Лабораторная работа №3
Геометрические преобразования изображений

Цель работы

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

Методические рекомендации

До начала работы студенты должны ознакомиться с основными функциями среды MATLAB или библиотеки OpenCV по работе с геометрическими преобразованиями изображений. Лабораторная работа рассчитана на 4 часа.

Теоретические сведения

Геометрические преобразования изображений подразумевают пространственное изменение местоположения множества пикселей с целочисленными координатами (x,y) в другое множество с координатами (x',y'), причем интенсивность пикселей сохраняется. В двумерных плоских геометрических преобразованиях, как правило, используется евклидово пространство $\mathbf{P}^2$ с ортонормированной декартовой системой координат. В этом случае пикселю изображения соответствует пара декартовых координат, которые интерпретируются в виде двумерного вектора, представленного отрезком из точки (0,0) до точки $X_i = (x_i,y_i)$. Двумерные преобразования на плоскости можно представить в виде движения точек, соответствующих множеству пикселей.

Для общности с дальнейшими преобразованиями будем использовать *однородные координаты*, обладающие тем свойством, что определяемый ими объект не меняется при умножении всех координат на одно и то же ненулевое число. Из-за этого свойства необходимое количество координат для представления точек всегда на одну

больше, чем размерность пространства $\mathbf{P}^n$, в котором эти координаты используются. Например, для представления точки $X = (x,y)$ на плоскости в двумерном пространстве $\mathbf{P}^2$ необходимо три координаты $X = (x,y,w)$. Проиллюстрируем это следующим примером:

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = w \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Leftrightarrow \begin{bmatrix} x' & y' & w \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} w, \tag{3.1}$$

где w — скалярный произвольный множитель, $x = \dfrac{x'}{w}$, $y = \dfrac{y'}{w}$.

При помощи троек однородных координат и матриц третьего порядка можно описать любое линейное преобразование плоскости. Таким образом, геометрические преобразования являются матричными преобразованиями, и множества координат пикселей преобразованного и исходного изображений связаны следующим матричным соотношением либо в строчном виде $X' = XT$, либо в столбцовом $X' = T^{\mathrm{T}}X$. Распишем эти матричные соотношения:

$$\begin{bmatrix} x' & y' & w' \end{bmatrix} = \begin{bmatrix} x & y & w \end{bmatrix} \cdot \begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix} \tag{3.2}$$

$$\Leftrightarrow \begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & I \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ w \end{bmatrix}. \tag{3.3}$$

Точка с декартовым координатами (x,y) в однородных координатах запишется как $(x,y,1)$:

$$\begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \cdot \begin{bmatrix} A & D & 0 \\ B & E & 0 \\ C & F & 1 \end{bmatrix} \tag{3.4}$$

$$\Leftrightarrow \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} A & B & C \\ D & E & F \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \tag{3.5}$$

Формулу (3.4) можно представить в виде следующей системы уравнений:

$$\begin{cases} x' = Ax + By + C, \\ y' = Dx + Ey + F. \end{cases} \tag{3.6}$$

Линейные преобразования

Конформное отображение — это отображение, при котором сохраняется форма бесконечно малых фигур и углы между кривыми в точках их пересечения. Основными линейными конформными преобразованиями являются евклидовы преобразования. К ним относятся сдвиг, отражение, однородное масштабирование и поворот. Конформные преобразования являются подмножеством аффинных преобразований.

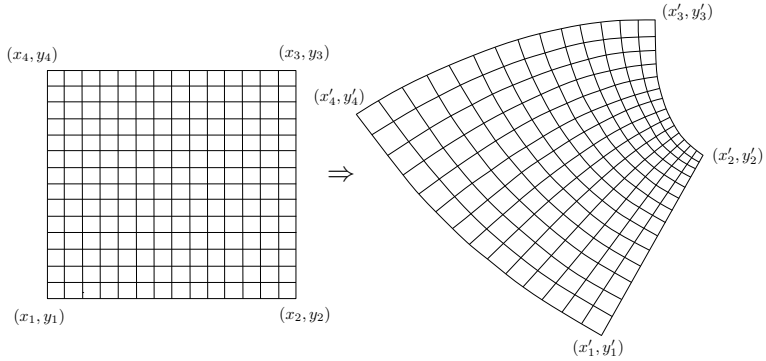

Рис. 3.1 — Конформное отображение: перпендикулярность линий в местах их пересечений сохраняется.

Сдвиг изображения

Система уравнений (3.6) и матрица преобразования координат T в случае *сдвига* изображения $A = E = 1$, $B = D = 0$ примут вид:

$$\begin{cases} x' = x + C, \\ y' = y + D, \end{cases} \tag{3.7}$$

$$\Leftrightarrow \begin{bmatrix} x' & y' & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} T \Rightarrow T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ C & F & 1 \end{bmatrix}, \qquad (3.8)$$

где C и F — сдвиг по осям Ox и Oy соответственно.

Листинг 3.1. Сдвиг изображения в среде MATLAB на 50 и 100 пикселей по осям Ox и Oy соответственно. Размер исходного рисунка `roadSign.jpg` 300×300 пикселей.

```
1  I = imread('roadSign.jpg');
2  T = [1 0 0; 0 1 0; 50 100 1];
3  tform = affine2d(T);
4  I_shift = imwarp(I, tform, ...
5    'Interp', 'nearest', ...
6    'OutputView', imref2d(size(I), ...
7    [1 size(I,2)], [1 size(I,1)]) ...
8  );
```

Функция `affine2d()` создает матрицу преобразования, которая применяется к изображению `I` при помощи функции `imwarp()`. Дополнительные параметры задают одинаковые координаты для исходного и преобразованного изображений, а также метод интерполяции для неопределенных пикселей.

В отличие от MATLAB, в библиотеке OpenCV матрицы хранятся в виде строк, поэтому следует использовать формулу (3.5) для задания матрицы преобразования:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = T \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \Rightarrow T = \begin{bmatrix} 1 & 0 & C \\ 0 & 1 & F \\ 0 & 0 & 1 \end{bmatrix}, \qquad (3.9)$$

Кроме того, последний ряд матрицы не хранится, поэтому матрица будет задаваться следующим образом:

$$T = \begin{bmatrix} 1 & 0 & C \\ 0 & 1 & F \end{bmatrix}, \qquad (3.10)$$

Листинг 3.2.Сдвиг изображения с использованием библиотеки OpenCV и языка программирования C++ на 50 и 100 пикселей по осям Ox и Oy соответственно. Размер исходного рисунка `roadSign.jpg` 300×300 пикселей.

```
Mat I =
  imread("roadSign.jpg", IMREAD_COLOR);
Mat T = (Mat_<double>(2, 3) <<
          1, 0, 50 ,
          0, 1, 100);
Mat I_shift;
warpAffine(I, I_shift, T,
  Size(I.cols, I.rows));
```

В случае использования библиотеки OpenCV и языка программирования C++ все матрицы (в том числе и матрица изображения) хранятся с использованием объекта `cv::Mat`, представляющего собой матрицу произвольного размера. Матрицу преобразования можно создать вручную с помощью операции `Mat_<double>(2, 3) <<`, где аргументы `(2, 3)` задают размеры матрицы (количество строк и количество столбцов матрицы соответственно), а аргументы, передаваемые после оператора `<<`, определяют элементы матрицы построчно, начиная с первой строки. Функция `cv::warpAffine()` используется для выполнения любого вида аффинного преобразования изображения. Первый аргумент — это изображение для преобразования, второй — место для сохранения результирующего изображения, третий — матрица преобразования, а четвертый — разрешение, которое будет использоваться при формировании результирующего изображения.

Листинг 3.3.Сдвиг изображения с использованием библиотеки OpenCV и языка программирования Python на 50 и 100 пикселей по осям Ox и Oy соответственно. Размер исходного рисунка `roadSign.jpg` 300×300 пикселей.

```
rows, cols = I.shape[0:2]
T = np.float32([[1, 0, 50], [0, 1, 100]])
I_shift = cv2.warpAffine(I, T, (cols, rows))
```

В случае использования языка программирования Python матрица преобразования представляет собой простой массив библиотеки NumPy размерами 3×2, поэтому его можно создать с помощью встроенной функции библиотеки NumPy `numpy.float32()`. Как и в C++, в Python функция `cv2.warpAffine()` используется для выполнения любого вида аффинного преобразования, однако, в отличие от реализации библиотеки OpenCV для C++, результирую-

щее изображение не является обязательным аргументом, а может быть и возвращаемым значением функции. Все остальные аргументы те же. В реализации библиотеки OpenCV для языка программирования Python массивы NumPy используются повсеместно для хранения изображений, поэтому все методы, применимые к массивам NumPy, применимы и для изображений. Следует обратить внимание, что в параметре формы изображения `I.shape` разрешение изображения хранится в виде количества строк и столбцов, а в функцию преобразования оно должно передаваться в виде количества столбцов и строк. Это связано с тем, что матрицы традиционно задаются указанием количества строк и столбцов, а изображения — указанием ширины и высоты.

Отражение изображения

Система уравнений (3.6) и матрица преобразования координат T в случае *отражения* изображения вдоль оси Ox при $A = 1$, $E = -1$, $B = C = D = F = 0$ примут вид:

$$\begin{cases} x' = x \\ y' = -y \end{cases} \Rightarrow T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.11}$$

Листинг 3.4. Отражение относительно оси Ox в среде MATLAB.

```
T = [1 0 0; 0 -1 0; 0 0 1];
tform = affine2d(T);
I_reflect = imwarp(I, tform);
```

В случае использования библиотеки OpenCV матрица преобразования для отражения изображения определяется следующим образом:

$$T = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & I.rows - 1 \end{bmatrix}, \tag{3.12}$$

Дополнительный сдвиг изображения необходим для устранения отрицательных координат пикселей изображения, которые будут получены после преобразования.

Листинг 3.5. Отражение относительно оси Ox с использованием библиотеки OpenCV и языка программирования C++.

```
1   Mat T = (Mat_<double>(2, 3) <<
2            1, 0, 0,
3            0, -1, I.rows - 1);
4   Mat I_reflect;
5   warpAffine(I, I_reflect, T,
6     Size(I.cols, I.rows));
```

Листинг 3.6. Отражение относительно оси Ox с использованием библиотеки OpenCV и языка программирования Python.

```
1   T = np.float32([[1, 0, 0],
2                   [0, -1, rows - 1]])
3   I_reflect = \
4     cv.warpAffine(I, T, (cols, rows))
```

В библиотеке OpenCV также присутствует функция `flip()`, предназначенная для выполнения отражения изображения. Данная функция требует передачи одного параметра в дополнение к изображению. Значение параметра 0 означает отражение относительно оси Ox, 1 — отражение относительно оси Oy, а -1 — отражение относительно обеих осей. Например, отражение изображения относительно оси Ox с использованием языка программирования C++ будет выглядеть следующим образом:

```
cv::flip(I, I_reflect, 0);
```

А с использованием языка программирования Python:

```
I_reflect = cv2.flip(I, 0)
```

Однородное масштабирование изображения

Система уравнений (3.6) и матрица преобразования координат T в случае *масштабирования* изображения $A = \alpha$, $E = \beta$, $B = C = D = F = 0$ примут вид:

$$\begin{cases} x' = \alpha x, \alpha > 0 \\ y' = \beta y, \beta > 0 \end{cases} \Rightarrow T = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.13}$$

если $\alpha < 1$ и $\beta < 1$, то изображение уменьшается, если $\alpha > 1$ и $\beta > 1$ — увеличивается. Если $\alpha \neq \beta$, то пропорции будут не одинаковыми

по ширине и высоте. В общем случае данное отображение будет являться аффинным, а не конформным.

Листинг 3.7. Увеличение исходного изображения в два раза в среде MATLAB.

```
T = [2 0 0; 0 2 0; 0 0 1];
tform = affine2d(T);
I_scale = imwarp(I, tform);
```

В случае использования библиотеки OpenCV матрица преобразования для масштабирования изображения определяется следующим образом:

$$T = \begin{bmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \end{bmatrix}, \tag{3.14}$$

Листинг 3.8. Увеличение исходного изображения в два раза с использованием библиотеки OpenCV и языка программирования C++.

```
Mat T = (Mat_<double>(2, 3) <<
          2, 0, 0,
          0, 2, 0);
Mat I_scale;
warpAffine(I, I_scale, T,
  Size(int(I.cols * 2), int(I.rows * 2)));
```

Листинг 3.9. Увеличение исходного изображения в два раза с использованием библиотеки OpenCV и языка программирования Python.

```
T = np.float32([[scale_x, 0, 0],
                [0, scale_y, 0]])
I_scale = cv2.warpAffine(I, T,
  (int(cols * scale_x),
  int(rows * scale_y)))
```

Чтобы учесть изменение разрешения изображения при масштабировании, размер результирующего изображения был вычислен с учетом коэффициента масштабирования.

В библиотеке OpenCV также присутствует функция `resize()`, предназначенная для изменения размера изображения. Например,

увеличение размера изображения в два раза с использованием языка программирования C++ будет выглядеть следующим образом:

```
cv::resize(I, I_scale,
  Size(int(I.cols * 2), int(I.rows * 2)));
```

А с использованием языка программирования Python:

```
I_scale = cv2.resize(I, None, fx = 2, fy = 2,
  interpolation = cv2.INTER_CUBIC)
```

Поворот изображения

Система уравнений (3.6) и матрица преобразования координат T в случае *поворота* изображения по часовой стрелке $A = \cos\varphi$, $B = -\sin\varphi$, $D = \sin\varphi$, $E = \cos\varphi$, $C = F = 0$ примут вид:

$$\begin{cases} x' = x\cos\varphi - y\sin\varphi \\ y' = x\sin\varphi + y\cos\varphi \end{cases} \Rightarrow T = \begin{bmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}. \qquad (3.15)$$

Если $\varphi = 90°$, то $\cos\varphi = 0$ и $\sin\varphi = 1$ и выражение (3.15) примет вид:

$$\begin{cases} x' = -y \\ y' = x \end{cases} \Rightarrow T = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \qquad (3.16)$$

Листинг 3.10. Поворот изображения на $\varphi = 17°$ в среде MATLAB.

```
1  phi = 17*pi/180;
2  T = [cos(phi) sin(phi) 0;
3    -sin(phi) cos(phi) 0;
4    0 0 1];
5  tform = affine2d(T);
6  I_rotate = imwarp(I, tform);
```

В случае использования библиотеки OpenCV матрица преобразования для поворота изображения определяется следующим образом:

$$T = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \end{bmatrix}, \tag{3.17}$$

Листинг 3.11. Поворот изображения на $\varphi = 17°$ с использованием библиотеки OpenCV и языка программирования C++.

```
double phi = 17.0 * M_PI / 180;
Mat T = (Mat_<double>(2, 3) <<
          cos(phi), -sin(phi), 0,
          sin(phi), cos(phi), 0);
Mat I_rotate;
warpAffine(I, I_rotate, T,
  Size(I.cols, I.rows));
```

Листинг 3.12. Поворот изображения на $\varphi = 17°$ с использованием библиотеки OpenCV и языка программирования Python.

```
phi = 17.0 * math.pi / 180
T = np.float32(
  [[math.cos(phi), -math.sin(phi), 0],
  [math.sin(phi), math.cos(phi), 0]])
I_rotate = \
  cv2.warpAffine(I, T, (cols, rows))
```

Так как при использовании матрицы преобразования (3.17) поворот осуществляется вокруг верхнего левого угла изображения с координатами $(0, 0)$, то поворот вокруг произвольной точки следует определять как сдвиг изображения так, чтобы желаемая точка вращения находилась в верхнем левом углу, затем поворот на заданный угол и обратный сдвиг. Это преобразование можно вычислить в матричной форме, а затем выполнить как одно аффинное преобразование. Например для поворота изображения вокруг его центра матрица поворота будет вычисляться следующим образом:

$$T1 = \begin{bmatrix} 1 & 0 & -(I.cols - 1)/2 \\ 0 & 1 & -(I.rows - 1)/2 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.18}$$

$$T2 = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.19}$$

$$T3 = \begin{bmatrix} 1 & 0 & (I.cols - 1)/2 \\ 0 & 1 & (I.rows - 1)/2 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.20}$$

где матрица `T1` — это прямой сдвиг для совмещения точки $(0, 0)$ с центром изображения, `T2` — матрица преобразования вращения на угол φ, а `T3` — обратный сдвиг. Полная матрица преобразования `T` вычисляется путем матричного умножения этих трех матриц преобразования.

$$T = T3 \cdot T2 \cdot T1 \tag{3.21}$$

Учитывая то, что в библиотеке OpenCV используются только первые две строки матрицы для аффинного преобразования, требуется убрать последнюю строку после умножения промежуточных квадратных матриц преобразования.

Листинг 3.13. Вычисление матрицы преобразования для поворота изображения на угол $\varphi = 17°$ вокруг центра изображения с использованием библиотеки OpenCV и языка программирования C++.

```
double phi = 17.0 * M_PI / 180;
Mat T1 = (Mat_<double>(3, 3) <<
          1, 0, -(I.cols - 1) / 2.0,
          0, 1, -(I.rows - 1) / 2.0,
          0, 0, 1);
Mat T2 = (Mat_<double>(3, 3) <<
          cos(phi), -sin(phi), 0,
          sin(phi), cos(phi), 0,
          0, 0, 1);
Mat T3 = (Mat_<double>(3, 3) <<
          1, 0, (img.cols - 1) / 2.0,
          0, 1, (img.rows - 1) / 2.0,
          0, 0, 1);
Mat T = Mat(T3 * T2 * T1, Rect(0, 0, 3, 2));
```

Листинг 3.14. Вычисление матрицы преобразования для поворота изображения на угол $\varphi = 17°$ вокруг центра изображения с использованием библиотеки OpenCV и языка программирования Python.

```
phi = 17.0 * math.pi / 180
T1 = np.float32(
  [[1, 0, -(cols - 1) / 2.0],
  [0, 1, -(rows - 1) / 2.0],
  [0, 0, 1]])
T2 = np.float32(
  [[math.cos(phi), -math.sin(phi), 0],
  [math.sin(phi), math.cos(phi), 0],
  [0, 0, 1]])
T3 = np.float32(
  [[1, 0, (cols - 1) / 2.0],
  [0, 1, (rows - 1) / 2.0],
  [0, 0, 1]])
T = np.matmul(T3, np.matmul(T2, T1))[0:2, :]
```

В библиотеке OpenCV также присутствует функция `getRotationMatrix2D()`, предназначенная для вычисления матрицы преобразования для поворота изображения против часовой стрелки на произвольный угол вокруг произвольной точки изображения одновременно с масштабированием изображения. Например, вычисление матрицы преобразования для поворота изображения на угол $\varphi = 17°$ вокруг центра изображения с использованием языка программирования C++ будет выглядеть следующим образом:

```
double phi = 17.0;
T = cv::getRotationMatrix2D(
  Point2f(float((I.cols - 1) / 2.0),
          float((I.rows - 1) / 2.0)), -phi, 1);
```

А с использованием языка программирования Python:

```
phi = 17.0
T = cv2.getRotationMatrix2D(
  ((cols - 1) / 2.0, (rows - 1) / 2.0), -phi, 1)
```

Аффинное отображение — это отображение, при котором параллельные прямые переходят в параллельные прямые, пересекающиеся в пересекающиеся, скрещивающиеся в скрещивающиеся; сохраняются отношения длин отрезков, лежащих на одной прямой (или на параллельных прямых), и отношения площадей фигур.

Рис. 3.2 — Конформные преобразования, верхний ряд слева направо: исходное изображение, сдвиг, отражение, вращение; нижний ряд: однородное масштабирование.

Базовыми преобразованиями являются конформные преобразования, скос и неоднородное масштабирование. Произвольное аффинное преобразование можно получить при помощи последовательного произведения матриц базовых преобразований. В непрерывной геометрии любое аффинное преобразование имеет обратное аффинное преобразование, а произведение прямого и обратного дает единичное преобразование, которое оставляет все точки на месте. Аффинные преобразования являются подмножеством проекционных преобразований.

В библиотеке OpenCV присутствует функция `getAffineTransform()`, предназначенная для вычисления матрицы произвольного аффинного преобразования, заданного набором координат исходных точек изображения $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ и их координат после преобразования $(x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3)$. Программная реализация вычисления матрицы преобразования для произвольного аффинного преобразования с использованием языка программирования C++ будет выглядеть следующим образом:

```
vector <Point2f> pts_src =
```

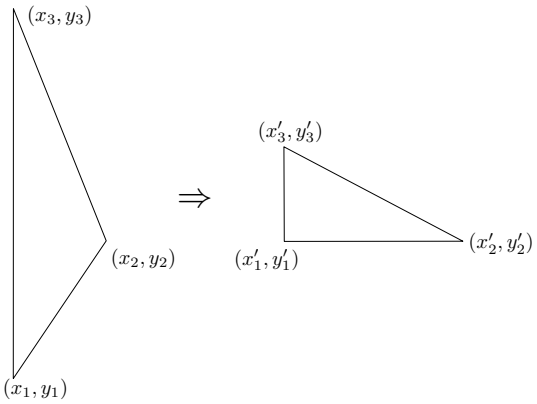

Рис. 3.3 — Аффинное отображение.

```
  { {50, 300}, {150, 200}, {50, 50} };
vector <Point2f> pts_dst =
  { {50, 200}, {250, 200}, {50, 10} };
T = cv::getAffineTransform(pts_src, pts_dst);
```

А с использованием языка программирования Python:

```
pts_src = np.float32([[50, 300], [150, 200], [50, 50]])
pts_dst = np.float32([[50, 200], [250, 200], [50, 100]])
T = cv2.getAffineTransform(pts_src, pts_dst)
```

Далее, вычисленная матрица может быть применена с использованием функции `warpAffine()`.

Скос изображения

Система уравнений (3.6) и матрица преобразования координат T в случае *скоса* изображения $A = E = 1$, $B = s$, $C = D = F = 0$ примут вид:

$$\begin{cases} x' = x + sy, \\ y' = y, \end{cases} \quad \Rightarrow T = \begin{bmatrix} 1 & 0 & 0 \\ s & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.22}$$

Листинг 3.15. Скос изображения в среде MATLAB, $s = 0.3$.

```
T = [1 0 0; 0.3 1 0; 0 0 1];
tform = affine2d(T);
I_bevel = imwarp(I, tform);
```

В случае использования библиотеки OpenCV матрица преобразования для скосв изображения определяется следующим образом:

$$T = \begin{bmatrix} 1 & s & 0 \\ 0 & 1 & 0 \end{bmatrix}, \tag{3.23}$$

Листинг 3.16. Скос изображения с использованием библиотеки OpenCV и языка программирования C++, $s = 0.3$.

```
double s = 0.3;
Mat T = (Mat_<double>(2, 3) <<
          1, s, 0,
          0, 1, 0);
Mat I_bevel;
warpAffine(I, I_bevel, T,
  Size(I.cols, I.rows));
```

Листинг 3.17. Скос изображения с использованием библиотеки OpenCV и языка программирования Python, $s = 0.3$.

```
s = 0.3
T = np.float32([[1, s, 0], [0, 1, 0]])
I_bevel = cv.warpAffine(I, T, (cols, rows))
```

Кусочно-линейные преобразования

Кусочно-линейное отображение — это отображение, при котором изображение разбивается на части, а затем к каждой из этих частей применяются различные линейные преобразования.

Листинг 3.18. Пример кусочно-линейного отображения в среде MATLAB — левая половина изображения остается без изменений, а правая растягивается в два раза вдоль оси Ox.

```
imid = round(size(I,2) / 2);
I_left = I(:, 1:imid, :);
stretch = 2;
I_right = I(:, (imid + 1:end), :);
T = [stretch 0 0; 0 1 0; 0 0 1];
```

```
6   tform = affine2d(T);
7   I_scale = imwarp(I_right, tform);
8   I_piecewiselinear = [I_left I_scale];
```

При использовании библиотеки OpenCV кусочно-линейные преобразования выполняются с использованием специального типа изображений, называемых ROI — Region of Interest, которые создаются ограничением области исходного изображения. Этот тип изображений имеет общие данные с исходным, но при этом может использоваться аналогично с обычным изображением. В C++ такой объект создается с помощью операции `()` и передачи области в качестве аргумента. Например, для создания фрагмента изображения прямоугольной формы необходимо передать результат работы функции `cv::Rect()`, как показано в следующем примере.

Листинг 3.19. Пример кусочно-линейного отображения с использованием библиотеки OpenCV и языка программирования C++ — левая половина изображения остается без изменений, а правая растягивается в два раза вдоль оси Ox.

```
1   double stretch = 2;
2   Mat T = (Mat_<double>(2, 3) <<
3            stretch, 0, 0,
4            0, 1, 0);
5   Mat I_piecewiselinear = I.clone();
6   Mat I_right = I_piecewiselinear(
7     Rect(int(I.cols / 2), 0,
8     I.cols - int(I.cols / 2), I.rows));
9   warpAffine(I_right, I_right, T,
10    Size(I.cols - int(I.cols / 2), I.rows));
```

При использовании языка программирования Python аналогичные объекты создаются средствами индексации массивов NumPy.

Листинг 3.20. Пример кусочно-линейного отображения с использованием библиотеки OpenCV и языка программирования Python — левая половина изображения остается без изменений, а правая растягивается в два раза вдоль оси Ox.

```
1   stretch = 2
2   T = np.float32([[stretch, 0, 0], [0, 1, 0]])
3   I_piecewiselinear = I.copy()
```

```
4   I_piecewiselinear[:, int(cols / 2):, :] = \
5     cv.warpAffine(
6     I_piecewiselinear[:, int(cols / 2):, :],
7     T, (cols - int(cols / 2), rows))
```

Рис. 3.4 — Слева — скос, справа — кусочно-линейное растяжение.

Нелинейные преобразования

При рассмотрении геометрических преобразований предполагается, что изображения получены при помощи идеальной модели камеры. В действительности формирование изображений сопровождается различными нелинейными искажениями, см. рис. 3.5. Для их коррекции используются различные нелинейные функции.

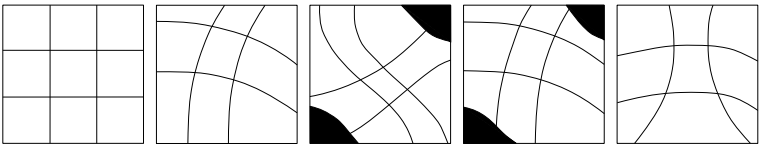

Рис. 3.5 — Примеры нелинейных искажений.

Проекционное преобразование

Проекционное отображение — это отображение, при котором прямые линии остаются прямыми линиями, однако геометрия фигуры может быть нарушена, т.к. данное отображение в общем случае не сохраняет параллельности линий. Свойством, сохраняющимся при проективном преобразовании, является *коллинеарность* точек: три точки, лежащие на одной прямой (коллинеарные), после преобразования остаются на одной прямой. Проекционное отображение может быть как *параллельным* (изменяется масштаб), так и *проективным* (изменяется геометрия фигуры).

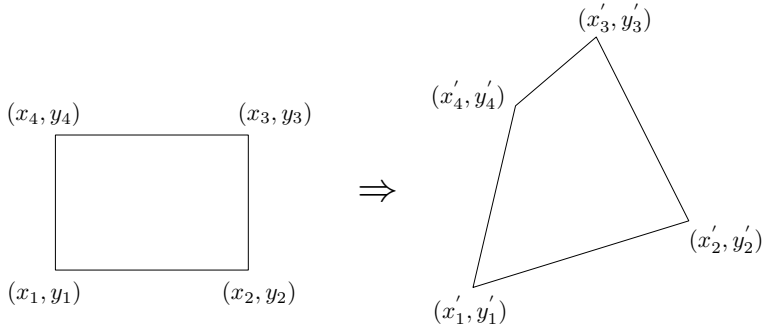

Рис. 3.6 — Проекционное проективное отображение: прямые остались прямыми, но параллельные отобразились в скрещивающиеся.

В случае проективного отображения точки трехмерной сцены $\mathbf{P}^3$ проецируются на двумерную плоскость $\mathbf{P}^2$ изображения. Такое преобразование $\mathbf{P}^3 \to \mathbf{P}^2$ отображает евклидову точку сцены $P = (x,y,z)$ (в однородных координатах (x',y',z',w)) в точку изображения $X = (x,y)$ (в однородных координатах (x',y',w)). Для нахождения декартовых координат точек из однородных координат воспользуемся следующими соотношениями: $P = (\frac{x'}{w}, \frac{y'}{w}, \frac{z'}{w})$ — координаты вектора $\mathbf{P}$ и $X = (\frac{x'}{w}, \frac{y'}{w})$ — координаты вектора $\mathbf{X}$. Подставляя в (3.2) $w = 1$, для вектора $\mathbf{X}$ получим систему уравнений:

$$\begin{cases} x' = \dfrac{Ax + By + C}{Gx + Hy + I}, \\ y' = \dfrac{Dx + Ey + F}{Gx + Hy + I}, \end{cases} \Rightarrow T = \begin{bmatrix} A & B & C \\ D & E & F \\ G & H & 1 \end{bmatrix}. \tag{3.24}$$

Из-за нормирования координат на w' в общем случае проекционное отображение является нелинейным.

Листинг 3.21. Пример проективного отображения в среде MATLAB при $A = 1.1$, $B = 0.35$, $C = F = 0$, $D = 0.2$, $E = 1.1$, $G = 0.00075$, $H = 0.0005$, $I = 1$.

```
1   T = [1.1 0.35 0; 0.2 1.1 0; 0.00075 0.0005 1];
```

```
2   tform = projective2d(T);
3   I_projective = imwarp(I, tform);
```

Листинг 3.22. Пример проективного отображения с использованием библиотеки OpenCV и языка программирования C++ при $A = 1.1$, $B = 0.35$, $C = F = 0$, $D = 0.2$, $E = 1.1$, $G = 0.00075$, $H = 0.0005$, $I = 1$.

```
1   Mat T = (Mat_<double>(3, 3) <<
2            1.1, 0.2, 0.00075,
3            0.35, 1.1, 0.0005,
4            0, 0, 1);
5   Mat I_projective;
6   cv::warpPerspective(I, I_projective, T,
7     Size(I.cols, I.rows));
```

Листинг 3.23. Пример проективного отображения с использованием библиотеки OpenCV и языка программирования Python при $A = 1.1$, $B = 0.35$, $C = F = 0$, $D = 0.2$, $E = 1.1$, $G = 0.00075$, $H = 0.0005$, $I = 1$.

```
1   T = np.float32(
2     [[1.1, 0.2, 0.00075],
3     [0.35, 1.1, 0.0005],
4     0, 0, 1]])
5   I_projective = cv2.warpPerspective(I, T,
6     (cols, rows))
```

В библиотеке OpenCV присутствует функция `getPerspectiveTransform()`, предназначенная для вычисления матрицы произвольного проекционного отображения, заданного набором координат исходных точек изображения $(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4)$ и их координат после преобразования $(x'_1, y'_1), (x'_2, y'_2), (x'_3, y'_3), (x'_4, y'_4)$. Программная реализация вычисления матрицы преобразования для произвольного проекционного отображения с использованием языка программирования C++ будет выглядеть следующим образом:

```
vector <Point2f> pts_src =
  { {50, 461}, {461, 461}, {461, 50}, {50, 50} };
vector <Point2f> pts_dst =
```

```
  { {50, 461}, {461, 440}, {450, 10}, {100, 50} };
T = cv::getPerspectiveTransform(pts_src, pts_dst);
```

А с использованием языка программирования Python:

```
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 = cv2.getPerspectiveTransform(pts_src, pts_dst)
```

Полиномиальное преобразование

Полиномиальное отображение — это отображение исходного изображения с помощью полиномов. В данном случае матрица преобразования координат T будет содержать коэффициенты полиномов соответствующих порядков для координат x и y. Например, в случае полиномиального преобразования *второго порядка* система уравнений 3.6 примет вид:

$$\begin{cases} x' = a_1 + a_2 x + a_3 y + a_4 x^2 + a_5 xy + a_6 y^2, \\ y' = b_1 + b_2 x + b_3 y + b_4 x^2 + b_5 xy + b_6 y^2, \end{cases} \tag{3.25}$$

где x, y — координаты точек в одной системе координат; x', y' — координаты этих точек в другой системе координат; $a_1 \dots a_6$, $b_1 \dots b_6$ — коэффициенты преобразования.

Листинг 3.24. Пример полиномиального отображения с заданной матрицей преобразования T и в котором не используются предопределенные функции MATLAB. Результирующее преобразованное изображение не будет содержать интерполированных пикселей.

```
1 [numRows, numCols, Layers] = size(I);
2 T = [0 0; 1 0; 0 1; 0.00001 0;
3     0.002 0; 0.001 0];
4 for k=1:1:Layers
5     for y=1:1:numCols
6         for x=1:1:numRows
7             xnew = round(T(1,1)+T(2,1)*x+...
8                 T(3,1)*y+T(4,1)*x^2+...
9                 T(5,1)*x*y+T(6,1)*y^2);
```

```
10              ynew = round(T(1,2)+T(2,2)*x+...
11                  T(3,2)*y+T(4,2)*x^2+...
12                  T(5,2)*x*y+T(6,2)*y^2);
13              I_polynomial(xnew,ynew,k) = ...
14                  I(x,y,k);
15          end
16      end
17  end
```

Листинг 3.25. Пример полиномиального отображения с заданной матрицей преобразования T с использованием библиотеки OpenCV и языка программирования C++. Результирующее преобразованное изображение не будет содержать интерполированных пикселей.

```
1   const double T[2][6] =
2     { { 0, 1, 0, 0.00001, 0.002, 0.002 },
3     { 0, 0, 1, 0, 0, 0 } };
4   Mat I_polynomial;
5   // Convert to floating points
6   if (I.depth() == CV_8U)
7     I.convertTo(I_polynomial, CV_32F,
8       1.0 / 255);
9   else
10    I_polynomial = I;
11  // We need to process layers separately
12  vector<Mat> I_BGR;
13  cv::split(I_polynomial, I_BGR);
14  for (int k = 0; k < I_BGR.size(); k++)
15  {
16    I_polynomial =
17    Mat::zeros(I_BGR[k].rows, I_BGR[k].cols,
18      I_BGR[k].type());
19    for (int x = 0; x < I_BGR[k].cols; x++)
20      for (int y = 0; y < I_BGR[k].rows; y++)
21      {
22        int xnew = int(round(T[0][0] +
23          x * T[0][1] + y * T[0][2] +
24          x * x * T[0][3] + x * y * T[0][4] +
```

```
25            y * y * T[0][5]));
26          int ynew = int(round(T[1][0] +
27            x * T[1][1] + y * T[1][2] +
28            x * x * T[1][3] + x * y * T[1][4] +
29            y * y * T[1][5]));
30          if (xnew >= 0 && xnew < I_BGR[k].cols
31           && ynew >= 0 && ynew < I_BGR[k].rows)
32            I_polynomial.at<float>(ynew, xnew) =
33              I_BGR[k].at<float>(y, x);
34          }
35          I_BGR[k] = I_polynomial;
36    }
37    // Merge back
38    cv::merge(I_BGR, I_polynomial);
39    // Convert back to uint if needed
40    if (I.depth() == CV_8U)
41      I_polynomial.convertTo(I_polynomial,
42        CV_8U, 255);
```

Листинг 3.26. Пример полиномиального отображения с заданной матрицей преобразования T с использованием библиотеки OpenCV и языка программирования Python. Результирующее преобразованное изображение не будет содержать интерполированных пикселей.

```
1   T = np.array([[0, 0], [1, 0], [0, 1],
2     [0.00001, 0], [0.002, 0], [0.001, 0]])
3   I_polynomial = np.zeros(I.shape, I.dtype)
4   x, y = np.meshgrid(np.arange(cols),
5                      np.arange(rows))
6   # Calculate all new X and Y coordinates
7   xnew = np.round(T[0, 0] + x * T[1, 0] +
8     y * T[2, 0] + x * x * T[3, 0] +
9     x * y * T[4, 0] +
10    y * y * T[5, 0]).astype(np.float32)
11  ynew = np.round(T[0, 1] + x * T[1, 1] +
12    y * T[2, 1] + x * x * T[3, 1] +
13    x * y * T[4, 1] +
14    y * y * T[5, 1]).astype(np.float32)
```

```
15    # Calculate mask of valid indexes
16    mask = np.logical_and(
17      np.logical_and(xnew >= 0, xnew < cols),
18      np.logical_and(ynew >= 0, ynew < rows))
19    # Apply reindexing
20    if img.ndim == 2:
21      I_polynomial[ynew[mask].astype(int),
22        xnew[mask].astype(int)] = \
23        img[y[mask], x[mask]]
24    else:
25      I_polynomial[ynew[mask].astype(int),
26        xnew[mask].astype(int), :] = \
27        img[y[mask], x[mask], :]
```

Следует отметить, что при использовании языка программирования Python попиксельная работа с изображением крайне неэффективна. Вместо этого необходимо вычислить новые координаты для каждого пикселя исходного изображения и применить их с использованием индексации NumPy. Для начального формирования массивов переиндексации используется специальный конструктор `numpy.meshgrid()`. Он создает два массива координат X и Y для каждого пикселя изображения. Затем эти координаты преобразуются и используются для сопоставления координат новых пикселей изображения со старыми. Матрица `mask` используется для исключения пикселей, вышедших за пределы допустимого диапазона после преобразования.

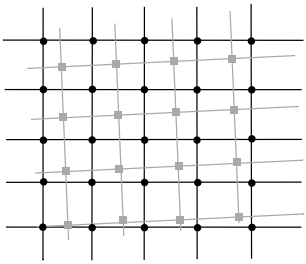

Рис. 3.7 — Смещение координат пикселей в преобразованном изображении.

Как правило, при нелинейном искажении коэффициенты $a_1 \dots a_6$, $b_1 \dots b_6$ и матрица преобразования координат T *неизвестны*. В случае, когда помимо искаженного изображения доступно исходное, можно вычислить коэффициенты преобразования, найдя пары соответствующих точек на исходном и преобразованном изображениях. Число минимально необходимых пар точек вычисляется по формуле:

$$t_{min} = \frac{(n+1)(n+2)}{2}, \tag{3.26}$$

где n — порядок преобразования.

Если имеет место полиномиальное искажение *второго порядка*, то согласно (3.26) необходимо знать координаты хотя бы *шести пар* соответствующих точек до и после трансформации: $(x_1, y_1) \dots (x_6, y_6)$ и $(x'_1, y'_1) \dots (x'_6, y'_6)$. Согласно (3.25) запишем уравнения для вычисления коэффициентов в матричном виде:

$$\begin{bmatrix} a_1 \\ \dots \\ a_6 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 & x_1^2 & x_1 y_1 & y_1^2 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_6 & y_6 & x_6^2 & x_6 y_6 & y_6^2 \end{bmatrix}^{-1} \begin{bmatrix} x'_1 \\ \dots \\ x'_6 \end{bmatrix}, \tag{3.27}$$

$$\begin{bmatrix} b_1 \\ \dots \\ b_6 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 & x_1^2 & x_1 y_1 & y_1^2 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & x_6 & y_6 & x_6^2 & x_6 y_6 & y_6^2 \end{bmatrix}^{-1} \begin{bmatrix} y'_1 \\ \dots \\ y'_6 \end{bmatrix}. \tag{3.28}$$

Для интерактивного указания одинаковых точек на исходном и преобразованном изображениях можно воспользоваться функцией MATLAB `cpselect('source.jpg','transformed.jpg')`, которая возвращает два массива `movingPoints` и `fixedPoints` с координатами преобразованного и исходного изображений соответственно.

Синусоидальное искажение

В качестве еще одного из примеров нелинейного преобразования можно рассмотреть гармоническое искажение изображения.

Листинг 3.27. Пример синусоидального искажения изображения в среде MATLAB.

```
[xi,yi] = meshgrid(1:ncols,1:nrows);
imid = round(size(I,2)/2);
u = xi + 20*sin(2*pi*yi/90);
v = yi;
tmap_B = cat(3,u,v);
resamp = makeresampler('linear','fill');
I_sinusoid = tformarray(I,[],resamp,...
    [2 1],[1 2],[],tmap_B,.3);
```

Листинг 3.28. Пример синусоидального искажения изображения с использованием библиотеки OpenCV и языка программирования C++.

```
  Mat u = Mat::zeros(I.rows, I.cols, CV_32F);
  Mat v = Mat::zeros(I.rows, I.cols, CV_32F);
  for (int x = 0; x < I.cols; x++)
    for (int y = 0; y < I.rows; y++)
    {
      u.at<float>(y, x) = float(x +
        20 * sin(2 * M_PI * y / 90));
      v.at<float>(y, x) = float(y);
    }
  Mat I_sinusoid;
  remap(I, I_sinusoid, u, v, INTER_LINEAR);
```

Листинг 3.29. Пример синусоидального искажения изображения с использованием библиотеки OpenCV и языка программирования Python.

```
  u, v = np.meshgrid(np.arange(cols),
    np.arange(rows))
  u = u + 20 * np.sin(2 * math.pi * v / 90)
  I_sinusoid = \
    cv2.remap(I, u.astype(np.float32),
    v.astype(np.float32), cv2.INTER_LINEAR)
```

Функция библиотеки OpenCV `remap()` преобразует изображение в соответствии с новыми координатами пикселей, заданными для каждого пикселя исходного изображения.

Рис. 3.8 — Слева — полиномиальное искажение, в центре — проективное, справа — синусоидальное.

Коррекция дисторсии

При формировании изображения оптической системой на нем может возникнуть *дисторсия*. *Дисторсия* — это оптическое искажение, выражающееся в искривлении прямых линий. Световые лучи, проходящие через центр линзы, сходятся в точке, расположенной дальше от линзы, чем лучи, проходящие через ее края. Прямые линии искривляются за исключением тех, которые лежат в одной плоскости с оптической осью. Например, изображение квадрата, центр которого пересекает оптическая ось, имеет вид подушки (*подушкообразная дисторсия*) при положительной дисторсии и вид бочки (*бочкообразная дисторсия*) при отрицательной дисторсии (рис. 3.9).

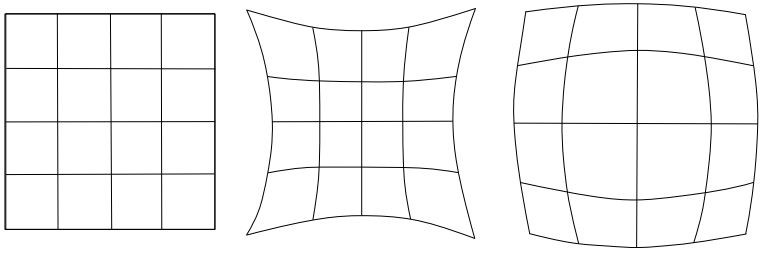

Рис. 3.9 — Примеры дисторсий. Слева — исходное изображение, по центру — подушкообразная дисторсия, справа — бочкообразная.

Пусть $\mathbf{r} = (x,y)$ — вектор, задающий две координаты в плоскости, расположенной перпендикулярно оптической оси. Для идеального изображения все лучи, вышедшие из этой точки и прошедшие через оптическую систему, попадут в точку изображения с координатами $\mathbf{R}$, которые определяются по формуле:

$$\mathbf{R} = b_0\mathbf{r}, \tag{3.29}$$

где b_0 — коэффициент линейного увеличения.

Если присутствует дисторсия более высокого порядка (для осесимметричных оптических систем дисторсия может быть только нечетных порядков: третьего, пятого, седьмого и т.д.), то в выражение (3.29) необходимо добавить соответствующие слагаемые:

$$\mathbf{R} = b_0\mathbf{r} + F_3 r^2\mathbf{r} + F_5 r^4\mathbf{r} + \ldots, \tag{3.30}$$

где r — длина вектора $\mathbf{r}$; $F_i, i = 3,5,\ldots,n$ — коэффициенты дисторсии n-го порядка, которые вносят наибольший вклад в искажение формы изображения. При дисторсии третьего порядка, если коэффициент F_3 имеет тот же знак, что и b_0: $\mathrm{sign}(F_3) = \mathrm{sign}(b_0)$, возникает подушкообразное искажение, в противном случае — бочкообразное.

Для коррекции дисторсии используется подход, описанный выше для проективного отображения. Используется изображение регулярной сетки и его искаженное изображение, находятся пары точек на этих изображениях и вычисляются коэффициенты корректирующего преобразования.

Бочкообразная дисторсия

Листинг 3.30. Пример наложения бочкообразной дисторсии пятого порядка на исходное изображение в среде MATLAB.

```
[xi,yi] = meshgrid(1:ncols,1:nrows);
imid = round(size(I,2)/2);
xt = xi(:) - imid;
yt = yi(:) - imid;
[theta,r] = cart2pol(xt,yt);
F3 = .000001;
F5 = .0000012;
R = r + F3 * r.^3 + F5 * r.^5;
```

```
 9 [ut,vt] = pol2cart(theta, R);
10 u = reshape(ut, size(xi)) + imid;
11 v = reshape(vt, size(yi)) + imid;
12 tmap_B = cat(3, u, v);
13 resamp = makeresampler('linear','fill');
14 I_barrel = tformarray(I,[],resamp,...
15     [2 1],[1 2],[],tmap_B,.3);
```

Для формирования сетки изображения используется функция `meshgrid()`. Затем на строках 2-4 происходит получение отцентрированного набора координат всех пикселей. После этого все координаты функцией `cart2pol()` переводятся из декартовой системы координат в полярную, для более удобного применения формулы (3.30) через радиус-векторы каждой точки. Затем координаты пикселей преобразуются при помощи функций `pol2cart()` и `reshape()` обратно в декартовы координаты. Из итоговой сетки формируется трехмерная матрица преобразования `tmap_B` для функции `tformarray`, которая производит само преобразование исходного изображения по заданной сетке.

Листинг 3.31. Пример наложения бочкообразной дисторсии пятого порядка на исходное изображение с использованием библиотеки OpenCV и языка программирования C++.

```
1   // Do the meshgrid() routine
2   Mat xi, yi;
3   std::vector<float> t_x, t_y;
4   for (int i = 0; i < img.cols; i++)
5     t_x.push_back(float(i));
6   for (int i = 0; i < img.rows; i++)
7     t_y.push_back(float(i));
8   cv::repeat(Mat(t_x).reshape(1, 1),
9     I.rows, 1, xi);
10  cv::repeat(Mat(t_y).reshape(1, 1).t(),
11    1, I.cols, yi);
12  // Shift and normalize grid
13  double xmid = xi.cols / 2.0;
14  double ymid = xi.rows / 2.0;
15  xi -= xmid;
16  xi /= xmid;
```

```
17  yi -= ymid;
18  yi /= ymid;
19  // Convert to polar and do transformation
20  Mat r, theta;
21  cartToPolar(xi, yi, r, theta);
22  double F3(0.1), F5(0.12);
23  Mat r3, r5;
24  pow(r, 3, r3); // r3 = r^3
25  pow(r, 5, r5); // r5 = r^5
26  r += r3 * F3;
27  r += r5 * F5;
28  // Undo conversion, normalization and shift
29  Mat u, v;
30  polarToCart(r, theta, u, v);
31  u *= xmid;
32  u += xmid;
33  v *= ymid;
34  v += ymid;
35  // Do remapping
36  Mat I_barrel;
37  remap(I, I_barrel, u, v, INTER_LINEAR);
```

Листинг 3.32. Пример наложения бочкообразной дисторсии пятого порядка на исходное изображение с использованием библиотеки OpenCV и языка программирования Python.

```
1  # Create mesh grid for X, Y
2  xi, yi = np.meshgrid(np.arange(cols),
3    np.arange(rows))
4  # Shift and normalize grid
5  xmid = cols / 2.0
6  ymid = rows / 2.0
7  xi = xi - xmid
8  yi = yi - ymid
9  # Convert to polar and do transformation
10 r, theta = cv.cartToPolar(xi / xmid, yi / ymid)
11 F3 = 0.1
12 F5 = 0.12
13 r = r + F3 * r ** 3 + F5 * r ** 5
```

```
14    # Undo conversion, normalization and shift
15    u, v = cv.polarToCart(r, theta)
16    u = u * xmid + xmid;
17    v = v * ymid + ymid;
18    # Do remapping
19    I_barrel = \
20      cv2.remap(I, u.astype(np.float32),
21      v.astype(np.float32), cv.INTER_LINEAR)
```

Подушкообразная дисторсия

Листинг 3.33. Пример наложения подушкообразной дисторсии третьего порядка на исходное изображение.

```
1 F3 = -0.003;
2 R = r + F3 * r.^2;
```

Рис. 3.10 — Слева — исходное изображение, в центре — бочкообразная дисторсия, справа — подушкообразная дисторсия.

«Сшивка» изображений

Геометрические преобразования можно использовать, например, для построения мозаики из нескольких изображений. Мозаика («сшивка», «склейка») — это объединение двух или более изображений в единое целое, причем системы координат склеиваемых изображений могут отличаться из-за разного ракурса съемки, изменения положения камеры или движения самого объекта. Однако необходимо, чтобы оба изображения имели области перекрытия, т.е. на них присутствовали одинаковые объекты.

Основной задачей обработки таких изображений является приведение их в общую систему координат. В качестве общей системы

координат можно использовать систему первого изображения, тогда требуется найти преобразование координат всех пикселей второго изображения (x,y) в общую систему координат (x',y'). Если имеет место полиномиальное искажение, то для пересчета координат можно воспользоваться системой уравнений (3.25). В случае аффиного отображения система (3.25) примет вид:

$$\begin{cases} x' = a_1 + a_2 x + a_3 y, \\ y' = b_1 + b_2 x + b_3. \end{cases} \tag{3.31}$$

Поэтому необходимо найти лишь по 3 коэффициента преобразования по каждой координате:

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}^{-1} \begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix}, \tag{3.32}$$

$$\begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}^{-1} \begin{bmatrix} y'_1 \\ y'_2 \\ y'_3 \end{bmatrix}. \tag{3.33}$$

Для этого на обоих изображениях следует выбрать соответствующие пары точек (три пары в случае аффинного искажения и не менее шести пар в случае полиномиального искажения). Интерактивно выполнить эту операцию можно при помощи функции MATLAB `cpselect()`, либо воспользоваться специальными алгоритмами. Например, можно определить эти точки на основе коэффициента корреляции, который в MATLAB вычисляется с помощью функции `corr2()`.

Рассмотрим простой случай «склейки» двух неискаженных изображений, имеющих одинаковую ширину. Необходимо склеить их по вертикали, т.е. добавить к первому второе снизу. Однако, граница склейки неизвестна. Эту задачу можно реализовать при помощи корреляционного подхода.

Листинг 3.34. Пример «склейки» изображений в среде MATLAB. Считывание верхней и нижней картинок соответственно:

```
1 topPart = imread('itmoTop.jpg');
2 botPart = imread('itmoBot.jpg');
```

Для простоты определения коэффициента корреляции преобразуем цветные изображения в полутоновые:

```
topPartHT = im2double(rgb2gray(topPart));
botPartHT = im2double(rgb2gray(botPart));
```

Определим размеры полутоновых изображений и инициализируем промежуточные массивы для вычислений:

```
[numRows, numCols, Layers] = size(topPart);
[numRowsBot, numColsBot] = size(botPartHT);
botPartCorrHT = zeros(intersecPart,numCols);
topPartCorrHT = zeros(intersecPart,numCols);
correlationArray = [];
```

Определим верхнюю строку нижнего изображения для вычисления коффициента корреляции со строками верхнего:

```
intersecPart = 5;
for j = 1:1:numCols
    for i = 1:1:intersecPart
        botPartCorrHT(i,j) = botPartHT(i,j);
    end
end
```

Сравним полученную строку со строками верхнего изображения и вычислим коэффициент корреляции:

```
for j = 0:1:numRows-intersecPart
    for i = 1:1:intersecPart
        topPartCorrHT(i,:) = topPartHT(i+j,:);
    end
    correlationCoefficient = ...
        corr2(topPartCorrHT, botPartCorrHT);
    correlationArray = ...
        [correlationArray
        correlationCoefficient];
    correlationCoefficient = 0;
end
[M, I] = max(correlationArray);
```

Построим цветное «склеенное» изображение `result_img` с границей склейки на основе полученного индекса, соответствующего номеру строки с максимальным коэффициентом корреляции:

```
numRowsBotCorr = numRowsBot + I - 1;
for k = 1:1:Layers
    for j = 1:1:numCols
        for i = 1:I-1
            result_img(i,j,k) = ..
                topPart(i,j,k);
        end
        for i = I:1:numRowsBotCorr
            result_img(i,j,k) = ...
                botPart(i-I+1,j,k);
        end
    end
end
```

Листинг 3.35. Пример «склейки» изображений с использованием библиотеки OpenCV и языка программирования C++. Считывание верхней и нижней картинок соответственно:

```
Mat topPart =
  imread("itmoTop.jpg", IMREAD_COLOR);
Mat botPart =
  imread("itmoBot.jpg", IMREAD_COLOR);
```

Создадим шаблон для вычисления коэффициента корреляции как нижние 10 строк от верхнего изображения:

```
int templ_size = 10;
Mat templ = topPart(
  Rect(0, topPart.rows - templ_size - 1,
       topPart.cols, templ_size));
```

Вычислим коэффициенты корреляции шаблона с нижним изображением:

```
Mat res;
cv::matchTemplate(botPart, templ, res,
  TM_CCOEFF);
```

В качестве точки «склейки» выберем точку с максимальным значением коэффициента корреляции:

```
12   Mat res;  double min_val, max_val;
13   Point2i min_loc, max_loc;
14   minMaxLoc(res, &min_val, &max_val,
15     &min_loc, &max_loc);
```

Построим «склеенное» изображение `result_img` с границей склейки на основе полученного индекса:

```
16   Mat result_img =
17     Mat::zeros(topPart.rows + botPart.rows -
18                max_loc.y - templ_size,
19                topPart.cols, topPart.type());
```

Скопируем данные из верхней части изображения:

```
20   topPart.copyTo(result_img(Rect(0, 0,
21     topPart.cols, topPart.rows)));
```

И из нижней части:

```
22   botPart(Rect(0, max_loc.y + templ_size,
23     botPart.cols, botPart.rows -
24     max_loc.y - templ_size)).
25     copyTo(result_img(Rect(0, topPart.rows,
26     botPart.cols,
27     botPart.rows - max_loc.y - templ_size)));
```

Листинг 3.36. Пример «склейки» изображений с использованием библиотеки OpenCV и языка программирования Python. Считывание верхней и нижней картинок соответственно:

```
1   topPart = cv2.imread("itmoTop.jpg", \
2                        cv.IMREAD_COLOR)
3   botPart = cv2.imread("itmoBot.jpg", \
4                        cv.IMREAD_COLOR)
```

Создадим шаблон для вычисления коэффициента корреляции как нижние 10 строк от верхнего изображения и вычислим коэффициенты корреляции шаблона с нижним изображением:

```
5   # Match template
6   templ_size = 10
7   templ = topPart[-templ_size:, :, :]
8   res = cv2.matchTemplate(botPart, templ,
9     cv2.TM_CCOEFF)
```

В качестве точки «склейки» выберем точку с максимальным значением коэффициента корреляции:

```
10  min_val, max_val, min_loc,
11    max_loc = cv2.minMaxLoc(res)
```

Построим «склеенное» изображение `result_img` с границей склейки на основе полученного индекса:

```
12  result_img = np.zeros((topPart.shape[0] +
13    botPart.shape[0] - max_loc[1] -
14    templ_size, topPart.shape[1],
15    topPart.shape[2]), dtype = np.uint8)
16  result_img[0:topPart.shape[0], :, :] = \
17    topPart
18  result_img[topPart.shape[0]:, :, :] = \
19    botPart[max_loc[1] + templ_size:, :, :]
```

Автоматическая склейка изображений в библиотеке OpenCV

Библиотека OpenCV предоставляет специальный класс для автоматического «склеивания» изображений, который называется `Stitcher`. Он предназначен для автоматической «склейки» панорамных фотографий (параметр `cv::Stitcher::PANORAMA` в C++ и `cv2.Stitcher_PANORAMA` в Python), и для автоматической

«склейки» отсканированных изображений (параметр `cv::Stitcher::SCANS` в C++ и `cv2.Stitcher_SCANS` в Python). Работать с объектом `Stitcher` очень просто, что продемонстрировано в следующем примере программной реализации склейки трех фотографий в одно панорамное изображение.

Листинг 3.37. Пример автоматический «склейки» изображений с использованием библиотеки OpenCV и языка программирования C++. Создание объекта `Stitcher`:

```
1   Ptr<Stitcher> stitcher =
2     cv::Stitcher::create(Stitcher::PANORAMA);
```

Далее необходимо создать массив исходных изображений для «склейки»:

```
3   vector<Mat> imgs;
4   imgs.push_back(I_1);
5   imgs.push_back(I_2);
6   imgs.push_back(I_3);
```

И передать этот массив в метод `stitch()`:

```
7   Mat I_stitch;
8   Stitcher::Status status =
9     stitcher->stitch(imgs, I_stitch);
```

Листинг 3.38. Пример автоматический «склейки» изображений с использованием библиотеки OpenCV и языка программирования Python. Создание объекта `Stitcher`:

```
1   stitcher = \
2     cv2.Stitcher.create(cv.Stitcher_PANORAMA)
```

Далее необходимо создать массив исходных изображений для «склейки» и передать его в метод `stitch()`:

```
3   status, I_stitch = \
4     stitcher.stitch([I_1, I_2, I_3])
```

Значение `status`, возвращаемое методом `stitch()`, показывает результат «склейки». В случае успеха он равен `cv::Stitcher::OK` для C++ и `cv2.Stitcher_OK` для Python, а в `I_stitch` сохраняется «склеенное» изображение.

Порядок выполнения работы

1. *Простейшие геометрические преобразования.* Выбрать произвольное изображение. Выполнить над ним линейные и нелинейные преобразования (конформные, аффинные и проективные отображения).

2. *Коррекция дисторсии.* Выбрать произвольное изображение либо с подушкообразной, либо с бочкообразной дисторсией. Выполнить коррекцию изображения.

3. *«Склейка» изображений.* Выбрать два изображения (снимки с фотокамеры, фрагменты сканированного изображения и пр.), на которых имеется область пересечения. Выполнить коррекцию второго изображения для его перевода в систему координат первого; затем выполнить автоматическую «склейку» из двух изображений в одно.

Содержание отчета

1. Титульный лист.

2. Цель работы.

3. Теоретическое обоснование применяемых методов и функций геометрических преобразований.

4. Ход выполнения работы:

   (a) Исходные изображения;

   (b) Листинги программных реализаций;

   (c) Комментарии;

   (d) Результирующие изображения.

5. Выводы о проделанной работе.

Вопросы к защите лабораторной работы

1. Каким образом можно выполнить поворот изображения, не используя матрицу поворота?

2. Какое минимальное количество соответствующих пар точек необходимо задать на исходном и искаженном изображениях, если порядок преобразования $n = 4$?

3. После геометрического преобразования изображения могут появиться пиксели с неопределенными значениями интенсивности. С чем это связано и как решается данная проблема?