Практическая работа 6
Оценка положения объектов в трехмерном пространстве

Материалы для выполнения практической работы могут быть получены с помощью команды:

```
$ wget https://cv-course.demo.3divi.com/download/task6/practical
```

Задания рекомендуется выполнять последовательно в среде Google-Colab или Jupyter

Однородные координаты

Как уже обсуждалось на лекции, точка плоскости с декартовыми координатами (x, y) имеет однородные координаты (xw, yw, w) для любого ненулевого вещественного числа w. Для лучшего понимания связи между декартовыми и однородными координатами рассмотрим визуализацию этой взаимосвязи.

Для начала зададим функции, необходимые для отрисовки точек трехмерного пространства.

Код:

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

def get_default_colors():
    return ["blue", "green", "red", "magenta"]

def plot_3d_points(points, axes=None, color="blue", show=True):
    if axes is None:
        figure = plt.figure()
        axes = figure.add_subplot(projection='3d')
        axes.set_xlabel('X')
        axes.set_ylabel('Y')
        axes.set_zlabel('Z')
        x, y, z = np.array([[-10, 0, 0], [0, -10, 0], [0, 0, -10]])
        u, v, w = np.array([[20, 0, 0], [0, 20, 0], [0 , 0, 20]])
        axes.quiver(x, y, z, u, v, w, arrow_length_ratio=0.1,
color="black")
        axes.plot(points[0, :], points[1, :], points[2, :], marker='o',
```

```
linestyle='', color=color)
        if show:
            plt.show()
        return axes

def plot_3d_point_sets(point_sets, colors=None):
    if colors is None:
        colors = get_default_colors()
        axes = None
        for i, point_set in enumerate(point_sets):
        axes = plot_3d_points(point_set, axes=axes, color=colors[i],
show=i+1==len(point_sets))
```

Далее, зададим функцию для перевода координат наборов точек из декартовой системы координат в однородную. Точки будем рассматривать в виде векторов-столбцов, то есть одна точка двумерного пространства будет храниться в массиве размера 2×1, а набор из n точек - в массиве размера 2×n. Функция просто приписывает к координатам каждой точки еще одну единичную координату. Она будет корректно работать и для точек трехмерного пространства.

Код:

```
def to_homogeneous(points):
    new_coordinates = np.ones(points.shape[:-2] + (1,) + points.shape[-
1:], points.dtype)
    points = np.concatenate([points, new_coordinates], axis=-2)
    return points
```

Наконец, реализуем сам пример визуализации. Возьмем четыре двумерные точки с декартовыми координатами $(1;\ 1), (2;\ -0{,}5), (-1;\ 2)$ и $(-0{,}5;\ -1)$ и переведем их в однородную систему координат. Так как каждой двумерной точке соответствует бесконечное множество троек однородных координат, отличающихся только умножением каждой координаты из этой тройки на какое-то ненулевое число, то рассмотрим различные варианты параметра w, задающего это разницу. Будем использовать значения w от -10 до 10 с шагом 1, за исключением значения 0. Для каждой точки построим тройки ее однородных координат для всех

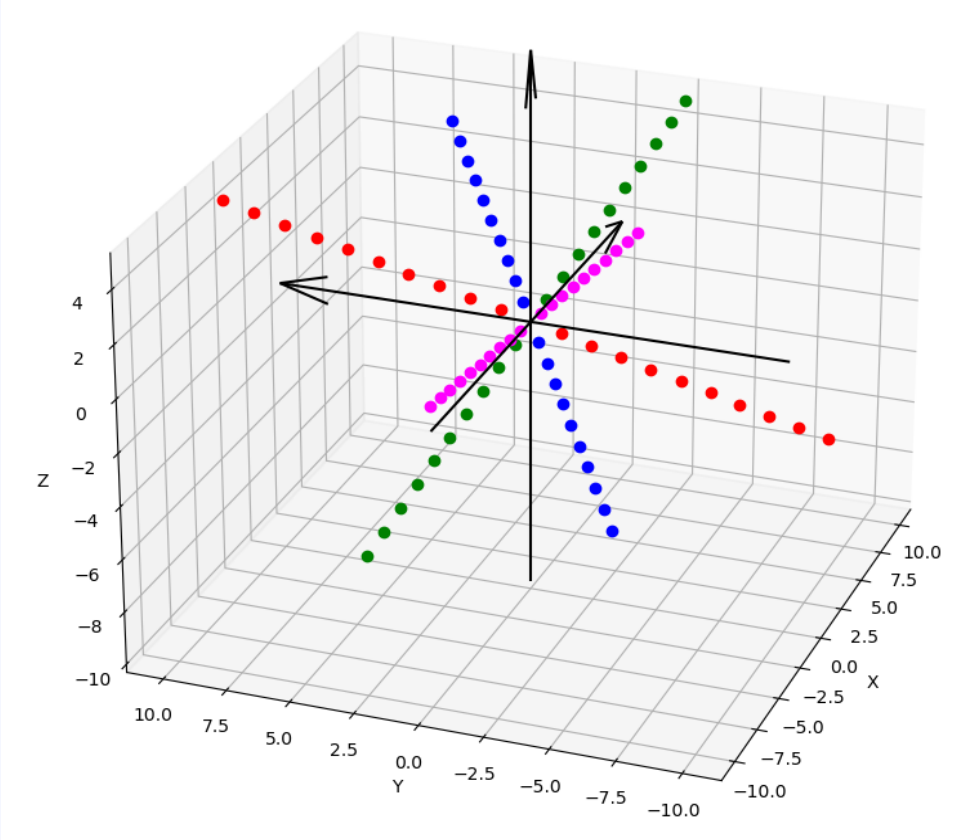
значений w из описанного набора. В результате для каждой двумерной точки из исходного набора получим набор точек трехмерного пространства, лежащих на одной прямой, проходящей через начало координат, причем разным исходным точкам соответствуют разные прямые.

Код:

```
def homogeneous_coordinates_example():
    points = np.array([[1, 2, -1, -0.5], [1, -0.5, 2, -1]])
    points = to_homogeneous(points)
    w_vals = np.concatenate([-0.5*(np.arange(10, 0, -1)),
0.5*np.arange(1, 11)])
    point_sets = (points[:, :, None]*w_vals[None, None,
:]).transpose([1, 0, 2])
    plot_3d_point_sets(point_sets)

homogeneous_coordinates_example()
```

Результат:



Матрицы преобразований в однородной системе координат

Реализуем функции, создающие матрицы различных преобразований для двумерных и трехмерных точек: параллельный перенос, поворот вокруг начала координат (2D) и координатных осей (3D), масштабирование и отражения относительно координатных осей (2D) или плоскостей (3D).

Код:

```
def get_translation_matrix(t_vector):
    t_vector = np.array(t_vector)
    Mt = np.eye(np.size(t_vector)+1)
    Mt[:-1, -1] = t_vector.flatten()
    return Mt

def get_rotation_matrix(angle, axis=None, _3d=False):
    if axis is not None:
        _3d = True
    m_base = np.array([[np.cos(angle), -np.sin(angle)], [np.sin(angle),
np.cos(angle)]])
    Mr = np.eye(4 if _3d else 3)
    if not _3d or axis == "z":
        row_col_nums = [0, 1]
    elif axis == "x":
        row_col_nums = [1, 2]
    elif axis == "y":
        row_col_nums = [0, 2]
    Mr[np.ix_(row_col_nums, row_col_nums)] = m_base
    if axis == "y":
        Mr = Mr.T
    return Mr

def get_scaling_matrix(scaling_vector):
    scaling_vector = np.array(scaling_vector)
    Ms = np.diag(np.concatenate([scaling_vector.flatten(), [1]]))
    return Ms

def get_reflection_matrix(reflection_flags):
    reflection_flags = np.array(reflection_flags)
    Mref = np.diag(np.concatenate([np.where(reflection_flags.flatten(),
-1, 1), [1]]))
    return Mref
```

Можно вывести матрицы, получаемые в результате работы этих функций.

Код:

```
Mt = get_translation_matrix([4, 2.5])
print(Mt)
Mt3 = get_translation_matrix([1.5, -3, 2])
print(Mt3)
Ms = get_scaling_matrix([4, 2])
print(Ms)
Ms3 = get_scaling_matrix([1.5, -3, 2])
print(Ms3)
Mref = get_reflection_matrix([False, True])
print(Mref)
Mref3 = get_reflection_matrix([True, False, True])
print(Mref3)
Mr = get_rotation_matrix(0.1)
print(Mr)
Mr3x = get_rotation_matrix(0.1, axis="x")
print(Mr3x)
Mr3y = get_rotation_matrix(0.1, axis="y")
print(Mr3y)
Mr3z = get_rotation_matrix(0.1, axis="z")
print(Mr3z)
```

Например, для первой матрицы, соответствующей преобразованию параллельного переноса точек двумерного пространства на вектор (4; 2,5), вывод будет выглядеть следующим образом:

Результат:

```
[[1.  0.  4. ]
 [0.  1.  2.5]
 [0.  0.  1. ]]
```

Задания и вопросы для самостоятельной работы:

- *Изучите остальные матрицы преобразований из примера выше и сравните их с матрицами, приведенными в лекции.*

Теперь возьмем какой-либо набор точек двумерного пространства и попробуем преобразовать его с помощью различных матриц преобразования.

Для начала приведем функции, необходимые для отрисовки точек двумерного пространства.

Код:

```
def plot_2d_points(points, axes=None, color="blue", show=True):
    if axes is None:
        figure = plt.figure()
        axes = figure.add_subplot()
        x_min, x_max, y_min, y_max = -14, 14, -14, 14
        axes.set(xlim=(x_min, x_max), ylim=(y_min, y_max),
aspect='equal')
        x_step, y_step = 2, 2
        x_ticks = np.arange(x_min, x_max+1, x_step)
        y_ticks = np.arange(y_min, y_max+1, y_step)
        axes.set_xticks(x_ticks[x_ticks != 0])
        axes.set_yticks(y_ticks[y_ticks != 0])
        axes.spines['bottom'].set_position('zero')
        axes.spines['left'].set_position('zero')
        axes.spines['top'].set_visible(False)
        axes.spines['right'].set_visible(False)

    plt.plot(points[0, :], points[1, :], marker='o', linestyle='',
color=color)
    if show:
        plt.show()
    return axes

def plot_2d_point_sets(point_sets, colors=None):
     if colors is None:
     colors = get_default_colors()
     axes = None
     for i, point_set in enumerate(point_sets):
     axes = plot_2d_points(point_set, axes=axes, color=colors[i],
show=i+1==len(point_sets))
```

Затем создадим набор точек двумерного пространства, переведем его в однородную систему координат, зададим матрицы двух преобразований (параллельного переноса на вектор (7; -3,5) и поворота на угол 45° против часовой стрелки вокруг начала координат), и преобразуем набор точек с помощью умножения на матрицы этих преобразований.

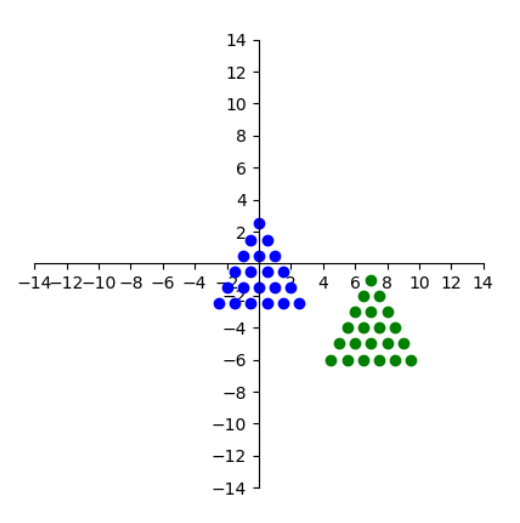
Код:

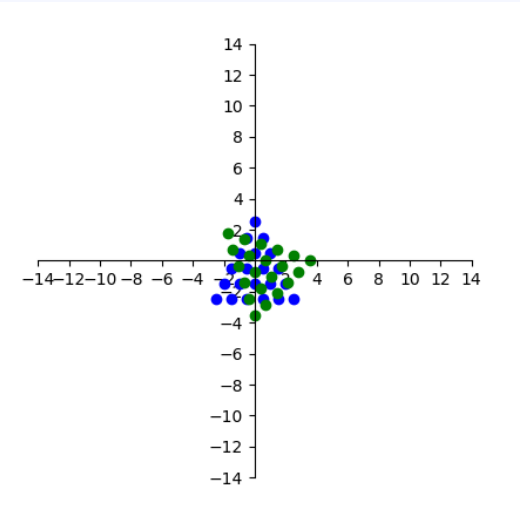
```
object_points = np.array([[j - 0.5*(i-1), 3.5-i] for i in range(1,
7) for j in range(i)]).T
object_points = to_homogeneous(object_points)

Mt = get_translation_matrix([7, -3.5])
Mr = get_rotation_matrix(np.pi/4)

object_points_translated = Mt @ object_points
plot_2d_point_sets([object_points, object_points_translated])
object_points_rotated = Mr @ object_points
plot_2d_point_sets([object_points, object_points_rotated])
```

Результат:





Задания и вопросы для самостоятельной работы:

- *Рассмотрите, как преобразуют точки матрицы других двумерных преобразований из рассмотренных ранее.*

Как уже говорилось на лекции, преобразования можно комбинировать при помощи перемножение матриц. Рассмотрим это свойство на примере.

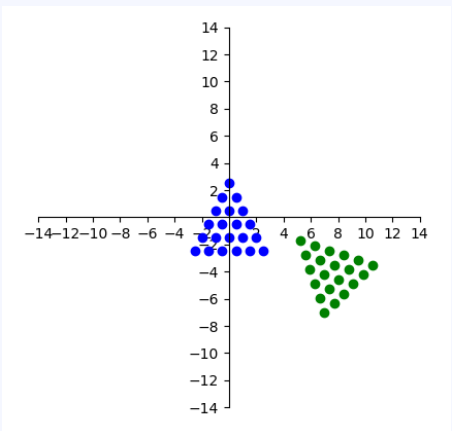
Возьмем набор двумерных точек и два преобразования, использовавшихся в предыдущем примере. Преобразуем набор точек двумя способами:

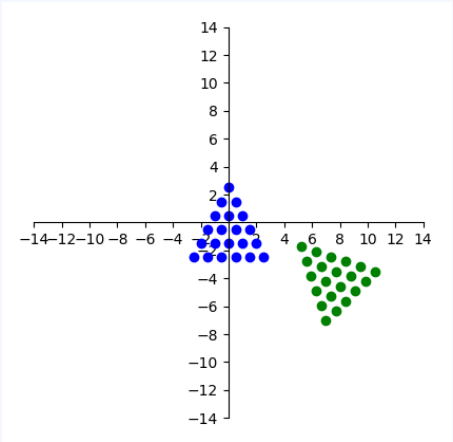
1. Сначала исходный набор точек повернем вокруг начала координат с помощью умножения на матрицу поворота, затем полученный новый набор точек преобразуем с помощью умножения на матрицу параллельного переноса.
2. Умножим матрицу поворота слева на матрицу параллельного переноса, и затем преобразуем исходный набор точек с помощью полученной матрицы преобразования.

Код:

```
object_points_t1 = Mr @ object_points
object_points_t2 = Mt @ object_points_t1
Mc = Mt @ Mr
object_points_t3 = Mc @ object_points
plot_2d_point_sets([object_points, object_points_t2])
plot_2d_point_sets([object_points, object_points_t3])
```

Результат:





Визуализации для обоих случаев получились одинаковые. Однако хотелось бы сравнить результаты более точно. Для этого посчитаем поэлементную разницу между наборами точек, полученными двумя рассмотренными способами, и найдем среди них наибольшую по модулю.

Код:

```
def get_max_abs_difference(a1, a2):
    return np.max(np.abs(a1-a2))

print(get_max_abs_difference(object_points_t2, object_points_t3))
```

Результат:

```
4.440892098500626e-16
```

Как видим, разница между соответствующими элементами двух массивов, вычисленных разными способами, очень мала, менее 10^{-15}. Эта разница получается только в результате погрешностей при вычислениях разными способами.

Важно помнить, что при комбинировании преобразований точек с помощью перемножения матриц этих преобразований матрицы нужно перемножать справа налево, то есть самая правая матрица из произведения должна соответствовать преобразованию, которое применяется раньше всех остальных, матрица слева от нее должна соответствовать преобразованию, применяющемуся сразу же после первого, и т.д.

Задания и вопросы для самостоятельной работы:

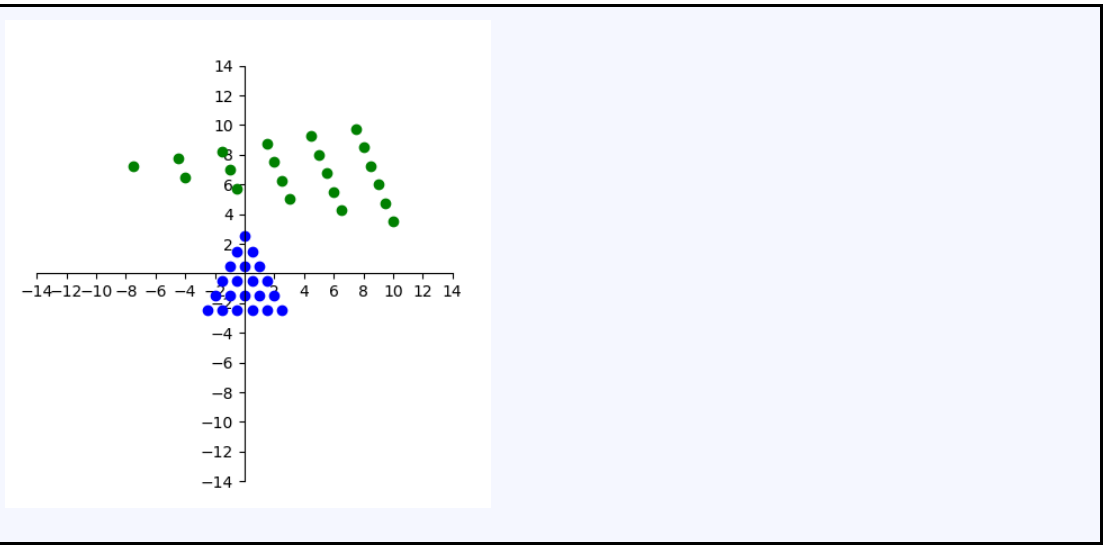
- *Проверьте важность порядка перемножения матриц при комбинировании преобразований.*
- *Проверьте свойство комбинирования преобразований при помощи перемножения матриц этих преобразований для набора из трех или более преобразований.*
- *Проверьте свойство получения обратного преобразования при помощи обращения матрицы этого преобразования.*
- *С помощью матричного умножения можно преобразовывать также и точки, записанные в виде векторов-строк. Для этого нужно транспонировать не только точки, но и матрицы преобразований, а домножение на матрицы преобразований будет производиться не справа налево, а слева направо. Проверьте это свойство.*

Попробуем преобразовать набор точек с помощью произвольного аффинного преобразования. У аффинного преобразования последняя (то есть нижняя) строка состоит из нулевых элементов, кроме последнего (в правом нижнем углу), который должен быть равным единице, а остальные элементы матрицы (из всех строк, кроме последней) могут принимать любые значения, не делающие определитель матрицы равным нулю.

Код:

```
Ma = np.array([[3, 2, 5], [0.5, -1, 6], [0, 0, 1]])
object_points_t6 = Ma @ object_points
plot_2d_point_sets([object_points, object_points_t6])
```

Результат:



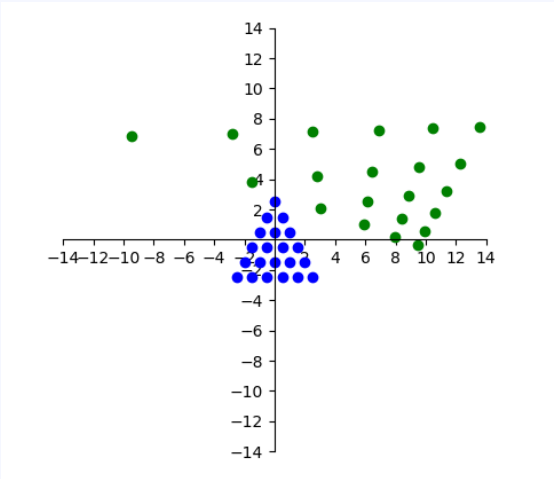
Обратите внимание, что точки, лежавшие на параллельных прямых, после преобразования все еще лежат на параллельных прямых. Свойством сохранения параллельности прямых обладает любое аффинное преобразование.

Теперь попробуем преобразовать набор точек при помощи произвольного проективного преобразования. Элементы нижней строки матрицы проективного преобразования могут принимать произвольные значения, вследствие чего при преобразовании двумерных точек их третья однородная координата может получаться отличной от единицы. Поэтому перед выведением, сравнением и различными другими операциями над точками нужно их нормализовать - поделить на третью однородную координату, то есть выбрать из всех троек однородных координат этой точки такую, которой соответствует значение параметра w, равное единице.

Код:

```
Mp = np.array([[3, 2, 8], [0.5, -1, 2], [0.06, 0.15, 1]])
object_points_t7 = Mp @ object_points
object_points_t7 = object_points_t7 / object_points_t7[2]
plot_2d_point_sets([object_points, object_points_t7])
```

Результат:



Обратите внимание, что точки, лежавшие на общей прямой, после преобразования все еще лежат на общей прямой, однако параллельность различных прямых после преобразования не сохранилась.

Задания и вопросы для самостоятельной работы:

- *При применении к точкам нескольких последовательных проективных преобразований обязательно ли нормализовывать координаты точек после каждого из этих преобразований?*

Применение преобразований координат к изображениям

Определим функцию для визуализации изображений, считаем изображение и выведем его на экран.

Код:

```
def plot_image(img):
    plt.figure()
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    plt.axes([0, 0, 1, 1])
    plt.axis('off')
    plt.imshow(img)
    plt.show()

img = cv2.imread("checkersandbooksmall.jpg")
plot_image(img)
```

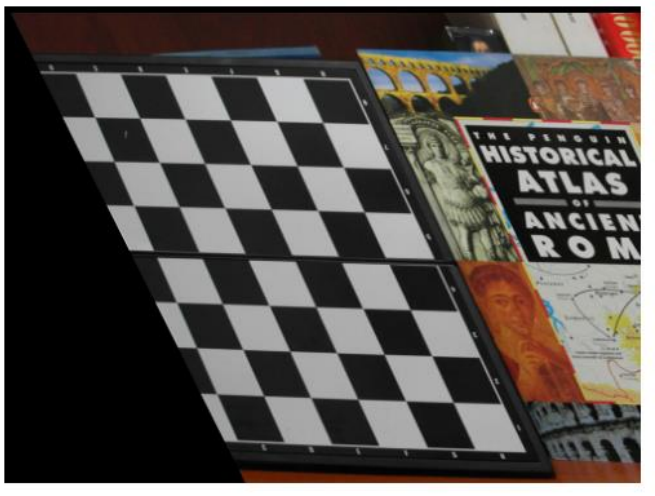
Результат:

Попробуем преобразовать это изображение с помощью функций OpenCV. Для начала воспользуемся функцией cv2.warpAffine. Она применяет к изображению аффинное преобразование, заданное в виде матрицы этого преобразования. Так как третья строка матрицы двумерного аффинного преобразования не несет в себе никакой информации (она одинакова для всех аффинных преобразований), то передавать в функцию cv2.warpAffine нужно матрицу размера 2×3.

Код:

```
Ma2 = np.array([[1, 0.5, 5], [0.0, 1, 6], [0, 0, 1]])
img_a = cv2.warpAffine(img, Ma2[:2], img.shape[1::-1],
flags=cv2.INTER_CUBIC)
plot_image(img_a)
```

Результат:



После применения преобразования к изображению необходимо определить цвета каждого пикселя нового изображения. Для этого потребуется каким-либо образом усреднять значения пикселей, получая промежуточные значения, то есть нужно выполнять интерполяцию. Чтобы указать, какой именно метод интерполяции требуется использовать, можно через параметр flags передать значения константы, соответствующей нужному методу, например, cv2.INTER_CUBIC задает бикубическую интерполяцию.

Важно понимать, что при интерполяции может ухудшаться качество изображения, а значит при применении нескольких последовательных преобразования к одному и тому же изображению нужно, если это возможно, делать это не с помощью многократного вызова функции **cv2.warpAffine**, а с помощью перемножения матриц этих преобразований и однократного применения этой функции с полученной матрицей. Такой подход не только помогает минимизировать потери качества, но и существенно ускоряет процесс обработки изображений.

Также важно помнить, что система координат изображения имеет начало не в центре этого изображения, а в левом верхнем углу, а ось Y направлена вниз, а не вверх. Если требуется выполнить с изображением какие-либо преобразования так, как будто бы система координат находится в центре, а ось Y направлена вверх (например, если требуется повернуть изображение вокруг его центра на заданный угол против часовой стрелки). то можно вначале сместить начало координат в центр изображения с помощью параллельного переноса, затем отразить ось Y, потом выполнить необходимое преобразование в полученной системе координат, и после этого вернуть систему координат обратно, еще раз отразив ось Y и выполнив параллельный перенос на противоположный вектор.

Рассмотрим этот подход на примере. Попробуем повернуть изображение относительно центра на заданный угол против часовой стрелки.

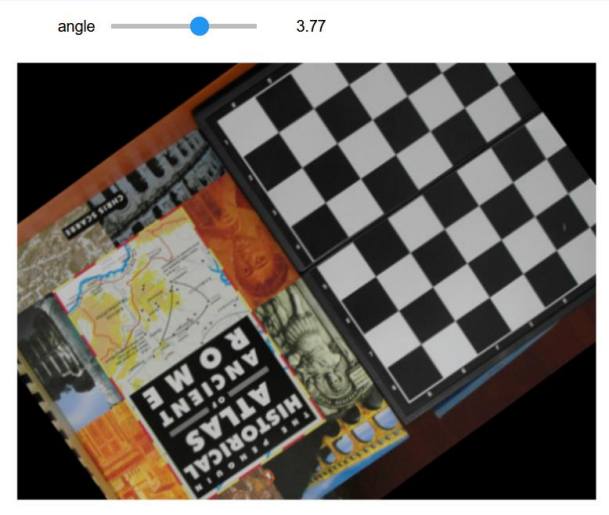
Код:

```
import ipywidgets as widgets

Mt_c = get_translation_matrix(-0.5*np.array(img.shape[1::-1]))
Mref = get_reflection_matrix([False, True])
Mc = Mref @ Mt_c
Mc_inv = np.linalg.inv(Mref @ Mt_c)

@widgets.interact(angle=(0.0, 2*np.pi, 0.2*np.pi))
def rotation_example(angle=np.pi/4):
    Mra = get_rotation_matrix(angle)
    Mrc = Mc_inv @ Mra @ Mc
    img_rc = cv2.warpAffine(img, Mrc[:2], img.shape[1::-1],
flags=cv2.INTER_CUBIC)
    plot_image(img_rc)
```

Результат:



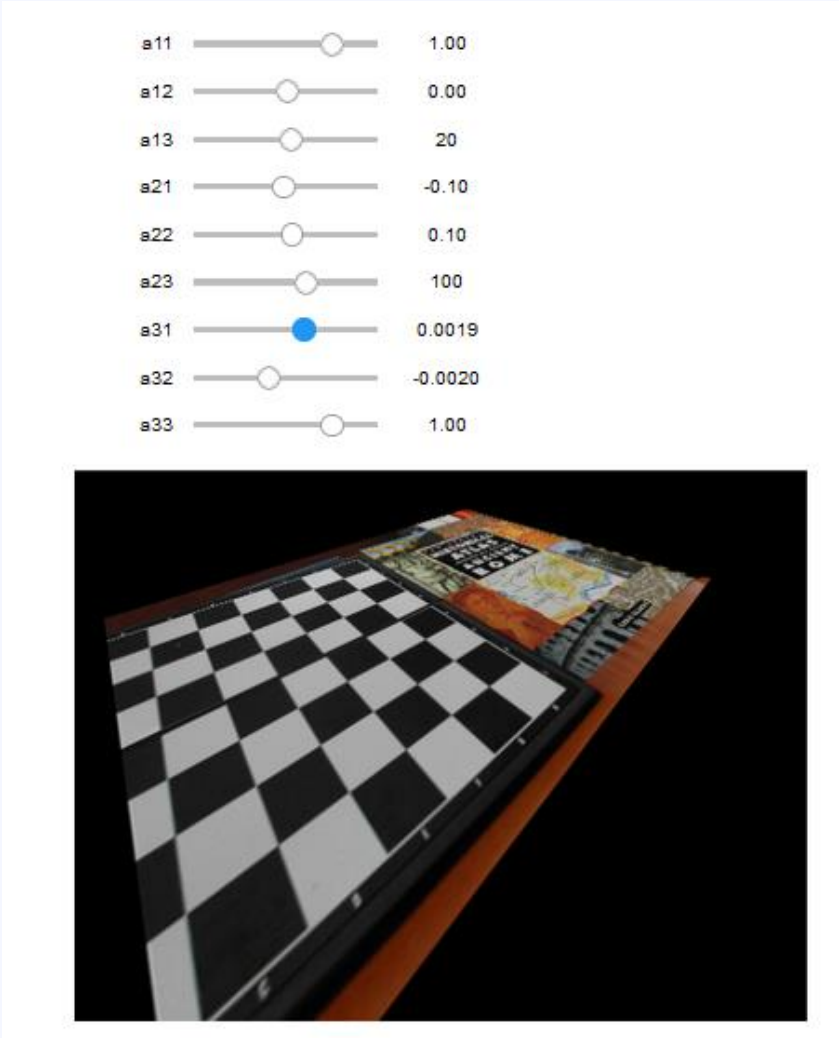
В OpenCV есть также функция **cv2.warpPerspective**, позволяющая применить к изображению проективное преобразование, заданное в виде матрицы. Работает она аналогично функции **cv2.warpAffine**, но принимает на вход матрицу размера 3×3.

Воспользуемся функцией cv2.warpPerspective и модулем **ipywidgets** для создания интерактивной визуализации, позволяющей отображать влияние элементов матрицы проективного преобразования на само это преобразование.

Код:

```
@widgets.interact(
    a11=(-2, 2, 0.1), a12=(-2, 2, 0.1), a13=(-500, 500, 10),
    a21=(-2, 2, 0.1), a22=(-2, 2, 0.1), a23=(-500, 500, 10),
    a31=widgets.FloatSlider(value=0, min=-0.01, max=0.01, step=0.0001,
readout_format='.4f'),
    a32=widgets.FloatSlider(value=0, min=-0.01, max=0.01, step=0.0001,
readout_format='.4f'),
    a33=(-2, 2, 0.1),
)
def perspective_matrix_example(
    a11=1, a12=0, a13=0,
    a21=0, a22=1, a23=0,
    a31=0, a32=0, a33=1,
):
    Mp2 = np.array([[a11, a12, a13], [a21, a22, a23], [a31, a32, a33]])
    img_p2 = cv2.warpPerspective(img, Mp2, img.shape[1::-1],
flags=cv2.INTER_CUBIC)
    plot_image(img_p2)
```

Результат:

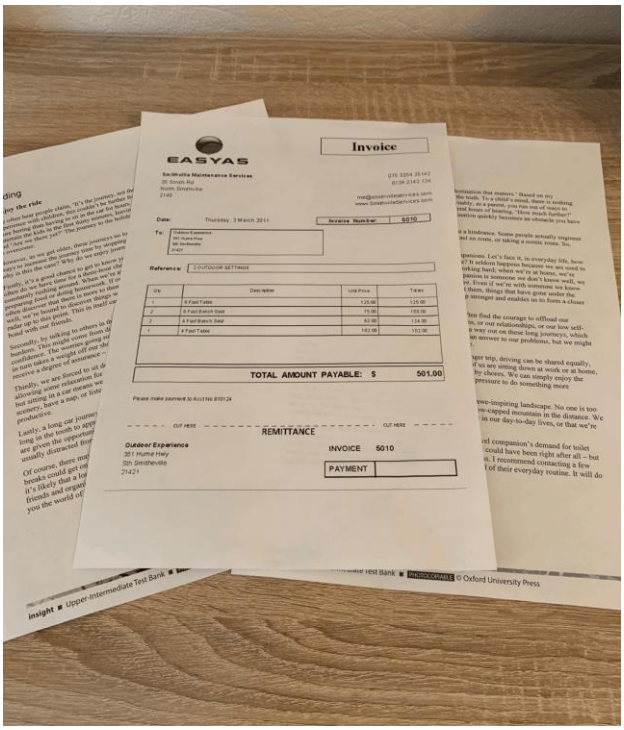


Задания и вопросы для самостоятельной работы:

- *Изучите, каким образом изменение различных элементов матрицы проективного преобразования влияет на это преобразование.*

Функцию **cv2.warpPerspective** можно использовать для исправления искажений перспективы. Для этого удобно использовать функцию **cv2.getPerspectiveTransform**, которая строит матрицу проективного преобразования по четырем точкам (никакие три из которых не лежат на одной прямой), для которых известны их координаты до и после преобразования. Также существует аналогичная функция **cv2.getAffineTransform**, которая строит матрицу аффинного преобразования по трем точкам (не лежащим на одной прямой), для которых известны координаты до и после преобразования.

Рассмотрим пример исправления искажений перспективы. Возьмем следующее изображение:



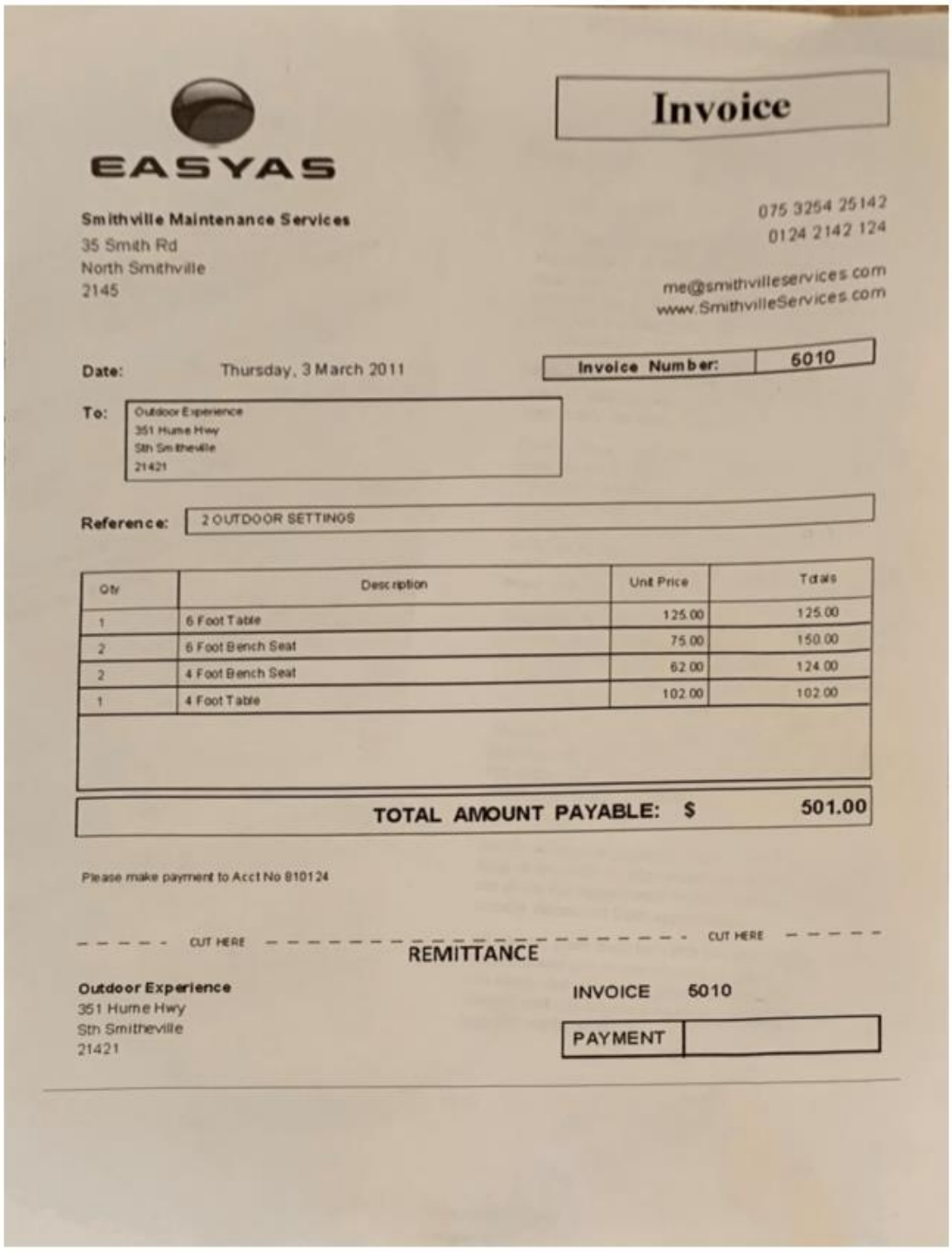
Попытаемся получить выровненное изображение листа бумаги. Для этого найдем (вручную) координаты пикселей, соответствующих углам листа бумаги, и поставим им в соответствие координаты угловых пикселей изображения. Затем вычислим матрицу проективного преобразования, переводящего углы листа бумаги в углы изображения с помощью функции **cv2.getPerspectiveTransform**, и, наконец, применим полученное преобразование к исходному изображению.

Код:

```
def perspective_correction_example():
    img2 = cv2.imread("2.png")
    plot_image(img2)
    src_points = np.float32([[303, 235], [940, 246], [1050, 1200],
[155, 1195]])
    dst_points = np.float32([[0, 0], [img2.shape[1]-1, 0],
[img2.shape[1]-1, img2.shape[0]-1], [0, img2.shape[0]-1]])
    Mp = cv2.getPerspectiveTransform(src_points, dst_points)
    img2_p = cv2.warpPerspective(img2, Mp, img2.shape[1::-1],
flags=cv2.INTER_CUBIC)
    plot_image(img2_p)

perspective_correction_example()
```

Результат:



EASYAS

Invoice

Smithville Maintenance Services
35 Smith Rd
North Smithville
2145

075 3254 25142
0124 2142 124

me@smithvilleservices.com
www.SmithvilleServices.com

Date: Thursday, 3 March 2011

Invoice Number: 5010

To:
Outdoor Experience
351 Hume Hwy
Sth Smitheville
21421

Reference: 2 OUTDOOR SETTINGS

Qty	Description	Unit Price	Totals
1	6 Foot Table	125.00	125.00
2	6 Foot Bench Seat	75.00	150.00
2	4 Foot Bench Seat	62.00	124.00
1	4 Foot Table	102.00	102.00

TOTAL AMOUNT PAYABLE: $ 501.00

Please make payment to Acct No 810124

CUT HERE — CUT HERE

REMITTANCE

Outdoor Experience
351 Hume Hwy
Sth Smitheville
21421

INVOICE 5010

PAYMENT

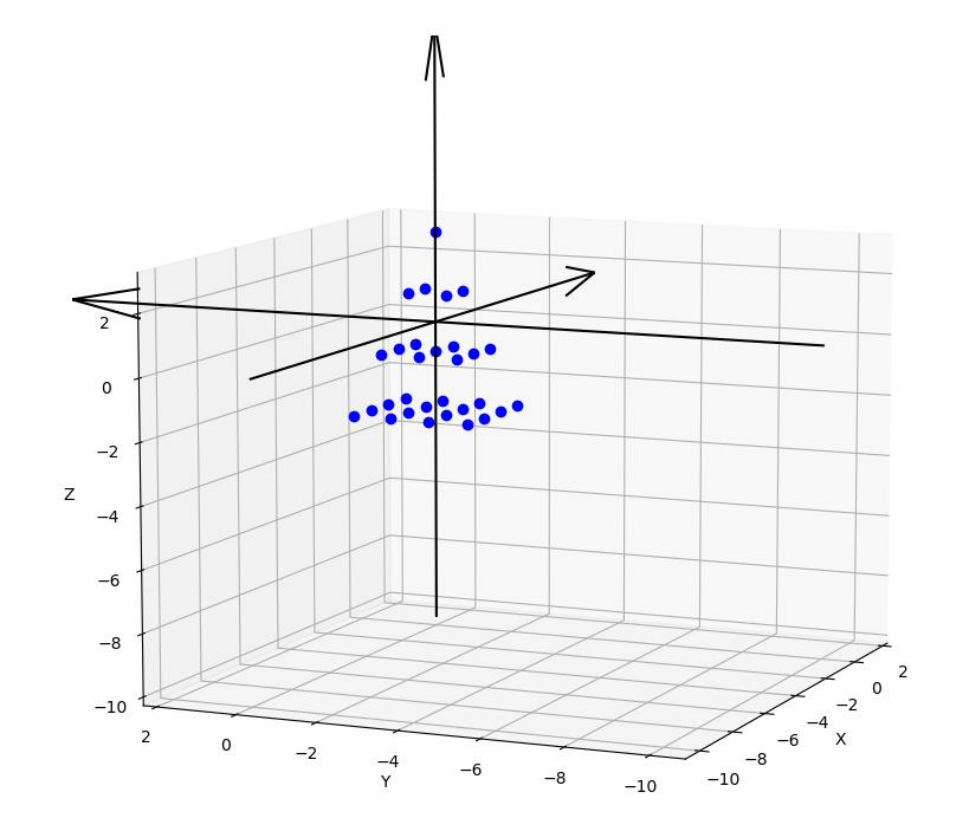
Представление вращений трехмерного пространства

Зададим какой-либо набор точек трехмерного пространства для экспериментов с ним.

Код:

```
object3d_points = np.array([[j - 0.5*(i-1), k - 0.5*(i-1), 5-2*i] for i
in range(1, 5) for j in range(i) for k in range(i)]).T
object3d_points = to_homogeneous(object3d_points)
plot_3d_point_sets([object3d_points])
```

Результат:

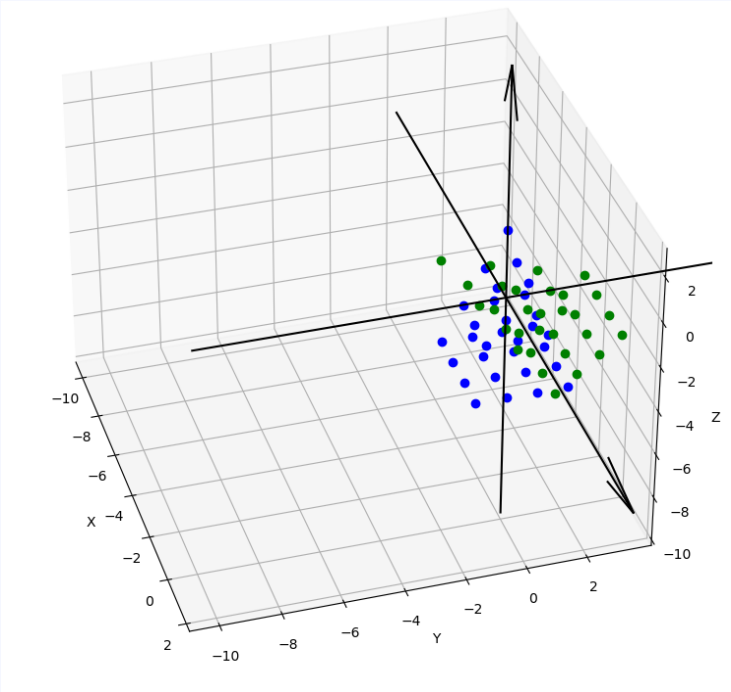


Как уже обсуждалось на лекции, любой поворот пространства можно задать с помощью матрицы поворота. Построим матрицу поворота вокруг оси $0x$ на $45°$ против часовой стрелки и применим этот поворот к набору точек.

Код:

```
Mr3dx = get_rotation_matrix(np.pi/4, axis="x")
object3d_points_r1 = Mr3dx @ object3d_points
plot_3d_point_sets([object3d_points, object3d_points_r1])
```

Результат:



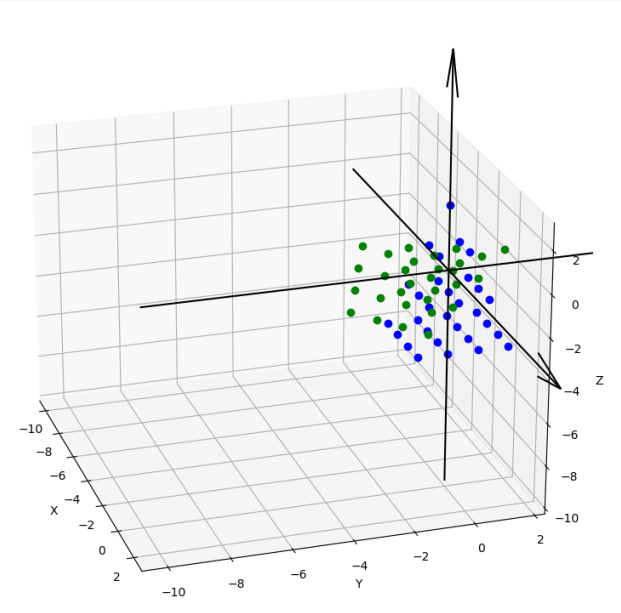
Любой поворот пространства можно также задать в виде углов Эйлера - набора из трех чисел, задающих последовательные повороты на заданные углы вокруг координатных осей, причем то, вокруг каких именно осей и в какой последовательности будут осуществляться повороты, должно фиксироваться заранее.

Будем рассматривать последовательность поворотов XYZ, то есть первый угол задает вращение вокруг оси Ox, второй - вокруг оси Oy и третий - вокруг оси Oz. Сами повороты будем осуществлять при помощи матриц поворота. Повернем набор точек на углы $45°$, $30°$ и $90°$ против часовой стрелки вокруг осей Ox, Oy и Oz соответственно.

Код:

```
Mr3dx = get_rotation_matrix(np.pi/4, axis="x")
Mr3dy = get_rotation_matrix(np.pi/6, axis="y")
Mr3dz = get_rotation_matrix(np.pi/2, axis="z")
Mr_e = Mr3dz @ Mr3dy @ Mr3dx
object3d_points_r2 = Mr_e @ object3d_points
plot_3d_point_sets([object3d_points, object3d_points_r2])
```

Результат:



Задания и вопросы для самостоятельной работы:

- *Проверьте, имеет ли значение порядок углов Эйлера.*
- *Попробуйте использовать другие последовательности координатных осей. Можно ли в последовательности осей использовать одну и ту же координатную ось дважды (например, XYX или XYY) для задания любого поворота трехмерного пространства?*

Повороты пространства можно представлять в виде оси поворота, задаваемой ее направляющим вектором, и величиной угла поворота вокруг этой оси. Поскольку длина направляющего вектора не имеет значения, ее можно использовать для хранения величины угла поворота - такое представление называется вектором поворота.

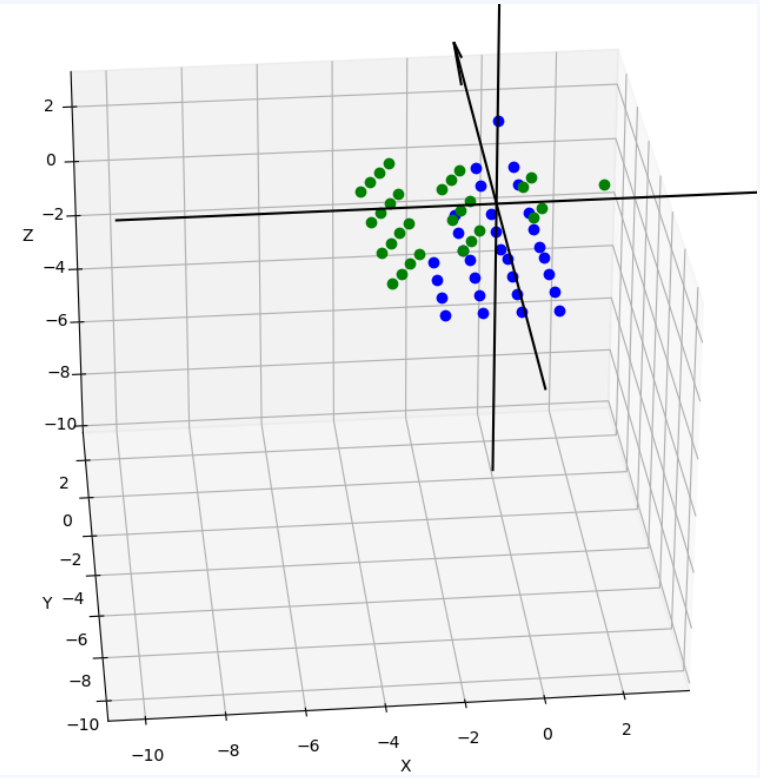
Преобразование Родрига позволяет по вектору поворота получить соответствующую ему матрицу поворота. Воспользуемся этим преобразованием, чтобы повернуть набор точек вокруг оси, проходящей через начало координат и точку с координатами (1; 1; 1), на угол 45° против часовой стрелки.

Код:

```
def rotation_vector_to_matrix(vr, angle=None):
    vr = np.array(vr)
    if angle is None:
        angle = np.linalg.norm(vr)
    va = vr / np.linalg.norm(vr)
    A = np.array([[0, -va[2], va[1]], [va[2], 0, -va[0]], [-va[1],
va[0], 0]])
    Mr = np.eye(4)
    Mr[:3, :3] = np.eye(3) + A * np.sin(angle) + (A @ A) * (1 -
np.cos(angle))
    return Mr

rot_vec = np.array([1, 1, 1])/np.sqrt(3)*np.pi/4
Mr_aa = rotation_vector_to_matrix(rot_vec)
object3d_points_r3 = Mr_aa @ object3d_points
plot_3d_point_sets([object3d_points, object3d_points_r3])
```

Результат:



Задания и вопросы для самостоятельной работы:

- *Дополните функцию get_rotation_matrix так, чтобы она могла выдавать матрицы поворота не только вокруг координатных осей. но и вокруг произвольной оси, то есть чтобы в качестве аргумента axis она могла принимать вектор поворота, выдавая при этом соответствующую ему матрицу поворота, получаемую из функции rotation_vector_to_matrix.*
- *В OpenCV есть функция cv2.Rodrigues, выполняющая преобразование Родрига. Эта функция работает в обе стороны, то есть может выдавать как матрицу поворота по соответствующему ей вектору поворота, так и вектор поворота по соотвтетствующей ему матрице поворота (вектор поворота задается не однозначно). Попробуйте воспользоваться этой функцией и сравнить полученные результаты с результатами, получаемыми с помощью приведенной выше функции rotation_vector_to_matrix.*
- *Попробуйте повернуть набор точек вокруг оси, не проходящей через начало координат.*

Вращения трехмерного пространства можно также производить с помощью кватернионов. Попробуем повернуть набор точек вокруг оси, проходящей через начало координат и точку с координатами (1; 1; 1), на угол 45° против часовой стрелки и сравнить полученный результат с результатами, полученными с помощью матриц поворота.

Код:

```
#all quaternion operations here are in (i, j, k, r) form and not
in (r, i, j, k) form
def quaternion_multiplication(q1, q2):
    b, c, d, a = q1
    f, g, h, e = q2
    qres = np.array([a*f+b*e+c*h-d*g, a*g-b*h+c*e+d*f, a*h+b*g-
c*f+d*e, a*e-b*f-c*g-d*h, ])
    return qres

def quaternion_inversion(q):
    q_inv = np.array([-q[0], -q[1], -q[2],
```

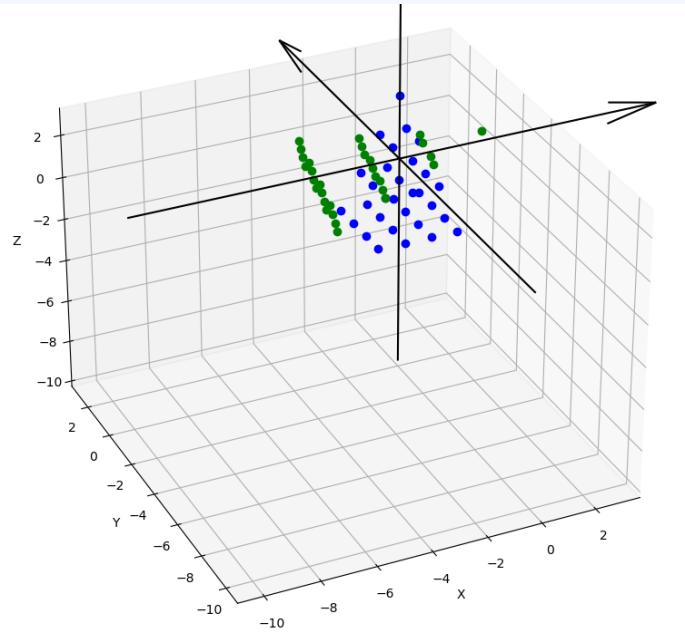
```
q[3]])/(q[0]**2+q[1]**2+q[2]**2+q[3]**2)
    return q_inv

def rotation_vector_to_quaternion(vr, angle=None):
    vr = np.array(vr)
    if angle is None:
        angle = np.linalg.norm(vr)
    va = vr / np.linalg.norm(vr)
    q = np.array([va[0]*np.sin(angle/2), va[1]*np.sin(angle/2),
va[2]*np.sin(angle/2), np.cos(angle/2)])
    return q

def rotate_points_with_quaternions(points, vr, angle=None):
    points_res = np.copy(points)
    q = rotation_vector_to_quaternion(vr, angle)
    for i in range(points.shape[1]):
        points_res[:3, i] =
quaternion_multiplication(quaternion_multiplication(q, [points[0,
i], points[1, i], points[2, i], 0.]), quaternion_inversion(q))[:3]
    return points_res

object3d_points_rq =
rotate_points_with_quaternions(object3d_points, rot_vec)
plot_3d_point_sets([object3d_points, object3d_points_rq])
print(get_max_abs_difference(object3d_points_rq,
object3d_points_r3))
```

Результат:




```
8.881784197001252e-16
```

Как видно, результаты, полученные при помощи кватернионов и при помощи матриц поворота, не отличаются (за исключением незначительных погрешностей).

Оценка положения объекта в двумерном пространстве

Рассмотрим следующую задачу.

Пусть имеется двумерный объект, задающийся набором из n точек. К объекту было применено какое-то преобразование движения (поворот и параллельный перенос) с неизвестными параметрами. Для каждой точки объекта известны ее координаты как в исходном, так и в преобразованном состоянии. Требуется определить положение объекта, задающееся углом поворота вокруг центра этого объекта и вектором сдвига центра. Начальное положение соответствует нулевым параметрам, при этом центр объекта находится в начале координат.

Для начала возьмем какой-либо конкретный пример этой задачи. Зададим каким-либо образом координаты точек объекта в начальном и преобразованном состоянии и выведем эти наборы точек на экран.

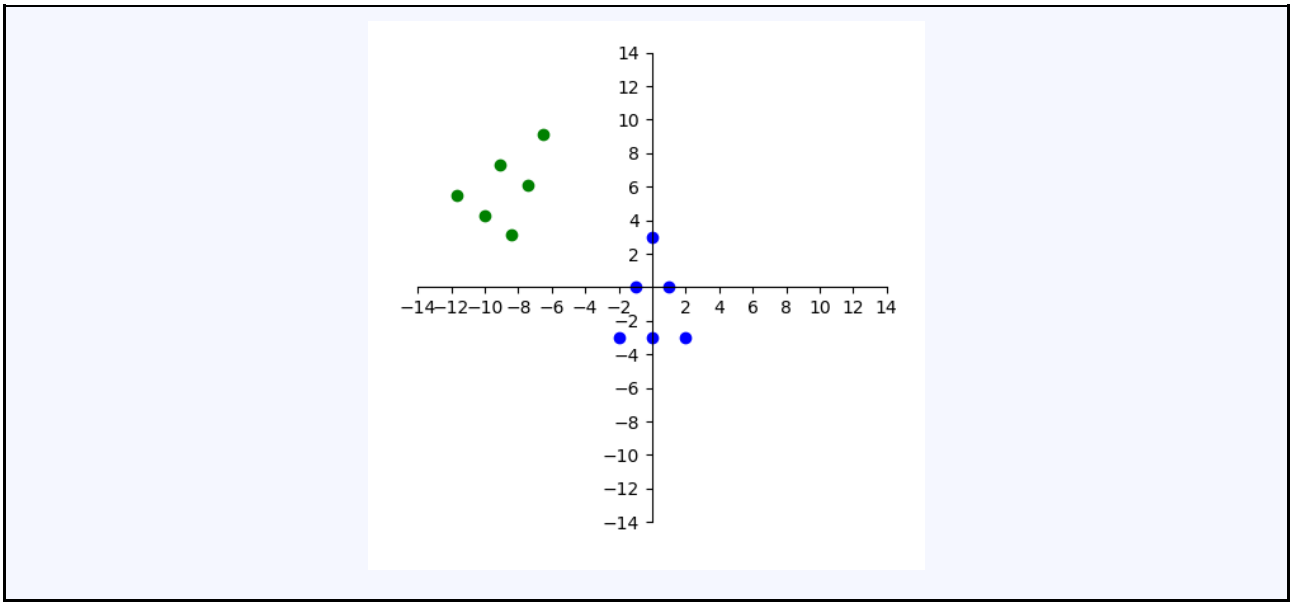
Код:

```
object_points = np.array([[2*(j - 0.5*(i-1)), 6-3*i] for i in range(1,
4) for j in range(i)]).T
object_points = to_homogeneous(object_points)

object_points_transformed = np.array([
    [-6.50664424, -9.07901699, -7.46098301, -11.65138975, -10.03335576,
-8.41532177],
    [9.13705098, 7.29778525, 6.12221475, 5.45851952, 4.28294902,
3.10737851]
])
object_points_transformed = to_homogeneous(object_points_transformed)

plot_2d_point_sets([object_points, object_points_transformed])
```

Результат:



Для того чтобы решить эту задачу, составим систему уравнений, из которой можно будет найти неизвестные параметры. Искомыми параметрами по условию являются угол поворота φ и две координаты вектора переноса tx и ty.

Пусть i-я точка объекта имеет координаты (xi, yi) в начальном состоянии и координаты (ui, vi) в преобразованном состоянии для всех i от 1 до n. Тогда начальное и преобразованные координаты каждой точки связаны следующей формулой:

$$\begin{bmatrix} u_i \\ v_i \end{bmatrix} = \begin{bmatrix} cos\,\varphi & -sin\,\varphi \\ sin\,\varphi & cos\,\varphi \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} = \begin{bmatrix} x_i \cdot cos\,\varphi - y_i \cdot sin\,\varphi + t_x \\ x_i \cdot sin\,\varphi + y_i \cdot cos\,\varphi + t_y \end{bmatrix} \quad (1)$$

Отсюда получаем систему из 2n уравнений следующего вида (i принимает значения от 1 до n включительно):

$$\begin{aligned} x_i \cdot cos\,\varphi - y_i \cdot sin\,\varphi + t_x &= u_i \\ x_i \cdot sin\,\varphi + y_i \cdot cos\,\varphi + t_y &= v_i \end{aligned} \quad (2)$$

Из полученной системы уравнений можно найти неизвестные величины tx, ty и φ. Будем решать эту систему методом Левенберга-Марквардта.

Для каждого уравнения системы зададим функции ошибки следующего вида:

$$f_i(t_x,\ t_y,\ \varphi) = (x_i \cdot cos\ \varphi - y_i \cdot sin\ \varphi + t_x - u_i)^2 \to 0$$
$$g_i(t_x,\ t_y,\ \varphi) = (x_i \cdot sin\ \varphi + y_i \cdot cos\ \varphi + t_y - v_i)^2 \to 0 \quad (3)$$

Эти $2n$ функций ошибки необходимо минимизировать для решения системы.

Для решения этой задачи методом Левенберга-Марквардта требуется найти частные производные каждой из этих функций по каждой из их переменным: tx, ty и φ. Посчитаем эти частные производные:

$$\frac{\partial f_i}{\partial t_x} = 2(x_i \cdot cos\ \varphi - y_i \cdot sin\ \varphi + t_x - u_i)$$
$$\frac{\partial g_i}{\partial t_x} = 0$$
$$\frac{\partial f_i}{\partial t_y} = 0$$
$$\frac{\partial g_i}{\partial t_y} = 2(x_i \cdot sin\ \varphi + y_i \cdot cos\ \varphi + t_y - v_i) \quad (4)$$
$$\frac{\partial f_i}{\partial \varphi} = 2(-x_i \cdot sin\ \varphi - y_i \cdot cos\ \varphi)(x_i \cdot cos\ \varphi - y_i \cdot sin\ \varphi + t_x - u_i)$$
$$\frac{\partial g_i}{\partial \varphi} = 2(x_i \cdot cos\ \varphi - y_i \cdot sin\ \varphi)(x_i \cdot sin\ \varphi + y_i \cdot cos\ \varphi + t_y - v_i)$$

Наконец, можем составить код программы, решающей данную задачу. Для решения системы методом Левенберга-Марквардта будем использовать функцию **least_squares** из модуля **scipy**. Для того, чтобы эта функция использовала метод Левенберга-Марквардта, в нее нужно передать параметр method="lm". Также в нее нужно передать начальное значение переменных, функцию, вычисляющую значение функции ошибки для каждого уравнения системы (можно не использовать возведения в квадрат, и тогда частные производные должны быть вычислены без учета возведения в квадрат), а также функцию, вычисляющую матрицу Якоби минимизируемой функции ошибки (в данном случае она будет иметь размер $2n \times 3$).

Код:

```
from scipy.optimize import least_squares

def pose_estimation_error(transform, src, dst):
    tx, ty, phi = transform
    Mt = get_translation_matrix([tx, ty])
    Mr = get_rotation_matrix(phi)
    src_transformed = Mt @ Mr @ src
    error_value = ((src_transformed[:2] - dst[:2])**2).flatten()
    return error_value

def pose_estimation_error_jac(transform, src, dst):
    num_points = src.shape[1]
    tx, ty, phi = transform

    df_dtx = 2*(src[0]*np.cos(phi)-src[1]*np.sin(phi)+tx-dst[0])
    dg_dtx = np.zeros([num_points])
    df_dty = np.zeros([num_points])
    dg_dty = 2*(src[0]*np.sin(phi)+src[1]*np.cos(phi)+ty-dst[1])
    df_dphi = 2*(-src[0]*np.sin(phi)-
src[1]*np.cos(phi))*(src[0]*np.cos(phi)-src[1]*np.sin(phi)+tx-
dst[0])
    dg_dphi = 2*(src[0]*np.cos(phi)-
src[1]*np.sin(phi))*(src[0]*np.sin(phi)+src[1]*np.cos(phi)+ty-
dst[1])

    jac = np.zeros([2*num_points, len(transform)])
    jac[:num_points, 0] = df_dtx
    jac[num_points:, 0] = dg_dtx
    jac[:num_points, 1] = df_dty
    jac[num_points:, 1] = dg_dty
    jac[:num_points, 2] = df_dphi
    jac[num_points:, 2] = dg_dphi
    return jac

optimization_result = least_squares(
    lambda x : pose_estimation_error(x, object_points,
object_points_transformed),
    x0=np.zeros(3),
    method="lm",
    jac = lambda x : pose_estimation_error_jac(x, object_points,
object_points_transformed),
    verbose=1,
)
tx, ty, phi = optimization_result.x
print(f"tx={tx}, ty={ty}, phi={phi}")
```

Задания и вопросы для самостоятельной работы:

- *Какие значения искомых параметров получились? Попробуйте проверить найденный результат. Сравните значения координат точек объекта в исходном состоянии, преобразованных с помощью найденного преобразования, со значениями, записанными в массиве* ***object_points_transformed****.*