_y, top_right_x] = cv.warpAffine(new_img[top_right_y,
top_right_x], T, (top_right_x.shape[1], top_right_x.shape[0]),
flags=cv.INTER_CUBIC)

# Shift by (-30, 40) bottom left part of the image
x_shift = -30
y_shift = 40
T = np.float32([[1, 0, x_shift], [0, 1, y_shift]])
new_img[bot_left_y, bot_left_x] = cv.warpAffine(new_img[bot_left_y,
bot_left_x], T, (bot_left_x.shape[1], bot_left_x.shape[0]),
flags=cv.INTER_CUBIC)

# Bottom right part leave unchanged
```

Figure 10: изображение после кусочно-линейного преобразования.

Это довольно полезный метод, так как используя его, можно проводить действия с интересующими нас отдельными областями изображения.

1.9. Произвольное проекционное преобразование

Заданию проективного отображения соответствует задание матрицы преобразования

$$T = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & 1 \end{bmatrix}.$$

Применим проективное преобразование к исходному изображению.

Listing 14. Проективное преобразование с помощью задания матрицы преобразования на Python.

```
# Set the transformation matrix
T = np.float32([[1.65, 0.3, 0], [0.2, 1.88, 0], [0.001, 0.003, 1]])


# Apply the transformation
new_img = cv.warpPerspective(img, T, (n_cols, n_rows))
```

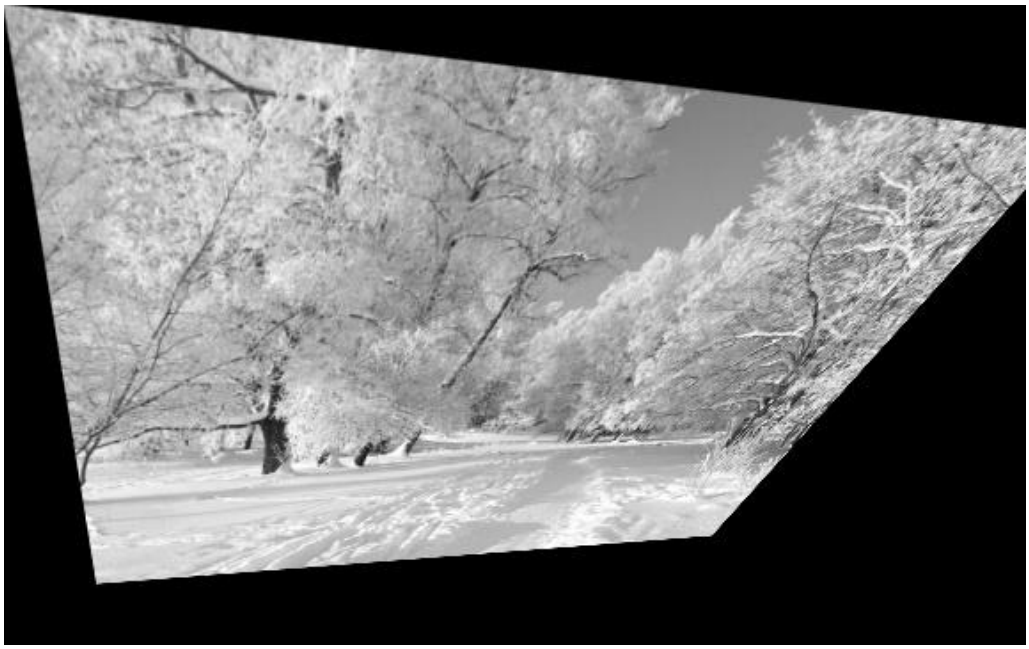

Figure 11: проекционное преобразование.

Отображение действительно является проекционным, так как параллельность линий не сохранилась.

Существует иной способ задания проекционного отображения. Используя функцию библиотеки OpenCV *getPerspectiveTransform()* можно задать такое отображение набором точек до и после необходимого преобразования.

Listing 15. Проекционное отображение с заданием матрицы отображения с помощью функции getPerspectiveTransform() на Python.

```
# or using builtin func
pts_src = np.float32([[50, 461], [461, 461], [461, 50], [50, 50]])
pts_dst = np.float32([[50, 461], [461, 440], [450, 10], [100, 50]])
T = cv.getPerspectiveTransform(pts_src, pts_dst)
```

```
# Apply the transformation
new_img = cv.warpPerspective(img, T, (n_cols, n_rows))
```

Figure 12: проекционное отображение.

Параллельность линий не сохраняется и здесь, что и ожидалось.

1.10. Полиномиальное отображение

Полиномиальное отображение задается несколько иным способом нежели отображения, рассмотренные выше, а именно через полиномы определенного порядка. Благодаря этому есть возможность строить отображения высокого порядка.

Например, рассмотрим и впоследствии применим отображение 2-го порядка:

$$\begin{cases} x' = 0.7x + 0.0001x^2 + 0.001y^2 \\ y' = 0.9y \end{cases}$$

Listing 16. Полиномиальное отображение на Python.

```
# Set the transformation matrix
T = np.float32([[0, 0], [0.7, 0], [0, 0.9], [0.00001, 0], [0.002, 0],
[0.001, 0]])

# Create grid corresponding to our pixels
x, y = np.meshgrid(np.arange(n_cols), np.arange(n_rows))

# Calculate all new X and Y coordinates
x_new = np.round(T[0, 0] + x * T[1, 0] + y * T[2, 0] + x * x * T[3, 0]
+ x * y * T[4, 0] + y * y * T[5, 0]).astype(np.float32)
y_new = np.round(T[0, 1] + x * T[1, 1] + y * T[2, 1] + x * x * T[3, 1]
+ x * y * T[4, 1] + y * y * T[5, 1]). astype(np.float32)
```

```
# Calculate mask for valid indices
mask = np.logical_and(np.logical_and(x_new >= 0, x_new < n_cols),
np.logical_and(y_new >= 0, y_new < n_rows))

# Apply reindexing
polyn_img = np.zeros_like(img)
polyn_img[y_new[mask].astype(int), x_new[mask].astype(int)] =
img[y[mask], x[mask]]
```

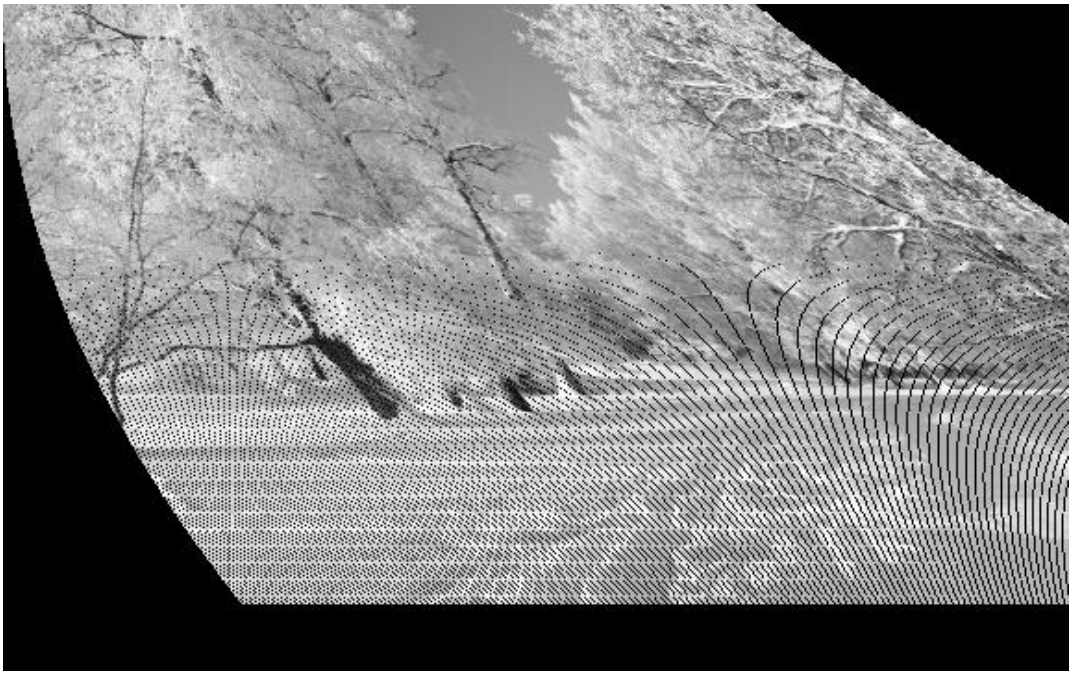

Figure 13: полиномиальное отображение.

По результирующему изображению видно, что отображение является нелинейным.

Так как мы не пользовались встроенными функциями OpenCV, интерполяция неопределенных писклей не проводилась, на изображении это видно, как черные полосы.

Попробуем реализовать интерполяцию – которая высчитывает среднюю интенсивность по области 5x5 и присваивает её неопределенному пикселю.

Для начала найдем все неопределенные пиксели, границу нового изображения и внутренние неопределённые пиксели (именно те, которые нам необходимо будет интерполировать).

Listing 17. Поиск неопределенных пикселей на Python.

```
# Find undefined pixels and borders of new image
undef_all = np.zeros_like(img)
borders = np.zeros like(img)
undef_inner = np.zeros_like(img)
undef_all[y_new[mask].astype(int), x_new[mask].astype(int)] = 255  #
Undefined pixels corresponds to the black
for i in range(n_rows):
    def pixels = np.argwhere(undef all[i, :] == 255)
    if def_pixels.size > 2:
```

```
        img_start = np.squeeze(def_pixels[0])
        img_end = np.squeeze(def_pixels[-1])
        borders[i, [img_start, img_end]] = 255
        row_undef_pix = np.argwhere(undef_all[i, img_start:img_end] == 0)
        undef_inner[i, row_undef_pix + img_start] = 255
```

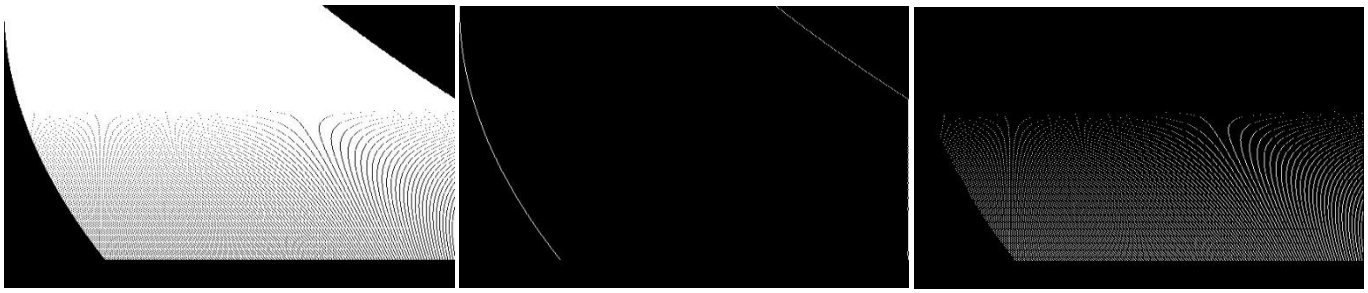

Figure 14: неопределенные пиксели (слева), границы нового изображения (в центре) и внутренние неопределенные пиксели (справа)

Итак, осталось только применить интерполяцию:

Listing 18. Интерполяция изображения на Python.

```
# Apply interpolation
undef_inner_arr = np.argwhere(undef_inner == 255)  # An array containing
inner undefined pixels
for undef_p in undef_inner_arr:
    intensities = []
    # Construct 5x5 punctured area
    area = np.array([[undef_p[0] - 2, undef_p[1] - 2], [undef_p[0] - 2,
undef_p[1] - 1], [undef_p[0] - 2, undef_p[1]], [undef_p[0] - 2, undef_p[1] +
1], [undef_p[0] - 2, undef_p[1] + 2],
                     [undef_p[0] - 1, undef_p[1] - 2], [undef_p[0] - 1,
undef_p[1] - 1], [undef_p[0] - 1, undef_p[1]], [undef_p[0] - 1, undef_p[1] +
1], [undef_p[0] - 1, undef_p[1] + 2],
                     [undef_p[0], undef_p[1] - 2], [undef_p[0], undef_p[1] -
1], [undef_p[0], undef_p[1] + 1], [undef_p[0], undef_p[1] + 2],
                     [undef_p[0] + 1, undef_p[1] - 2], [undef_p[0] + 1,
undef_p[1] - 1], [undef_p[0] + 1, undef_p[1]], [undef_p[0] + 1, undef_p[1] +
1], [undef_p[0] + 1, undef_p[1] + 2],
                     [undef_p[0] + 2, undef_p[1] - 2], [undef_p[0] + 2,
undef_p[1] - 1], [undef_p[0] + 2, undef_p[1]], [undef_p[0] + 2, undef_p[1] +
1], [undef_p[0] + 2, undef_p[1] + 2]])
    # Find the area average intensities of the defined pixels
    for p in area:
        if (p[0] >= 0) and (p[0] < n_rows) and (p[1] >= 0) and (p[1] <
n_cols) and (undef_all[p[0], p[1]] == 255):
            intensities.append(polyn_img[p[0], p[1]])
    polyn_img[undef_p[0], undef_p[1]] = int(np.mean(intensities))  # Assign
the found intensity
```

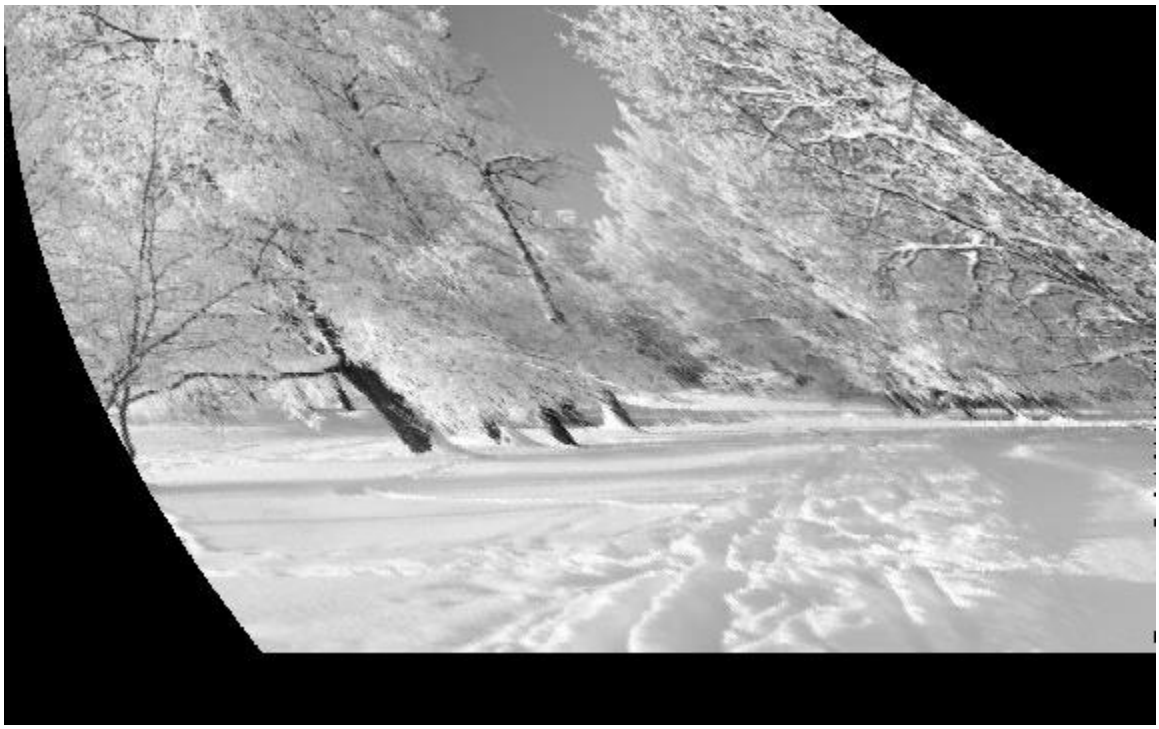

Figure 15: изображение после интерполяции 5x5.

Интерполяция удалась, неопределённые пиксели устранились, картинка стала более сглаженной из-за метода интерполяции, возможно необходимо использовать другой. Также необходимо оптимизировать программу, используя больше векторных операций, а не попиксельные вычисления.

Сравним результат работы с программой, использующей встроенную функцию *remap()*.

Важно, что при геометрических преобразованиях изображения с помощью функции *remap()* алгоритм вычисления массивов-сеток *x_new*, *y_new* будет другим, так как функция $dst = remap(src, map_x, map_y)$ делает следующее: для каждого пикселя в конечном изображении ищет, откуда он взялся в исходном изображении, а затем присваивает интерполированное значение, то есть

$$dst(x,y) = src\left(map_x(x,y), map_y(x,y)\right).$$

В то же время алгоритм реализованный выше делал так:

$$dst\left(map_x(x,y), map_y(x,y)\right) = src(x,y).$$

Таким образом для реализации алгоритма с использованием функции *remap()*, нужно передавать не map_x, map_y, а обратные функции map_x^{-1}, map_y^{-1}.

Listing 19. Полиномиальное отображение с помощью remap().

```
# Set the transformation matrix
T = np.float32([[0, 0], [0.7, 0], [0, 0.9], [0.00001, 0], [0.002, 0],
[0.001, 0]])

# Create grid corresponding to our pixels
x, y = np.meshgrid(np.arange(n_cols), np.arange(n_rows))

# Calculate all new X and Y coordinates
x_new = np.round(np.sqrt((x / T[3,0]) + ((T[1,0] + T[4,0] * y) / (2 *
T[3,0])) ** 2 - ((T[5,0] * y ** 2)/T[3,0])) - ((T[1,0] + T[4,0] * y) /
(2 * T[3,0]))).astype(np.float32)
y_new = np.round(y / T[2,1]).astype(np.float32)

# Apply remapping
polyn_img = cv.remap(img, x_new, y_new, cv.INTER_LINEAR)
```

Figure 16: изображение после полиномиального преобразования.

Результирующие изображения совпадают, а значит все теоретические выкладки приведенные выше верны.

1.11. Синусоидальное отображение

Рассмотрим синусоидальное преобразование. По сути, оно отличается от рассмотренного ранее полиномиального только тем, что вместо полиномов мы использует гармонические функции.

Listing 20. Синусоидальное отображение на Python.

```
# Set the parameters
amp = 20
freq_factor = 2
phase = 0

# Create grid corresponding to our pixels
x, y = np.meshgrid(np.arange(n_cols), np.arange(n_rows))
# Calculate all new X coordinates
x = x + amp * np.sin((freq_factor * math.pi * y + phase) / 180)

# Apply remapping
sin_img = cv.remap(img, x.astype(np.float32), y.astype(np.float32),
cv.INTER_NEAREST)
```

Figure 17: синусоидальное преобразование.

Изображение действительно изменилось в соответствии с предоставленным гармоническим законом.

2. Коррекция дисторсии

Исследуем явление дисторсии на практике. Причины появления данного искажения – в искривлении прямых линий, из-за того, что световые лучи, проходящие через центр линзы сойдутся в точке, расположенной дальше линзы, нежели те лучи, которые проходят через края линзы. Итого, дисторсию можно выразить следующим уравнением:

$\boldsymbol{R} = b_0\boldsymbol{r} + F_3 r^2\boldsymbol{r} + F_5 r^4\boldsymbol{r} + F_7 r^6\boldsymbol{r} + \cdots$, где $\boldsymbol{r}$ – вектор, задающий точку в плоскости, расположенную перпендикулярно оптической оси, $\boldsymbol{R}$ – соответствующая точка на изображении, b_0

– коэффициент линейного увеличения, r – длина вектора $\boldsymbol{r}$, F_i – коэффициенты дисторсии порядка i.

Дисторсия бывает двух типов: подушкообразная $(sign(F_3) = sign(b_o))$ и бочкообразная в противном случае.

Попробуем устранить дисторсию с изображения. Для начала возьмём исходное изображение и искусственно создадим дисторсию.

Figure 18: исходное изображение.

В программе ниже используется функция *remap()* поэтому при расчете новых координат используется обратная функция к $R^{-1}(r)$.

Listing 21. Наложение подушкообразной дисторсии на изображение на Python.

```
# Form x,y grid
x, y = np.meshgrid(np.arange(n_cols), np.arange(n_rows))

# Shift and normalize grid
x mid = n cols / 2
y_mid = n_rows / 2
x = (x - x mid) / x mid
y = (y - y_mid) / y_mid

# Set linear expansion factor
b 0 = 1.005
```

```
# Convert to polar and do transformation
r, theta = cv.cartToPolar(x, y)
F3 = 0.05
r_new = (((2/3)**(1/3))*b_0)/((np.sqrt(3*(4*b_0**3*F3**3+27*F3**4*r**2))-
9*F3**2*r)**(1/3)) - ((np.sqrt(3*(4*b_0**3*F3**3+27*F3**4*r**2))-
9*F3**2*r)**(1/3))/(2**(1/3)*3**(2/3)*F3)

# Undo conversion, normalization and shift
u, v = cv.polarToCart(r_new, theta)
u = u * x_mid + x_mid
v = v * y_mid + y_mid

# Do remapping
dist_img = cv.remap(img, u.astype(np.float32), v.astype(np.float32),
cv.INTER_LINEAR)
```

Figure 19: изображение после наложения подушкообразной дисторсии.

На изображение действительно наложилась подушкообразная дисторсия.

Для устранения дисторсии необходимо отметить одинаковые точки на исходном и искаженном изображениях.

$A(197,124) \rightarrow A'(186,117), \qquad B(724,450) \rightarrow B'(729,453)$

После перевода в полярные координаты получается, что $r(A) = 0.84297,\ r(B) = 0.67321,$ $r(A') = 0.875037,\ r(B') = 0.68734.$

Теперь, необходимо составить систему из 2-х уравнений и найти коэффициенты b_0, F_3:

$$\begin{cases} r(A') = b_0 r(A) + F_3 r(A)^3 \\ r(B') = b_0 r(B) + F_3 r(B)^3 \end{cases} \Rightarrow \begin{cases} b_0 = 0.9938 \\ F_3 = 0.0539 \end{cases}$$

Теперь выполним наложение бочковидной дисторсии, то есть коэффициенты будут равны $b_0 = 0.9938,\ F_3 = -0.0539.$

Figure 20: результат изображения с устраненной дисторсией.

Методом выше удалось подобрать коэффициенты преобразования, и они оказались очень близки к истинным значениям ($b_0 = 1,\ b_{0_{calc}} = 0.9938,\ F_3 = 0.05,\ F_{3_{calc}} = 0.0539$).

Действительно, результирующее изображение не имеет признаков наличия дисторсии, а значит работа выполнена успешно.

3. «Склейка» изображений

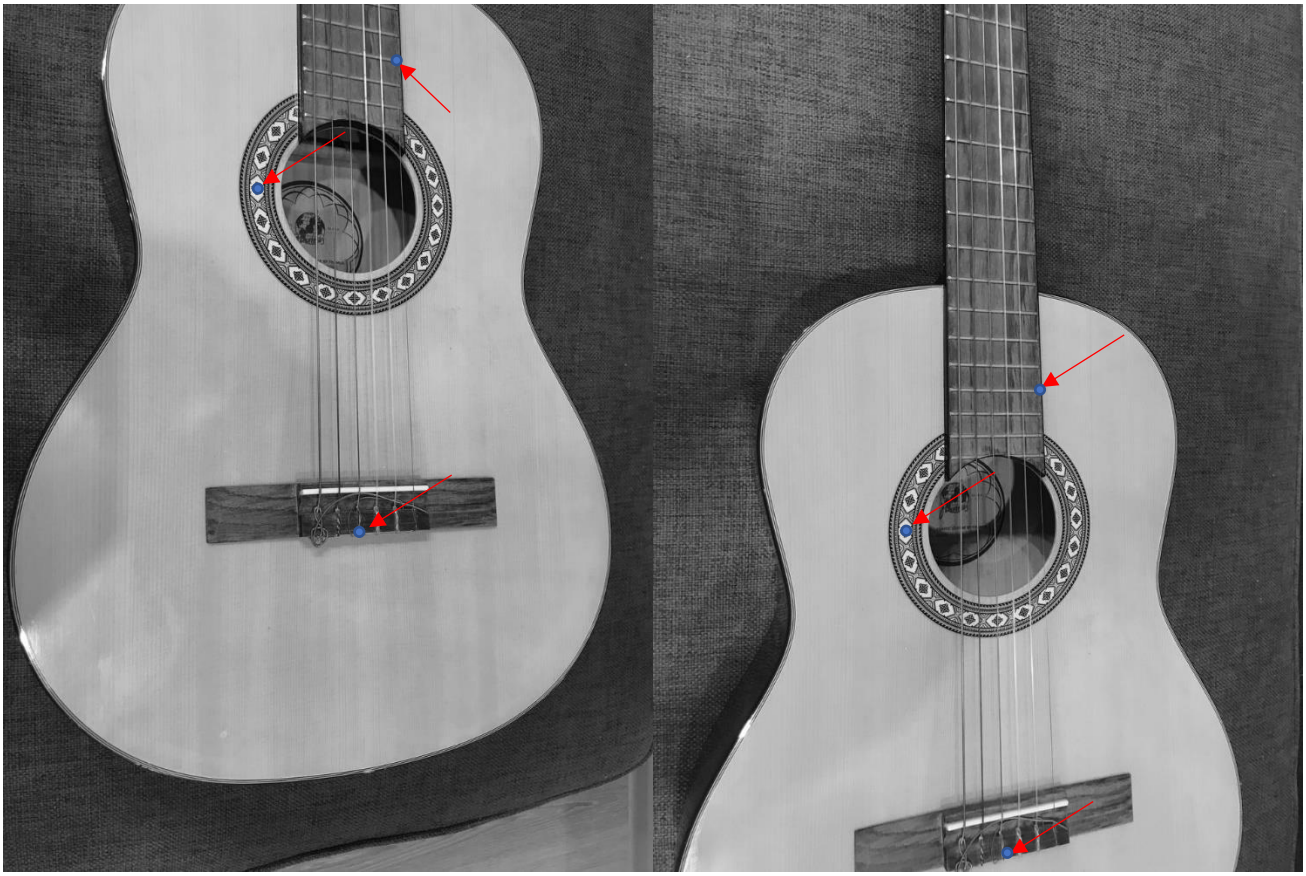

Figure 21: исходные изображения.

Для корректной склейки изображений для начала необходимо преобразовать их в общую систему координат. Будем преобразовывать правое изображение в систему координат левого. Для этого отметим 3 одинаковые точки на обоих изображениях и составим систему для поиска параметров полиномиального преобразования 1-го порядка. Выбор полиномиального отображения 1-го обусловлен тем, что параллельные линии остались параллельными, пересекающиеся-пересекающимися и отношения площадей фигур не изменилось.

Listing 22. Преобразование изображения в систему координат другого на Python.

```
# Dots
d1 = [[516, 1259], [372, 778], [566, 571]]
d2 = [[526, 779], [374, 271], [580, 84]]

# Find parameters of the transform
# To translate the top image into the bot coordinate system
buf = np.linalg.inv([[1, d1[0][0], d1[0][1]], [1, d1[1][0], d1[1][1]],
                     [1, d1[2][0], d1[2][1]]])
x param = np.matmul(buf, np.reshape([d2[0][0], d2[1][0], d2[2][0]], (3, 1)))
y_param = np.matmul(buf, np.reshape([d2[0][1], d2[1][1], d2[2][1]], (3, 1)))

# Create grid corresponding to our pixels
x, y = np.meshgrid(np.arange(n cols), np.arange(n rows))
```

```
# Calculate all new X and Y coordinates
x_new = np.round(x_param[0, 0] + x_param[1, 0] * x + x_param[2, 0] *
y).astype(np.float32)
y_new = np.round(y_param[1, 0] * x + y_param[2, 0] * y).astype(np.float32) #
Do notuse y_param[0, 0] because we don't need oY shift

# Calculate mask for valid indices
mask = np.logical_and(np.logical_and(x_new >= 0, x_new < n_cols),
np.logical_and(y_new >= 0, y_new < n_rows))

# Apply reindexing
new_img = np.zeros_like(img_1)
new img[y new[mask].astype(int), x new[mask].astype(int)] = img 1[y[mask],
x[mask]]
```

Также была применена ручная интерполяция, код который был выше.

Figure 22: результат перевода изображения в другую систему координат.

Теперь применим алгоритм «склеивания» изображений по оси oY. Для этого найдем ту область изображения, корреляция которой будет максимальна и по этой области совместим изображения.

Listing 23. Вертикальное склеивание двух изображений на Python.

```
# Match template
template_size = 10
template = img_top[-template_size:, :]
res = cv.matchTemplate(img_bot, template, cv.TM_CCOEFF)

# Find max and min correlation values
min_val, max_val, min_loc, max_loc = cv.minMaxLoc(res)

# Construct stitched image
full_img = np.zeros((img_top.shape[0] + img_bot.shape[0] - max_loc[1] -
template_size, img_top.shape[1]), dtype=np.uint8)
full_img[:img_top.shape[0], :] = img_top
full_img[img_top.shape[0]:, :] = img_bot[max_loc[1] + template_size:, :]
```

Figure 23: результирующее изображение.

Итого, «склеивание» изображений прошло успешно, однако из-за неточного перевода первого изображения в систему координат второго (отображение имеет более сложный характер и точно не описывается полиномом 1 степени), мы наблюдаем артефакты в правой нижней части результирующего изображения.

Выводы

В первой части работы были исследованы на практике геометрические преобразования изображений. В начале были рассмотрены конформные (сдвиг, отражение, однородное масштабирование и поворот), далее была группа аффинных преобразований – базовыми отображениями которого являются конформные преобразования, наряду со скосом и неоднородным масштабированием. В конце была рассмотрена еще более обширная группа отображений – проекционные отображения (параллельность линий может нарушаться). Было выявлено, что геометрические преобразования очень удобно представлять в матричном виде (тогда при наложения нескольких преобразований – результирующее – перемножение матриц). Нелинейные преобразования сильно нарушают геометрию фигур и не сохраняют в общем случае параллельность линий. Описываться могут как в матричном, так и с помощью нелинейных уравнений.

Немаловажную роль при геометрических преобразованиях играет интерполяция или интерполяция неопределенных писклей, так как в результате отображений очень часто получаются нецелые координаты пикселей. В данной работе была использована как встроенная интерполяцию, так и собственноручно написанная.

Также стоит упомянуть о линейно-кусочном преобразовании – это достаточно полезный на практике прием позволяющий преобразовывать нужным образом отдельные области изображения.

Во второй части работы предлагалось наложить дисторсию, а затем скорректировать её путем наложения координатной сетки. Итоговый вывод такой, что после нахождения параметров дисторсии (путем выбора одинаковых точек и решения систему уравнений) бочкообразная дисторсию устраняется подушкообразной и наоборот.

В заключительной части предстояло «склеить» изображения, находящиеся в разных системах координат. Путем анализа изображений и теоретических знаний об отображениях я определил примерное отображение (полиномиальное 1-го порядка) – которое позволит перевести оба изображения в систему координат второго. После этого осталось произвести склейку, используя поиск максимальной корреляции маленького шаблона одного изображения на другом.

В результате проделанной работы были поняты алгоритмы геометрических преобразований изображений, коррекции дисторсий, а также «склеивания» изображений.