

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ
Кафедра дифференциальных уравнений и системного анализа

**РАЗРАБОТКА И РЕАЛИЗАЦИЯ TELEGRAM-БОТА ДЛЯ ВИЗУАЛИЗАЦИИ
ПРЕОБРАЗОВАНИЯ ФУРЬЕ НА ЯЗЫКЕ PYTHON**

Курсовая работа

Клименко Кирилла
Владимировича

студента 2 курса,
специальность 1-31 03 09
Компьютерная математика
и системный анализ

Научный руководитель:
кандидат физ.-мат. наук,
доцент А. Э. Малевич

Минск, 2020

ОГЛАВЛЕНИЕ

Введение	3
Глава 1. Обработка входного изображения	6
1.1 Постановка задачи	6
1.2 Получение границы и контура изображения	6
1.3 Цветовая сегментация изображения	8
1.4 Обработка изображения нейронной сетью	11
1.5 Визуализация контуров	14
Глава 2. Дискретное преобразование Фурье	15
2.1 О преобразовании и эпициклах	15
2.2 Задание функции контура изображения	20
2.3 Вычисление коэффициентов преобразования Фурье	20
2.4 Вычисление бесконечной суммы преобразования Фурье	21
2.5 Проверка, итог и визуализация результата	21
Глава 3. Создание видео-анимации	23
3.1 Подготовка области анимации	23
3.2 Инициализация объектов анимации	23
3.3 Преобразование координат окружностей	24
3.4 Функция сортировки окружностей	24
3.5 Функция анимации эпициклов	25
3.6 Визуализация преобразования Фурье в виде анимации	26
Глава 4. Конвертация анимации в видео	27
4.1 Формулировка задачи	27
4.2 Многопользовательский режим	27
4.3 Конвертация анимации в видеофайл	28
Глава 5. Создание и описание интерфейса Telegram-бота	30
5.1 Кратко о ботах платформы Telegram	30
5.2 Создание интерфейса Telegram-бота	31

Глава 6. Реализация Telegram-бота	33
6.1 Реализация мультязычности и хранения данных	33
6.2 Взаимодействие с пользователем и описание команд	34
6.3 Обработчики событий	36
6.4 Использование бота	39
Глава 7. Развёртывание Telegram-бота на сервере	40
7.1 Подготовка данных	40
7.2 Файл зависимостей бота	41
7.3 Файл запуска бота	42
7.4 GitHub как хранилище данных	43
7.5 Развёртывание бота на Heroku	43
Заключение	46
Список использованной литературы	48
Приложения	50

ВВЕДЕНИЕ

Огромное значение в жизни человека имеет восприятие информации из окружающего мира. Одним из основных инструментов восприятия является слух. Когда мы слышим звук, мы не слышим саму звуковую волну, но мы слышим различные частоты синусоидальных волн, из которых она состоит. Человеческий мозг преобразует поступающую в ухо звуковую волну в воспринимаемую информацию с помощью некоторого преобразования, представляющего звуковые волны в виде колебательного движения частиц в упругой среде со своими значениями громкости и интенсивности для тонов различных высот.

С помощью математических методов можно проводить аналогичные операции над любыми колебательными процессами: от человеческого голоса и электрических сигналов до световых волн и циклов солнечной активности. Пользуясь преобразованием Фурье, можно раскладывать волны, функции или сигналы, на альтернативные представления в виде набора синусоидальных составляющих.

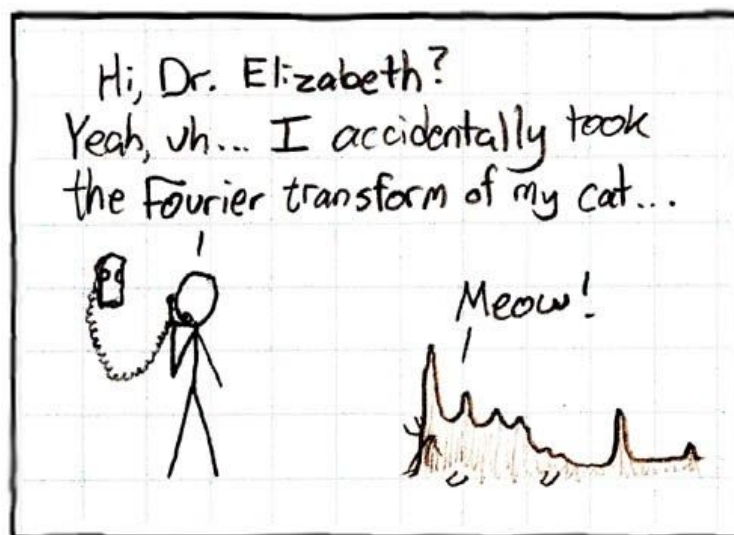
В настоящее время преобразование Фурье является мощным инструментом, применяемым в различных научных областях: в физике, математике, акустике, океанологии, археологии, оптике, геометрии, и многих других. С его помощью можно решать сложные уравнения, описывающие динамические процессы электрической, тепловой или световой энергии. Также оно позволяет выделять регулярные составляющие в сложном колебательном сигнале, что помогает правильно интерпретировать экспериментальные наблюдения в астрономии, медицине и химии.

В рамках данной работы будет практически рассмотрены:

- **дискретное преобразование Фурье** – один из самых мощных инструментов в цифровой обработке сигналов;
- **обработка и сегментация изображений** – неотъемлемая часть визуализации и распознавания графических объектов;
- **разработка и реализация Telegram-бота** – мобильный, быстрый и актуальный пользовательский интерфейс.

Цель данной работы будет заключаться в разработке мультязычного бота на основе платформы-мессенджера Telegram с возможностью обработки и конвертирования изображений в видео-анимацию, на которой с помощью эпциклов и дискретного преобразования Фурье будет отрисовываться контур объекта на заданном изображении.

Вся техническая и практическая часть работы будет реализована посредством языка Python 3.7 и подключаемых к нему модулей. Готовый проект будет развёрнут на сервере и запущен в Telegram. Исходный код можно будет найти в GitHub-репозитории: <https://github.com/Defaultin/FourierTransformBot>.



ГЛАВА 1. ОБРАБОТКА ВХОДНОГО ИЗОБРАЖЕНИЯ

1.1 Постановка задачи

Наша задача – представить объект, изображённый на входном изображении, на координатной плоскости и получить координаты его границы. Чтобы получить чёткую и понятную границу, будем пользоваться алгоритмами цветовой сегментации, которые используются для обнаружения злокачественных опухолей по МРТ-снимку, а также будем получать список всех контуров, распознанных на изображении посредством функционала *Python Imaging Library*. Чтобы расширить спектр качественно обрабатываемых изображений и получить возможность обрабатывать более сложные картинки, воспользуемся нейросетевой моделью для пиксельной сегментации изображений *PSPNet* [7] натренированной на *Pascal VOC 2012 dataset* и библиотекой *Keras* [6] для работы с нейронными сетями. Для построения графиков воспользуемся библиотекой *matplotlib* [1], для работы с контурами изображения – библиотекой *PIL* [4], для цветовой сегментации изображений – библиотекой *OpenCV* [5], для математических выражений и констант воспользуемся библиотеками *numpy* [2] и *math* [3].

1.2 Получение границы и контура изображения

Создадим функцию, которая будет принимать картинку и возвращать координаты на плоскости контура объекта на входном изображении. При

получении картинки на вход запишем её в массив и сразу закроем во избежание сбоев в файловой системе и искажения самого изображения.

```
image = Image.open(image_name).convert('L')
im = array(image)
image.close()
```

Преобразуем полученный массив в массив контуров и выберем самый первый контур с наибольшим периметром, это позволит нам в случае сложных изображений получить осмысленную границу одного из объектов (далее с помощью цветокоррекции и нейросети мы улучшим распознавание сложных объектов на изображении).

```
contour_plot = contour(im, levels=level, colors='black', origin='image')
contour_path = contour_plot.collections[0].get_paths()[0]
```

После получения контура изображения создадим таблицу координат этого контура, а также таблицу значений от 0 до 2π , размерность которой равна количеству точек в контуре.

```
x_table, y_table = contour_path.vertices[:, 0], contour_path.vertices[:, 1]
time_table = np.linspace(0, 2*pi, len(x_table))
```

Выровняем координаты контура по центру (относительно начала координат).

```
x_table = x_table - min(x_table)
y_table = y_table - min(y_table)
x_table = x_table - max(x_table) / 2
y_table = y_table - max(y_table) / 2
```

Объединим всё в итоговую функцию *create_contour* [см. Приложение 1], которая по изображению будет строить его внешний контур.

Стоит заметить, что можно было не брать внешний контур, а замкнуть все детали изображения в одну сплошную линию. То есть найти единственный кратчайший путь через все найденные пиксели с помощью поиска в ширину

(алгоритм Дейкстры или A^*). Но в большинстве случаев, например, в сложных изображениях с множеством объектов, такой подход будет некорректен, так как соединения всех частей изображения захламляют общую картину [см. Рис. 1.1], поэтому для сохранения смысла изображения мы берём только внешнюю границу.

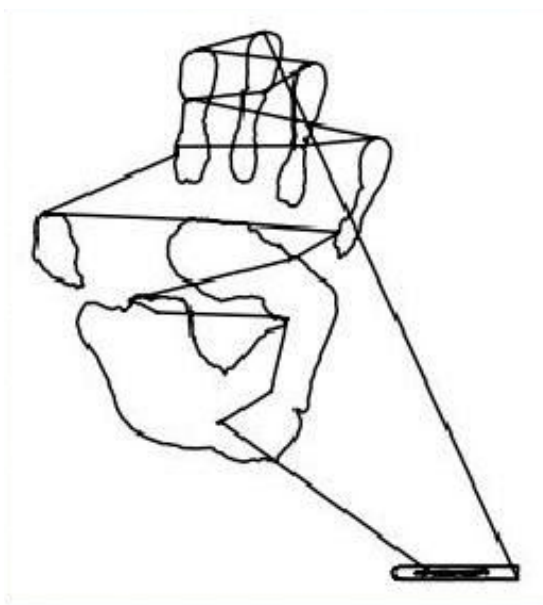


Рисунок 1.1

1.3 Цветовая сегментация изображения

Если текстура изображения неровная и неоднородная, или на картинке много цветов, распознать объект и его контур довольно сложно. Однако, уменьшив интенсивность главных цветов RGB, мы можем контролировать насыщенность изображения и сводить его к одному оттенку. С помощью таких манипуляций при оконтуривании сложные объекты изображения будут рассматриваться едиными объектами. Мы, конечно же, могли бы разработать фильтры для всех трёх цветов – красного, зелёного и синего – но тогда нам придётся позаботиться о контроле доминантных цветов изображения, чтобы точно знать, какие фильтры применять в заданной ситуации. Однако, так как самым удобным цветом для компьютерной обработки является зелёный, так как он наиболее яркий,

выделяющийся среди других и часто встречается в виде фона (поэтому чаще всего фильмы снимают на "зелёнке"), то лучше всего будет унифицировать зеленый цвет и свести его к одному оттенку, перед этим "отнегативив" изображение. Тогда нам не придётся использовать много фильтров, следить за доминантными цветами, и это значительно улучшит распознавание контура, особенно если мы нарисуем что-то на зелёной доске и сфотографируем.

Создадим функцию, которая будет осуществлять цветокоррекцию и сохранять откорректированное изображение. Для начала необходимо определить HSV-представление цвета. Это можно сделать путем преобразования его RGB в HSV.

```
green = np.uint8([[[0, 255, 0]]])
green_hsv = cv2.cvtColor(green, cv2.COLOR_BGR2HSV)
```

Прочитаем и инвертируем входное изображение: *[255-Red, 255-Green, 255-Blue]*.

```
image = cv2.imread(image_name)
image = ~image if reverse else image
```

С помощью HSV проще получить полный диапазон одного цвета. Н здесь означает тон (*hue*), S – насыщенность (*saturation*), а V – значение (*value*). Зеленый цвет (*green_hsv*) в HSV-представлении – это [60, 255, 255]. Весь зеленый цвет в мире находится в диапазоне с [45, 100, 50] по [75, 255, 255], то есть с [60-15, 100, 50] по [60+15, 255, 255], где 15 – это примерное значение изменения оттенка.

Превратим этот диапазон в [75, 255, 200] или любой другой светлый цвет (третье значение должно быть сравнительно большим). Последняя цифра представляет собой яркость цвета – как раз та величина, которая красит область в белый после порогового преобразования изображения.

```
hsv_img = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
green_low = np.array([45, 100, 50])
```

```

green_high = np.array([75, 255, 255])
curr_mask = cv2.inRange(hsv_img, green_low, green_high)
hsv_img[curr_mask > 0] = ([75, 255, 200])

```

Преобразуем HSV-изображения к оттенкам серого для дальнейшего оконтуривания.

```

RGB_again = cv2.cvtColor(hsv_img, cv2.COLOR_HSV2RGB)
gray = cv2.cvtColor(RGB_again, cv2.COLOR_RGB2GRAY)
ret, threshold = cv2.threshold(gray, 90, 255, 0)

```

Сохраним откорректированное изображение под названием "*ID_before2.jpg*", где ID – это *Telegram id* пользователя [\[см. 6.1\]](#).

```

cv2.imwrite(f'{ID}_before2.jpg', threshold)

```

В итоге получим функцию *color_correction* [см. Приложение 2], которая приводит изображение к одному оттенку, чтобы из неё было удобно получить контур. Промежуточные результаты работы функции изображены на Рисунке 1.2.



Рисунок 1.2

1.4 Обработка изображения нейронной сетью

Пиксельная сегментация изображения является хорошо изученной проблемой в компьютерном зрении. Задача сегментации изображения состоит в том, чтобы классифицировать каждый пиксель в изображении из заранее определенного набора классов. Наша цель состоит в том, чтобы взять изображение размером $W \times H \times 3$ и сгенерировать матрицу $W \times H$, содержащую предсказанные идентификаторы класса, соответствующие всем пикселям [см. Рис. 1.3].

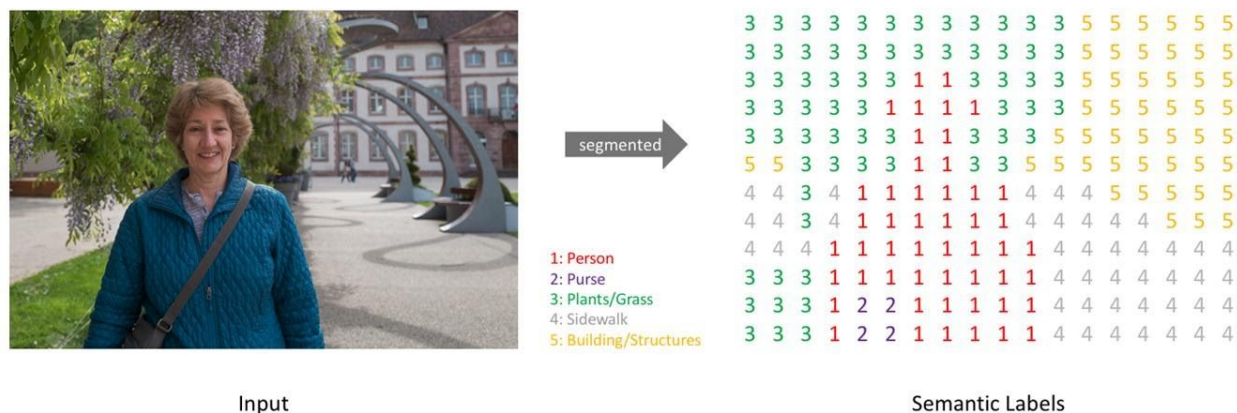


Рисунок 1.3

Обычно на изображении с различными объектами мы хотим знать, какой пиксель принадлежит какому объекту. Например, на классической фотографии мы можем сегментировать небо, землю, деревья, людей и т. д. В нашем случае пиксельная сегментация поможет классифицировать объекты на изображении и выделить их соответствующим цветом. После чего мы сможем применить цветовой фильтр, чтобы впоследствии получить чёткий контур нужных объектов.

Цветовая сегментация также очень полезна в таких сферах, как медицина (модели могут быть обучены сегментации опухолей), разработка автономных транспортных средств (беспилотники могут обнаруживать другие автомобили, пешеходов, дорогу и т.д.), космические и геологические исследования (аэрофотоснимки можно использовать для сегментирования различных типов земель, планет и т.д.).

Для наших нужд достаточно будет уже натренированной модели. Выбирать будем из наиболее подходящих к нашей задаче вариантов, а именно: *ResNet*, *VGG-16*, *MobileNet*, *FCN*, *UNet*, *PSPNet*.

1) *ResNet* – модель, предложенная Microsoft, которая получила 96,4% точности в конкурсе *ImageNet 2016*. Имеет большое количество слоёв, что делает её тренировку и использование довольно продолжительными во времени.

2) *VGG-16* – модель, предложенная Оксфордом, которая получила 92,7% точности в конкурсе *ImageNet 2013*. По сравнению с *ResNet* она имеет меньше слоёв, поэтому тренируется намного быстрее. Для большинства существующих тестов сегментации *VGG* не работает так же хорошо, как *ResNet*, с точки зрения точности.

3) *MobileNet* – модель, предложенная Google, которая оптимизирована для более быстрого получения результата. Это идеально для работы на мобильных телефонах и устройствах с ограниченными ресурсами. Из-за небольшого размера точность модели немного страдает.

4) *FCN* – является одной из первых предложенных моделей для цветовой сегментации, что делает её не очень популярной в наше время.

5) *UNet* – используется для сегментации изображений в медицинской области. Благодаря пропущенным соединениям *UNet* не пропускает мельчайшие детали. Отлично работает с объектами небольшого размера.

6) *PSPNet* – оптимизирована для лучшего представления глобального контекста на изображении. Сначала изображение передается в базовую сеть для получения карты объектов. Карта объектов понижена до разных масштабов. Свертка применяется к объединенным картам объектов. После этого все карты объектов преобразуются в общий масштаб и объединяются вместе. Наконец, другой слой свертки используется для получения окончательных результатов сегментации. Отлично работает с объектами разного размера, не теряя общего смысла изображения.

Тщательно изучив и протестировав предложенные модели, выберем модель *PSPNet (trained on Pascal VOC 2012 dataset)* [7].

Установим модель *PSPNet* и веса к ней, скопировав всё в рабочий каталог с нашим проектом. Для работы с нейросетью будем использовать библиотеку *Keras* [6]. Создадим функции для инициализации модели и скомпилируем модель с её уже настроенными весами [см. Приложение 3], а также напишем функцию для обработки изображений по нужным нам сценариям. Результаты обработки нейронной сетью с цветовой фильтрацией изображены на рисунке 1.5.



Рисунок 1.5

1.5 Визуализация контуров

Создадим конструкторы графиков и отобразим полученные контуры. Сделаем две области: в первой будем отображать получившуюся замкнутую границу с помощью функции *create_contour* и входного изображения, во второй – замкнутую границу с помощью той же функции [\[см. 1.2\]](#) и откорректированного с помощью функции *color_correction* [\[см. 1.3\]](#) входного изображения. Эти графики позже будем отправлять пользователю для того, чтобы он убедился в корректности выделенных границ и выбрал удовлетворяющую его обработку изображения.

```
fig, ax = plt.subplots(1, 2, figsize=(10, 5))
ax[0].set_aspect('equal', 'datalim')
ax[1].set_aspect('equal', 'datalim')
ax[0].set_title('Algorithm 1')
ax[1].set_title('Algorithm 2')
```

Получим два контура изображения: с цветокоррекцией и без. Отобразим получившиеся контуры в соответствующих областях и сохраним в виде изображения с названием *"ID_after.jpg"*, где ID – это *Telegram id* пользователя [\[см. 6.1\]](#).

```
color_correction(image_name, ID=ID, reverse=True)
contour_1 = create_contour(image_name)
contour_2 = create_contour(f'{ID}_before2.jpg')
ax[0].plot(contour_1[1], contour_1[2], 'k-')
ax[1].plot(contour_2[1], contour_2[2], 'k-')
plt.savefig(f'{ID}_after.jpg')
plt.close('all')
```

Соединим это всё в функцию *get_contours* [см. Приложение 4], которая визуализирует полученные контуры. Будем ожидать, что в любом случае хотя бы

один из алгоритмов даст удовлетворяющий нас результат. Иначе – будем пользоваться нейросетевой обработкой изображения.

Результат работы функции проиллюстрирован на Рисунке 1.6.

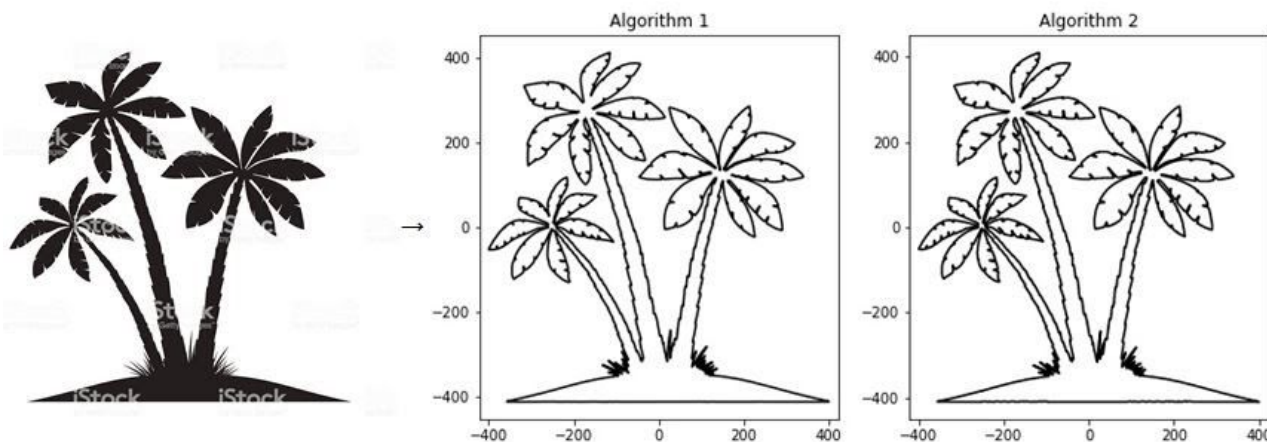


Рисунок 1.6

ГЛАВА 2. ДИСКРЕТНОЕ ПРЕОБРАЗОВАНИЕ ФУРЬЕ

2.1 О преобразовании и эпициклах

Вспомним всем известную формулу Эйлера $e^{it} = \cos(t) + i \sin(t)$, которая даёт нам очень компактный способ параметризации точек, лежащих на единичной окружности, проиллюстрированный на Рисунке 2.1.

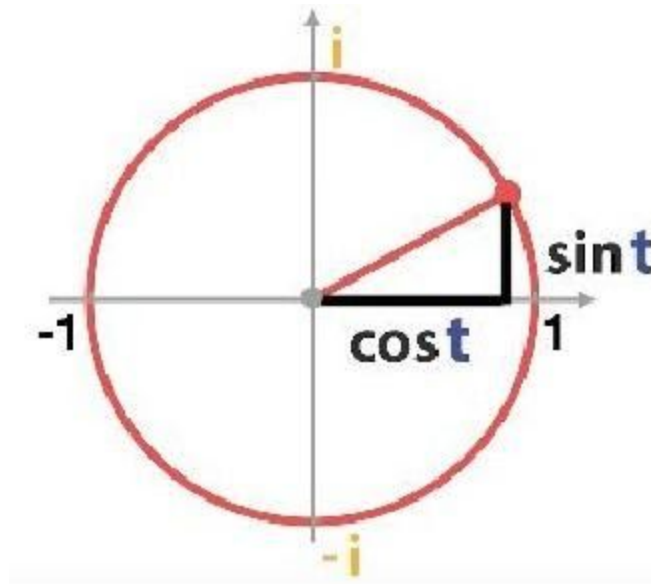


Рисунок 2.1

Если мы положим $t = \pi$, то красная точка сдвинется к -1 , так как $e^{i\pi} = -1$, положим $t = 0$ и получим 1 . Значит, когда мы меняем значение t от 0 до π , красная точка проходит по единичной окружности один раз против часовой стрелки. Если мы рассмотрим случай с отрицательным t , то это будет соответствовать движению точки по той же окружности в противоположном направлении – по часовой стрелке. Если мы рассмотрим $2t$ вместо t , то есть e^{2it} - это будет соответствовать движению нашей точки по окружности дважды против часовой стрелки, и так далее. Рассмотрим еще одно наблюдение: умножим нашу "круговую экспоненту" на 4: $4e^{it}$. Получим, что красная точка движется уже не по единичной окружности, а по окружности с радиусом 4, что показано на Рисунке 2.2.

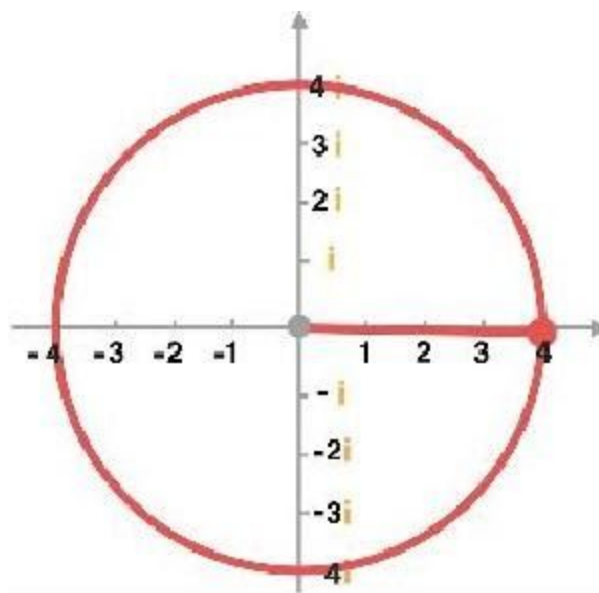


Рисунок 2.2

Иными словами, красная точка движется в комплексной плоскости, начиная с того комплексного числа, на которое мы умножаем. Если же мы домножим экспоненту на комплексное число $1 + i$, то перейдём к движению по кругу, начинающемуся и заканчивающемуся в точке $1 + i$ (см. Рис. 2.3).

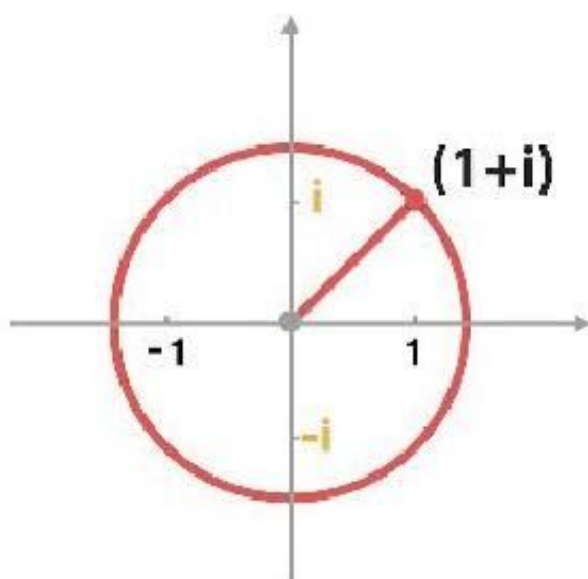


Рисунок 2.3

Именно такие выражения – комплексное число, умноженное на e^{nit} - обозначают разнообразные эпициклы, полученные при помощи механизма Фурье. Интересно то, что интегралы подобных экспонент от 0 до 2π будут всегда равны 0: $\int_0^{2\pi} e^{nit} dt = 0$. Рассмотрим 4 круга с индивидуальным радиусом и начальным положением точек на них (см. Рис. 2.4).

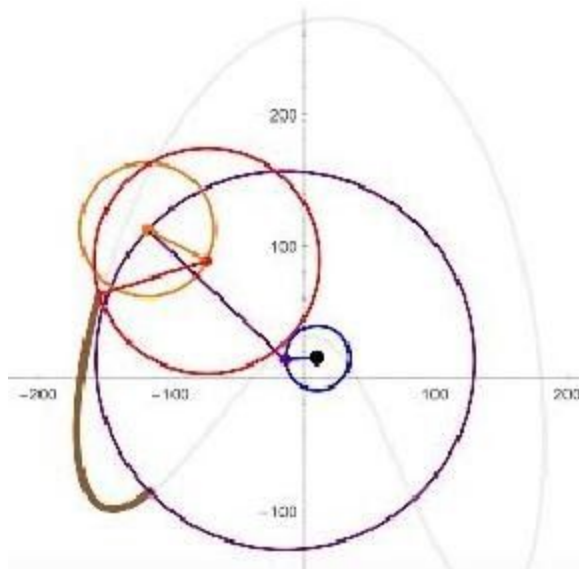


Рисунок 2.4

Эти круги также могут быть представлены в виде суммы:

$$(a_0 + ib_0) e^{0it} + (a_1 + ib_1) e^{1it} + (a_{-1} + ib_{-1}) e^{-1it} + (a_2 + ib_2) e^{2it} + (a_{-2} + ib_{-2}) e^{-2it}$$

Эта сумма в точности описывает интересующее нас движение по контуру. В общем случае она представляется в следующем виде:

$\leftarrow \dots + c_{-2} e^{-2it} + c_{-1} e^{-1it} + c_0 e^{0it} + c_1 e^{1it} + c_2 e^{2it} + \dots \rightarrow$, где $t \in [0; 2\pi]$, то есть в виде двойной бесконечной суммы в обоих направлениях. Все слагаемые являются периодическими функциями с областью определения $[0; 2\pi]$ и областью

значений в комплексной плоскости. Механизм преобразований Фурье перерабатывает любую периодическую функцию из интервала $[0; 2\pi]$ в комплексные числа и представляет её в виде двусторонней бесконечной суммы:

$$f(t) = \dots + c_{-2} e^{-2it} + c_{-1} e^{-1it} + c_0 e^{0it} + c_1 e^{1it} + c_2 e^{2it} + \dots$$

Коэффициент c_0 будет центром первой окружности, вокруг которой, по сути, вращаются все остальные.

Интересно то, что каждый из коэффициентов c_i мы можем легко найти. Для примера найдём коэффициент c_2 . Для этого домножим обе части равенства на e^{-2it} , преобразуем и проинтегрируем их:

$$f(t) e^{-2it} = \dots + c_1 e^{1it} e^{-2it} + c_2 e^{2it} e^{-2it} + c_3 e^{3it} e^{-2it} + \dots \Leftrightarrow$$

$$\Leftrightarrow f(t) e^{-2it} = \dots + c_1 e^{(1-2)it} + c_2 e^{(2-2)it} + c_3 e^{(3-2)it} + \dots \Leftrightarrow$$

$$\Leftrightarrow f(t) e^{-2it} = \dots + c_1 e^{-it} + c_2 + c_3 e^{it} + \dots \Leftrightarrow$$

$$\Leftrightarrow \int_0^{2\pi} f(t) e^{-2it} dt = \dots + \int_0^{2\pi} c_1 e^{-it} dt + \int_0^{2\pi} c_2 dt + \int_0^{2\pi} c_3 e^{it} dt + \dots \Leftrightarrow$$

$$\Leftrightarrow \int_0^{2\pi} f(t) e^{-2it} dt = \int_0^{2\pi} c_2 dt \Leftrightarrow c_2 = \frac{1}{2\pi} \int_0^{2\pi} f(t) e^{-2it} dt$$

Тогда n -ый коэффициент находится следующим образом: $c_n = \frac{1}{2\pi} \int_0^{2\pi} f(t) e^{-nit} dt$

Таким образом, когда нам нужно обрисовать контур, задаваемый функцией $f(t)$, механизм преобразований Фурье вычисляет все коэффициенты c_n , каждый из которых даёт нам ровно один эпицикл.

2.2 Задание функции контура изображения

Представим функцию $f(t)$, описывающую контур изображения в виде кусочно-линейного интерполянта по заданным точкам t , которые мы вычислили ранее. Для интерполяции используем функцию *interp* библиотеки *numpy* [2].

```
def f(t, time_table, x_table, y_table):  
    return np.interp(t, time_table, x_table) + 1j*np.interp(t, time_table,  
y_table)
```

2.3 Вычисление коэффициентов преобразования Фурье

Коэффициенты c_n преобразования Фурье будем вычислять по формуле $c_n = \frac{1}{2\pi} \int_0^{2\pi} f(t) e^{-n i t} dt$, где $f(t)$ – контур изображения, n – номер коэффициента, t – параметр (позже обозначим как количество точек). Так же в функцию вычисления коэффициентов будем передавать размерность двусторонней суммы *order* для контроля количества коэффициентов. Для интегрирования воспользуемся функцией *quad* библиотеки *scipy*.

```
from scipy.integrate import quad  
  
def coef_list(time_table, x_table, y_table, order=10):  
    coef_list = []  
    for n in range(-order, order+1):  
        real_coef = quad (lambda t: np.real(f(t, time_table, x_table,  
y_table) * np.exp(-n*1j*t)), 0, 2*pi, limit=100, full_output=1)[0]/(2*pi)  
        imag_coef = quad (lambda t: np.imag(f(t, time_table, x_table,  
y_table) * np.exp(-n*1j*t)), 0, 2*pi, limit=100, full_output=1)[0]/(2*pi)  
        coef_list.append([real_coef, imag_coef])  
    return np.array(coef_list)
```

2.4 Вычисление бесконечной суммы преобразования Фурье

Представим комплексный ряд Фурье в виде следующей суммы:

$$S(t) = \sum_{-\infty}^{\infty} C_n e^{-n i t} = -A \dots + c_{-2} e^{-2it} + c_{-1} e^{-1it} + c_0 e^{0it} + c_1 e^{1it} + c_2 e^{2it} + \dots + A,$$

в которой количество слагаемых составляет $2A + 1$, оно же – количество окружностей, которые будут обрисовывать наше изображение (c_i – радиусы окружностей). В дальнейшем константу A будем назовём "уровнем аппроксимации", так как от неё будет зависеть то, насколько функция будет похожа на исходную. Действительно, чем больше слагаемых в сумме $S(t)$, тем лучше аппроксимация. Создадим функцию [см. Приложение 5], принимающую параметр t , список коэффициентов Фурье и размерность суммы. Она будет возвращать нам список действительных и мнимых чисел, которые в будущем мы представим на комплексной плоскости как координаты по оси абсцисс и ординат соответственно.

2.5 Проверка, итог и визуализация результата

Проверим работу всех функций. Изобразим оленя (см. Рис. 2.5), состоящего из 300 точек, установив уровень аппроксимации равным 10. Чёрным цветом изображён контур оленя, а красным – его образ Фурье.

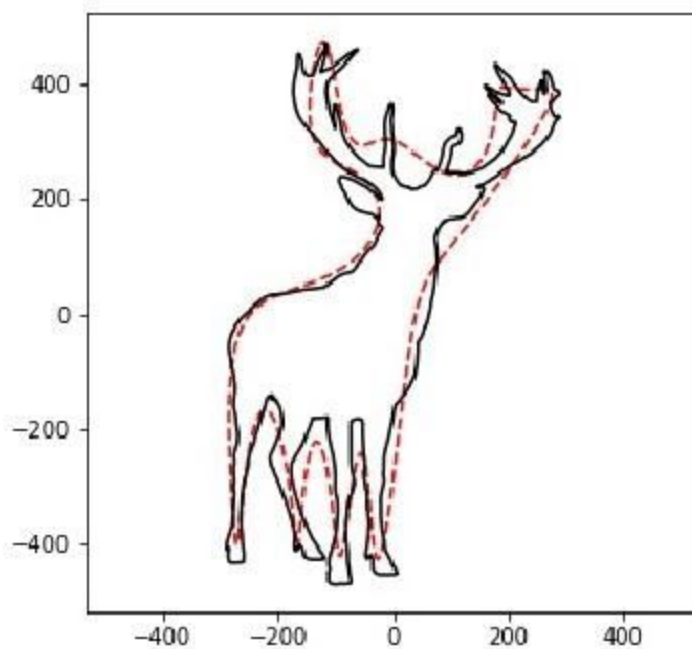


Рисунок 2.5

Образ фурье практически повторяет исходный контур изображения. Увеличим уровень аппроксимации до 25 (см. Рис. 2.6) и до 50 (см. Рис. 2.7).

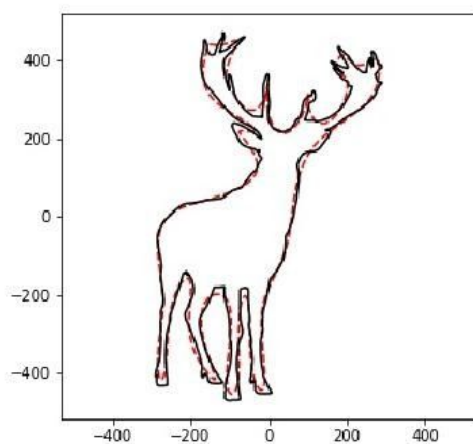


Рисунок 2.6

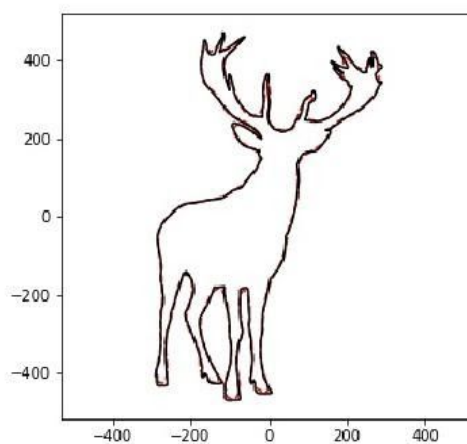


Рисунок 2.7

ГЛАВА 3. СОЗДАНИЕ ВИДЕО-АНИМАЦИИ

3.1 Подготовка области анимации

Создадим функцию визуализации, которая будет принимать на вход список координат точек, список коэффициентов Фурье, уровень аппроксимации, количество точек (также и количество кадров в анимации) и границы области анимации. Для анимации объектов воспользуемся библиотекой *matplotlib animation* [1]. Инициализируем рабочую область и установим её границы, так же сделаем равными масштабы осей во избежание различных искажений и отключим отображение координатных осей.

```
from matplotlib import animation

def visualize(x_DFT, y_DFT, coef, order, space, fig_lim):
    fig, ax = plt.subplots()
    lim = 3*max (fig_lim)/2
    ax.set_xlim([-lim, lim])
    ax.set_ylim([-lim, lim])
    ax.set_aspect('equal')
    ax.axis('off')
```

3.2 Инициализация объектов анимации

В той же функции инициализируем линией чёрного цвета толщины 2 контур изображения, который мы будем обрисовывать с помощью окружностей. Окружности и их радиусы будем рисовать красным цветом с толщиной 0.5.

```
line = plt.plot([], [], 'k-', linewidth=2)[0]
radius = [plt.plot([], [], 'r-', linewidth=0.5)[0] for _ in range(2*order+1)]
circles = [plt.plot([], [], 'r-', linewidth=0.5)[0] for _ in range(2*order+1)]
```

3.3 Преобразование координат окружностей

Создадим функцию, которая будет преобразовывать координаты радиус-векторов окружностей ($Re(c_n)$, $Im(c_n)$) при повороте на каждом кадре анимации. Каждый кадр будем обновлять за время $\frac{2\pi}{space}$, где $space$ – количество кадров в анимации. В качестве выходных значений будем получать новые координаты радиус-векторов окружностей в виде массива. Преобразование координат будем осуществлять с помощью матрицы поворота, где обозначим за (a, b) старые координаты, а за (x, y) – новые координаты:

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} \Leftrightarrow \begin{cases} x = a\cos(\theta) - b\sin(\theta) \\ y = a\sin(\theta) + b\cos(\theta) \end{cases}$$

```
def update_c(c, t):
    new_c = []
    for i, j in enumerate(range(-order, order + 1)):
        theta = -j * t
        cos, sin = np.cos(theta), np.sin(theta)
        v = [cos * c[i][0] - sin * c[i][1], sin * c[i][0] + cos *
c[i][1]]
        new_c.append(v)
    return np.array(new_c)
```

3.4 Функция сортировки окружностей

Из-за того, что в нашем списке радиусов окружностей эти радиусы расположены не по порядку "обращения вокруг друг друга" окружностей, мы должны расположить их в нужном нам порядке: $[c_1, c_{-1}, c_2, c_{-2}, \dots]$, чтобы удобнее было их анимировать. Для удобства индексации в цикле упорядочим не сами коэффициенты, а их индексы в списке. Пример ожидаемого результата последовательности индексов для упорядочивания окружностей по порядку

"обращения вокруг друг друга": [6,4,7,3,8,2,9,1,10,0] (см. Рис. 3.1).

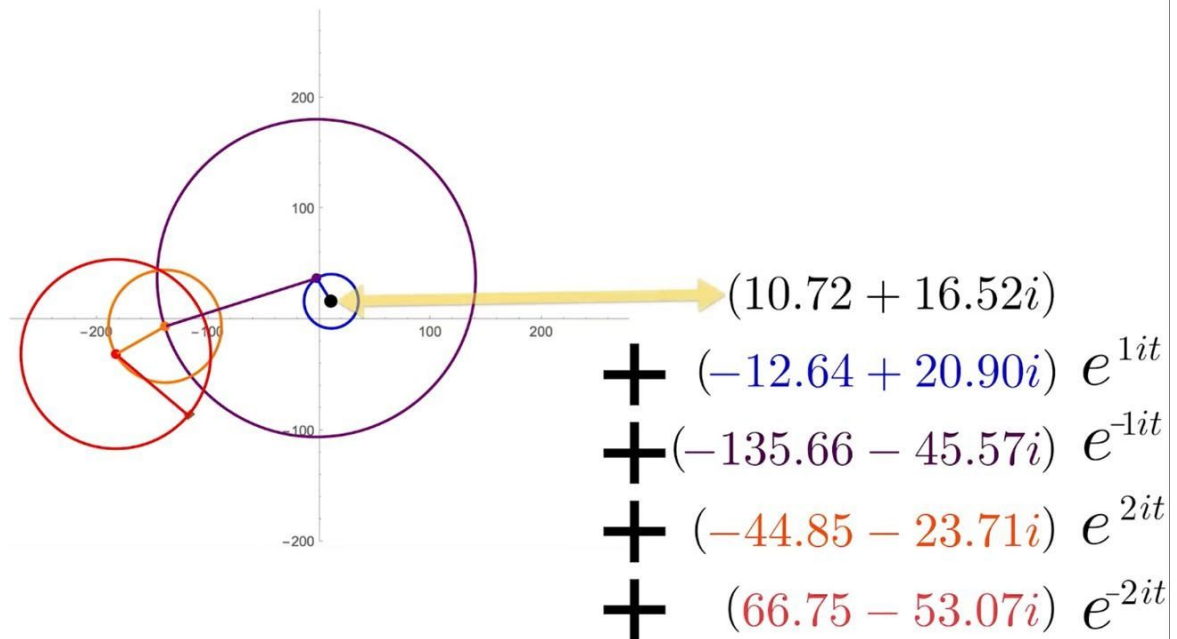


Рисунок 3.1

```
def circle_sorting(order):
    idx = []
    for i in range(1, order+1):
        idx.extend([order+i, order-i])
    return idx
```

3.5 Функция анимации эпициклов

Реализуем функцию для анимации каждого кадра. В качестве координат контура будут выступать значения, вычисленные с помощью преобразования Фурье для каждого кадра, также вычислим координаты начала и конца каждого радиуса для каждого кадра (будем отображать их в виде отрезков), окружности в каждом кадре для удобства будем задавать в параметрическом виде:

$$\begin{cases} x = R \cos(\theta) + x_0 \\ y = R \sin(\theta) + y_0 \end{cases}, \text{ где } (x_0, y_0) - \text{ координаты центра окружности.}$$

```
def animate(i):
    line.set_data(x_DFT[:i], y_DFT[:i])
    r = [np.linalg.norm(coef[j]) for j in range(len(coef))]
    pos = coef[order]
    c = update_c(coef, i / len(space) * (2*pi))
    idx = circle_sorting(order)
    for j, rad, circle in zip(idx, radius, circles):
        new_pos = pos + c[j]
        rad.set_data([pos[0], new_pos[0]], [pos[1], new_pos[1]])
        theta = np.linspace(0, 2*pi, 50)
        x, y = r[j]*np.cos(theta) + pos[0], r[j]*np.sin(theta) + pos[1]
        circle.set_data(x, y)
        pos = new_pos
```

3.6 Визуализация преобразования Фурье в виде анимации

Соберём все функции воедино [см. Приложение 6] и передадим в конструктор *animation.FuncAnimation* нашу область, функцию анимации, количество кадров и интервал обновления. В результате запуска *Python*-скрипта получим нужную нам анимацию (см. Рис. 3.2).

```
plt.title('t.me/FourierTransformBot')
ax.axis('off')
ani = animation.FuncAnimation(fig, animate, frames=len(space), interval=50)
```

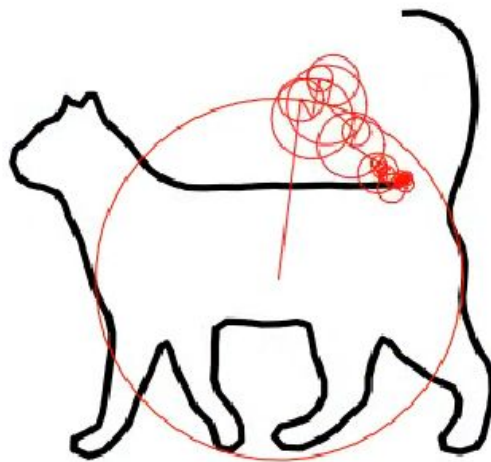


Рисунок 3.2

ГЛАВА 4. КОНВЕРТАЦИЯ АНИМАЦИИ В ВИДЕО

4.1 Формулировка задачи

Для удобного использования нашего преобразования Фурье при реализации *Telegram*-бота создадим функцию конвертации, на вход которой будет поступать данное пользователем изображение, а на выходе будет видео-анимация. Также реализуем многопользовательский режим, то есть возможность работы с ботом одновременно несколькими пользователями. Для этого все данные о пользователе (его входное изображение, действия, итоговая видео-анимация и язык интерфейса) будем хранить в глобальном, загружаемом из серверной системы *Python*-словаре, доступ к которому реализуем по уникальному идентификатору пользователя на платформе *Telegram* (*telegram-id*), чтобы боту было удобно различать действия разных пользователей.

4.2 Многопользовательский режим

При запуске бот будет получать *id* того пользователя, который его запустил. Будем хранить изображения и видео в следующих форматах:

- *{user_id}_before1.jpg* – изображение, отправленное пользователем с идентификатором *user_id*;
- *{user_id}_before2.jpg* – изображение после цветокоррекции пользователя с идентификатором *user_id*;

- *{user_id}_after.jpg* – изображение после нахождения его контура (будем отсылать его обратно пользователю, чтобы он удостоверился в правильности границ и дал своё согласие на дальнейшую обработку);
- *{user_id}_Fourier.mp4* – видео-анимация преобразования.

Для этого немного изменим некоторые функции путём добавления в качестве входного параметра идентификатор пользователя и будем сохранять все данные в вышеописанном формате.

4.3 Конвертация анимации в видеофайл

Реализуем функцию [см. Приложение 7], осуществляющую преобразование Фурье и конвертирующую *matplotlib.animation* в видеофайл формата *mp4*. При реализации бота будем конвертировать преобразование Фурье в видео-анимацию простой строчкой: `get_ani('image')`, где *image* – имя входного изображения.

Для конвертации анимации в видео существует ряд "писателей", которые различаются принципами своей работы:

- 1) *PillowWriter* – основан на библиотеке *Pillow* для записи анимации. Сохраняет все данные в памяти, не очень производителен, но способен работать со многими форматами.
- 2) *FFMpegWriter* – основан на *Pipe*, делает раскадровку. Как правило, самый производительный, но работает не со всеми форматами.
- 3) *ImageMagickWriter* – основан на *ImageMagick*. Такой же по производительности, как и *FFMpeg*, более современный, но работает не на всех системах.

4) *AVConvWriter* – основан на *avconv*. Поддерживает больше форматов, но менее производительный, чем *FFMpeg* и *ImageMagick*.

Тестирование всех писателей показало, что для конвертации в нашем случае лучше всего подойдёт *matplotlib.animation.FFMpegWriter*, так как он наиболее производительный, подходит как для нашей системы, так и для многих серверных систем и поддерживает нужные нам форматы видео (*mov*, *avi*, *mp4*, и т.д.). Однако какие бы наборы программ-писателей мы ни пробовали, конвертация занимает значительное время на фоне выполнения всего алгоритма. То есть только процесс конвертации занимает около 90% времени исполнения программы.

В результате поиска решений оптимизации выяснилось, что время конвертации зависит от трёх параметров:

- уровня аппроксимации (*approx_level*);
- количества кадров в анимации (*frames*);
- количества точек на дюйм (*dpi*).

Если мы будем уменьшать или увеличивать количество точек на дюйм, мы будем терять качество нашей анимации (проверено опытным путём), поэтому оставим значение *dpi* = 150 в качестве рекомендованного. Методом проб и ошибок можно добиться наиболее качественной анимации за минимальное время. При уровне аппроксимации, равном 60 (*approx_level* = 60), контур достаточно похож на исходное изображение, даже сложное, а при количестве кадров, равном 180 (*frames* = 180), анимация достаточно плавная и красивая. Однако, к сожалению, конвертация с данным параметром занимает около 30 секунд, но это довольно хороший результат, по сравнению с другими "писателями" или с более высокими параметрами аппроксимации и кадров. Время обработки можно также ускорить за

счёт понижения качества видео, однако вряд ли стоит жертвовать качеством. В зависимости от сервера, на котором будет размещён бот, время обработки, преобразования и конвертации будет варьироваться. Поэтому для развёртывания нашего проекта мы должны позаботиться о производительном серверном хранилище.

ГЛАВА 5. СОЗДАНИЕ И ОПИСАНИЕ ИНТЕРФЕЙСА TELEGRAM-БОТА

5.1 Кратко о ботах платформы Telegram

Боты в *Telegram* – это разновидность чат-ботов. Это примерно такой же чат, как и с обычным пользователем. По правилам все их имена должны оканчиваться словом «*bot*». По своей сути это те же пользовательские аккаунты, которыми вместо людей управляют программы.

Боты помогают выполнять разные действия: переводить и комментировать, обучать и тестировать, искать и находить, спрашивать и отвечать, играть и развлекать, транслировать и агрегировать, встраиваться в другие сервисы и платформы, взаимодействовать с датчиками и приборами, подключенными к Интернету. Ботов *Telegram* можно добавить в чат-группу, или ими поделиться. И это далеко не все возможности, которые предоставляет *Telegram* для их создания. Идеи ограничиваются фантазией разработчика и, возможно, еще нереализованными возможностями самой платформы. В любом случае, данная платформа отлично подходит для нашего проекта в качестве десктопного или мобильного пользовательского интерфейса.

5.2 Создание интерфейса Telegram-бота

Все боты в *Telegram* создаются с помощью отца всех ботов – *@BotFather*, созданного разработчиками *Telegram*. Создать бота очень просто:

- 1) запускаем *@BotFather*;
- 2) создаём нового бота с помощью команды */newbot*, которую отправляем в виде сообщения;
- 3) придумываем ему имя и короткую ссылку;
- 4) *@BotFather* создаёт чат нашего бота и генерирует строку-токен для доступа к нему при реализации;
- 5) при желании можно добавить аватар, описание и список команд во всплывающей подсказке.

После создания бота и получения его токена установим команды (см. Рис. 5.1):

- *\start* – запуск бота;
- *\commands* – вывод списка поддерживаемых команд;
- *\language* – установка языка бота (бот будет поддерживать русский, английский и немецкий языки);
- *\about* – вывод краткой информации о преобразовании Фурье;
- *\examples* – примеры входных изображений ;
- *\help* – справочная информация.

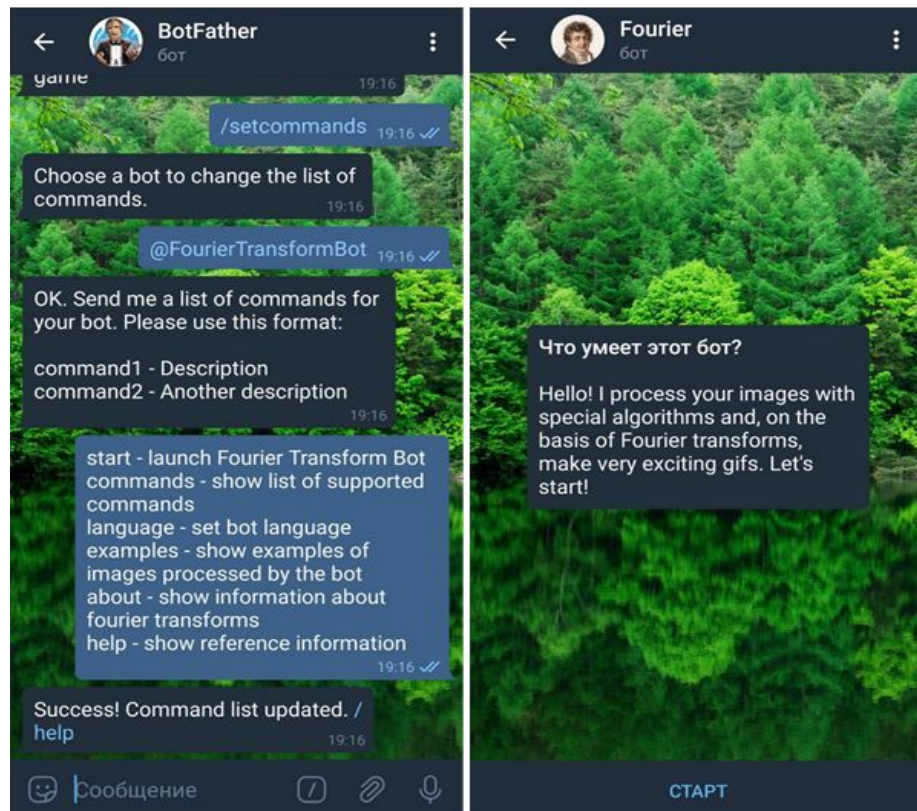


Рисунок 5.1

Бот успешно создан. Его можно найти по имени *@FourierTransformBot*. Если его запустить, то он не будет реагировать на наши действия, поэтому мы должны описать логику его работы посредством специально разработанного для языка *Python* интерфейса – *pyTelegramBotAPI* [11].

ГЛАВА 6. РЕАЛИЗАЦИЯ TELEGRAM-БОТА

6.1 Реализация мультиязычности и хранения данных

Для начала решим вопрос "мультиязычности" и хранения настроек для каждого пользователя. Чтобы нам не пришлось работать с каждым языком отдельно, создадим словари, в которые будем записывать нужные нам блоки текста на разных языках и позже просто обращаться по ключу. Эти словари поместим в список, где нулевым элементом списка будет словарь с русским текстом, первым – с английским и вторым – с немецким. Например, если нужно будет вывести немецкий текст для команды *start*, то мы напишем `languages[2]['start']`. Чтобы сохранять язык для каждого пользователя, использующего бот, будем хранить индексы языкового списка `languages` по *telegram-id* пользователя.

```
russian = {'command1': 'text1'}
english = {'command2': 'text2'}
deutsch = {'command3': 'text3'}
languages = [russian, english, deutsch]

user_language = {'user1_id': 0, 'user2_id': 1, 'user3_id': 2}
```

Также по *telegram-id* пользователя будем хранить и изображения с видео-анимацией.

Если вдруг возникнут какие-то ошибки на сервере, или пользователь захочет удалить чат с ботом и впоследствии запустить его обратно, нашему боту нужно вспомнить пользователя и восстановить его языковые настройки. Так как это не масштабные системные данные, требующие определённой иерархии, мы не будем

создавать базу данных. Для этого будем сохранять закодированный словарь с данными в файл, который будет храниться у нас на сервере. Бот сможет "вспомнить" пользователя по его *id*, загрузив информацию из файла.

Реализуем методы сохранения в файл и загрузки из файла нужной нам информации [см. Приложение 8]. Сам файл с данными назовём *dataset.txt*.

6.2 Взаимодействие с пользователем и описание команд

При реализации бота будем использовать встроенный интерфейс от разработчиков *Telegram* – *TelegramBotAPI* [\[11\]](#) и библиотеку *telebot*, поддерживающую его. Импортируем все ранее подготовленные модули в основной файл проекта:

- *DFT.py* – функции для анимации преобразования Фурье;
- *PSPNet.py* – модель нейросети для распознавания контуров;
- *data_tools.py* – методы безопасного сохранения и выгрузки информации;
- *bot_lang.py* – языковые словари;
- *telebot* – API разработки телеграмм ботов;
- *logging* – библиотека для логгирования ошибок на сервер;
- *os* – для работы с серверной системой.

После подключения модулей создадим экземпляр *Telegram*-бота с помощью заранее известного нам токена, чтобы оповестить сервисы *Telegram*, что реализация относится к созданному нами боту, а также экземпляр нейросетевой модели *PSP-Net*.

```
TOKEN = '*****'
```

```
bot = telebot.TeleBot(TOKEN)
pspn_model, class_colors = pspn.load_PSPNet()
```

Чтобы пользователю было удобнее взаимодействовать с функционалом бота, например, установить язык и дать согласие на поставленную боту задачу, опишем всплывающие кнопки-подсказки в виде плиток с надписями, которые будем вызывать в нужных нам местах.

```
lang_keyboard = telebot.types.ReplyKeyboardMarkup(True, True, True)
lang_keyboard.row('Русский', 'English', 'Deutsch')
...
```

Кроме всплывающих кнопок, взаимодействие с ботом будет осуществляться посредством команд, каждую из которых предстоит запрограммировать на нужное действие:

- */start* – будет вызываться сразу после запуска бота и загружать настройки бота из файла *dataset.txt*. Если бот ранее не общался с пользователем, то он предложит пользователю установить язык с помощью всплывающей подсказки. После выбора языка пользователю будет высылаться текст, описывающий навигацию и бота, а так же видео-пример возможностей бота;
- */help* – будет высылать вспомогательную информацию по навигации в интерфейсе бота;
- */commands* – будет выводить список поддерживаемых ботом команд;
- */language* – предложит изменить язык интерфейса бота;
- */examples* – отправит в виде альбома ранее заготовленные примеры картинок для обработки ботом. Примеры будут храниться на сервере;
- */about* – будет отправлять подробную информацию о преобразовании Фурье и некоторые формулы.

При разработке обработчиков команд используются специальные хэндлеры, в которых указывается имя события или команды. В случае вызова команды пользователем через интерфейс бота вызывается соответствующая функция, описывающая функционал команды. Пример реализации команды */commands*:

```
@bot.message_handler(commands=['commands'])
def bot_commands(message):
    user_language.update(data.load_dataset())
    bot.send_message(message.chat.id,
    languages[user_language[message.chat.id]]['commands'])
```

6.3 Обработчики событий

Запрограммируем две основных реакции на действия пользователя: отправка фотографии и текста. Все остальные события, такие как отправка документа, аудиофайла, стикера или голосового сообщения и т. п. обрабатывать не будем, а просто сообщим пользователю о нераспознавании команды.

По нашей задумке полученная ботом фотография должна обрабатываться в видео-анимацию преобразования Фурье. При отправке пользователем изображения бот будет скачивать его на сервер в формате *{user_id}_before1.jpg* и там же обрабатывать, получая цветосегментированное изображение *{user_id}_before2.jpg* и замкнутый контур изображения в формате *{user_id}_after.jpg*, который пересылается обратно пользователю. Далее бот будет спрашивать, удовлетворяет ли пользователя обработка фотографии каким либо алгоритмом, вызывая всплывающие подсказки на соответствующем языке. Все текстовые запросы будем позже обрабатывать как отдельное событие. Также

предусмотрим обработку исключительных ситуаций, например: если пользователь отправил боту однотонное изображение, не имеющее контур, то бот сообщит ему о том, что на картинке не обнаружено каких-либо объектов и контуров. Если же пользователя не устроит ни один вариант обработки изображения, то бот будет предлагать обработать изображение нейросетью. Если пользователь согласится на эту процедуру, то нейросеть преобразует исходное изображение в цветосегментированное, после чего к результату применится цветовой фильтр и повторится обработка ранее описанными алгоритмами.

Опишем событие, обрабатывающее все текстовые запросы от пользователя. Итак, если пользователь даст положительный ответ на один из полученных алгоритмом контуров, тогда бот уберёт всплывающую подсказку, выведет соответствующий текст и начнёт конвертацию выбранного пользователем изображения в видео-анимацию, после которой отправит её пользователю и удалит данные, полученные в процессе обработки, дабы не захламлять сервер.

Если же пользователь даст отрицательный ответ на полученные алгоритмом контуры, то есть не согласится, тогда бот также уберёт всплывающую подсказку и предложит обработать изображение нейросетью.

Если пользователь даст отрицательный ответ и на эту процедуру, то бот выведет соответствующий текст и удалит все данные об изображении.

Когда пользователь выберет один из языков, бот сохранит этот язык в глобальный словарь с языковыми настройками и также в файл на сервере, выводя соответствующее сообщение. Если боту было отправлено однотонное изображение, не имеющее контура, то выведется соответствующее сообщение о том, что не удалось обнаружить какие-либо объекты на картинке.

Более подробно разобраться в архитектуре обработчиков поможет схема на Рисунке 6.1.

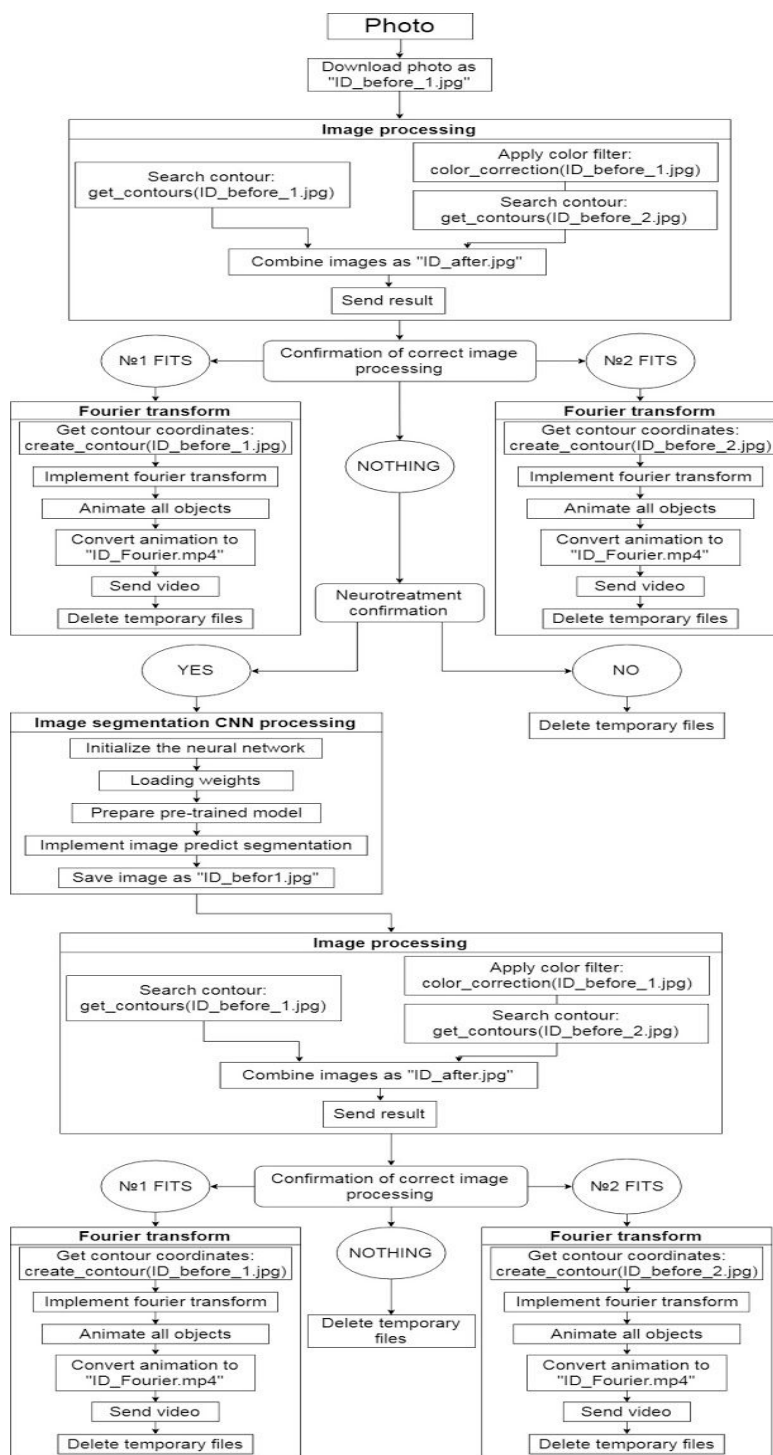


Рисунок 6.1

Если пользователь введёт что-то непредусмотренное в реализации бота, то выведется сообщение о нераспознавании команды.

```
@bot.message_handler(func=lambda message: True, content_types=['document',  
'audio', 'sticker', 'voice'])  
def bot_get_other(message):  
    bot.reply_to(message, languages[user_language[message.chat.id]]['other'])
```

6.4 Использование бота

После реализации всех обработчиков событий осталось запустить бота в режиме постоянной обработки информации, приходящей от серверов *Telegram*. В случае любой возникшей ошибки бот перезапустится и отправит лог ошибок в серверную консоль.

```
while True:  
    try:  
        bot.polling(none_stop=True)  
    except Exception as e:  
        logging.error(e)
```

Бот готов (см. Рис. 6.2). Проверить его работу можно, запустив написанный нами скрипт в *python*-интерпретаторе.

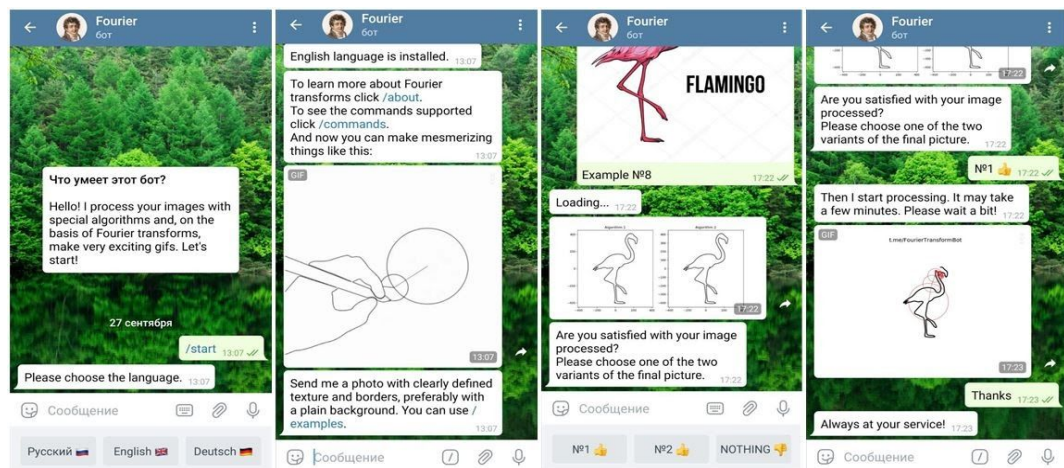


Рисунок 6.2

ГЛАВА 7. РАЗВЁРТЫВАНИЕ TELEGRAM-БОТА НА СЕРВЕРЕ

7.1 Подготовка данных

Так как мы уже полностью реализовали *Telegram*-бота, осталось загрузить его на сервер, чтобы он смог работать без помощи нашего компьютера. Создадим папку, в которой распределим все необходимые *Telegram*-боту файлы (см. Рис. 7.1):

- примеры изображений (*001.jpg-010.jpg*);
- аватар бота (*avatar.jpg*);
- языковой словарь (*bot_lang.py*);
- коды функций сохранения настроек бота (*data_tools.py*);
- текстовый документ с настройками бота (*dataset.txt*);
- коды обработки входного изображения (*DFT.py*);
- коды преобразования Фурье (*DFT_tools.py*);
- изображение главных формул (*formula.jpg*);
- реализация *Telegram*-бота (*main.py*);
- файлы зависимостей, весов и скриптов модели нейросети (*PSPNet.py*);
- видео с возможностями бота (*start.mp4*).





















 001.jpg	03.05.2020 13:26	Рисунок JPEG	10 КБ
 002.jpg	03.05.2020 13:26	Рисунок JPEG	111 КБ
 003.jpg	03.05.2020 13:26	Рисунок JPEG	680 КБ
 004.jpg	03.05.2020 13:26	Рисунок JPEG	71 КБ
 005.jpg	03.05.2020 13:26	Рисунок JPEG	23 КБ
 006.jpg	03.05.2020 13:26	Рисунок JPEG	54 КБ
 007.jpg	03.05.2020 13:26	Рисунок JPEG	78 КБ
 008.jpg	03.05.2020 13:26	Рисунок JPEG	61 КБ
 009.jpg	03.05.2020 13:26	Рисунок JPEG	79 КБ
 010.jpg	03.05.2020 13:26	Рисунок JPEG	54 КБ
 avatar.jpg	03.05.2020 13:26	Рисунок JPEG	71 КБ
 bot_lang.py	03.05.2020 13:26	Python File	11 КБ
 data_tools.py	03.05.2020 13:26	Python File	1 КБ
 dataset.txt	03.05.2020 13:26	Текстовый документ	1 КБ
 DFT.py	03.05.2020 13:26	Python File	2 КБ
 DFT_tools.py	03.05.2020 13:26	Python File	4 КБ
 formula.jpg	03.05.2020 13:26	Рисунок JPEG	257 КБ
 main.py	03.05.2020 13:26	Python File	13 КБ
 PSPNet.py	03.05.2020 13:26	Python File	3 КБ
 start.mp4	03.05.2020 13:26	Файл "MP4"	421 КБ

Рисунок 7.1

7.2 Файл зависимостей бота

В каждом языке программирования есть свои файлы, помогающие правильно установить окружение приложений. В *Python* таким файлом является *requirements.txt* – файл зависимостей приложения для автоматической установки пакетов *Python* с помощью утилиты *pip*. Данный формат является обычным текстовым файлом, где указано название пакета *Python*, его версия и условия: больше, меньше, равно.

Установка указанных зависимостей будет осуществляться сервером с помощью файла *requirements.txt*, который мы и создадим.

Так как в процессе создания бота мы работали в окружении, то все используемые нами зависимости сохранялись там (для создания окружения используют *conda create -n env_name python=3.7*). С помощью консоли *Anaconda* для *Windows* мы можем просмотреть все зависимости в нашем проекте.

```
$ conda activate fourier_env  
$ pip freeze > requirements.txt
```

В результате этих команд в директории с файлами нашего *Telegram*-бота автоматически создаётся файл зависимостей *requirements.txt*, с помощью которого мы сможем легко установить *Python* и нужные библиотеки для работы бота на сервер.

7.3 Файл запуска бота

Файл запуска приложения (*Procfile*) – это механизм объявления команд, выполняемых приложениями на различных платформах. *Procfile* используется, чтобы сообщить серверу, как запускать различные части нашего приложения. Для этого создадим текстовый файл, назовём его *Procfile* и запишем в него одну строку.

```
worker: python main.py
```

Синтаксис довольно прост: часть слева от двоеточия на каждой строке – тип процесса, часть справа – команда для запуска этого процесса. С помощью этого файла сервер будет знать, какой скрипт ему запускать. В данном случае это файл с реализацией *Telegram*-бота (*main.py*).

7.4 GitHub как хранилище данных

Так как веб-сервис *GitHub* – отличная система контроля версий IT-проектов, будем хранить и при случае обновлять нашего бота именно на *GitHub*. Также через него будет очень удобно размещать файлы на хостинге. Создадим репозиторий и загрузим в него файлы нашего бота. Все дальнейшие релизы нашего проекта мы сможем эффективно экспортировать на сервер.

7.5 Развёртывание бота на Heroku

Heroku – облачная *PaaS*-платформа, которая позволяет разработчикам создавать, запускать и эксплуатировать приложения исключительно в облаке. Этот сервис любят за то, что мы можем протестировать свои проекты в режиме "*production*". Это дает нам гарантию, что наш проект будет работать стабильно, когда мы его запустим в "боевом режиме".

Особой разницы в размещении бота на *Heroku* или на отдельном сервере нет, однако *Heroku* предоставляет нам следующие бесплатные услуги:

- поддержка до 5 бесплатных приложений;
- 1000 бесплатных часов работы приложений;
- пользовательские домены для каждого приложения;
- доступ к панели инструментов *Heroku* и интерфейсу командной строки для управления приложениями;
- 512 мегабайт ОЗУ.

Этого функционала нам вполне достаточно. Начало достаточно простое:

1. Регистрируемся на сайте *Heroku*.
2. Входим в свой аккаунт и создаём проект, нажав на "*Create new app*".
3. Называем наш проект "*fourier-transform-bot*". Название должно быть уникальным. Выбор региона не принципиален.
4. После создания проекта нам будут предложены варианты размещения проекта на *Heroku*. А именно: *HerokuGit* или *Container Registry* – с помощью *CLI* от *Heroku* и *GitHub* – подключив свой аккаунт *GitHub* и клонировав репозиторий на сервер *Heroku* (см. Рис. 7.2).

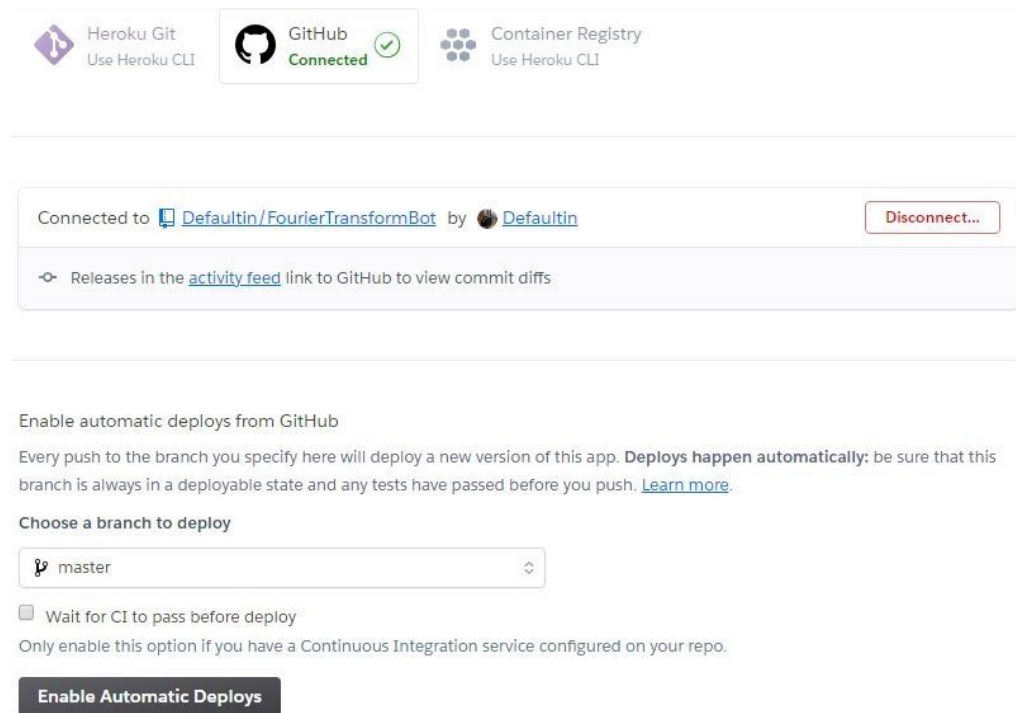


Рисунок 7.2

В настройках (см. Рис. 7.3) добавляем дополнительные пакеты (*buildpack*) для языка *Python* и писателя *matplotlib.animation ffmpeg* (выберем

последнюю версию на *GitHub*) и нажимаем *Deploy Branch*, тем самым начиная загрузку *Telegram*-бота на сервер *Heroku*.



Рисунок 7.3

После успешной загрузки во вкладке "*Resources*" запускаем нашего бота простым перемещением тумблера (см. Рис. 7.4). Теперь наш бот полностью функционирует самостоятельно. Платформа *Heroku* предоставляет примерно месяц бесплатного обслуживания нашего бота с ограничением вычислительной мощности на сервере. Вычислительного потенциала бесплатного периода достаточно, чтобы бот мог обслуживать до четырёх или пяти пользователей одновременно.



Рисунок 7.4

ЗАКЛЮЧЕНИЕ

В данной работе на основе дискретных преобразований Фурье, возможностей языка *Python*, различных методов обработки и нейросегментации изображений, а также опций для создания бота, предоставленных платформой *Telegram*, был создан многопользовательский интерактивный бот, конвертирующий пользовательские изображения в увлекательные видео-анимации. Реализованный бот позволяет пользователю выбирать один из трёх наиболее распространённых в мировом сообществе языков интерфейса, предоставляет различные варианты нахождения контуров объектов на изображении и несколько примеров изображений, которые он может обработать, а также умеет реагировать как на картинки, контур на которых выделить невозможно (это, как правило, однотонные текстуры), так и на сообщения различного характера и содержания. Помимо этого, на любом из предлагаемых языков пользователь может прочесть подробную информацию о функционале бота и краткое теоретическое введение в преобразования Фурье.

Созданный *Telegram*-бот решил сразу несколько задач. Во-первых, это интересная и наглядная демонстрация возможностей преобразований Фурье, которая может быть использована в образовательных целях, то есть для изучения самих преобразований или для интересной визуализации прорисовки любого контура. Во-вторых, это функционирующая система сегментации, которая способна выделить контур практически на любом изображении. А в-третьих, это пример подробно разработанного мультиязычного и многопользовательского *Telegram*-бота, шаблон которого можно применить практически для любых нужд.

На данный момент бот доступен в Telegram по имени *@FourierTransformBot*.
Со всеми подробностями реализации, а также с исходным кодом можно
ознакомиться в моём [GitHub-репозитории](#).

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Документация Matplotlib:
 - <https://matplotlib.org/3.1.1/api/index.html>
2. Документация Numpy:
 - <https://docs.scipy.org/doc/numpy/reference/routines.html>
3. Документация Math:
 - <https://docs.python.org/3/library/math.html>
4. Документация Pillow:
 - <https://pillow.readthedocs.io/en/3.1.x/reference/Image.html>
5. Документация OpenCV:
 - <https://docs.opencv.org/4.1.2/>
6. Документация Keras:
 - <https://keras.io/>
7. Image segmentation CNN models:
 - <https://divamgupta.com/image-segmentation/2019/06/06/deep-learning-semantic-segmentation-keras.html>
 - <https://github.com/divamgupta/image-segmentation-keras>
8. Преобразования Фурье:
 - https://en.wikipedia.org/wiki/Fourier_transform
 - https://en.wikipedia.org/wiki/Discrete_Fourier_transform

- <https://betterexplained.com/articles/an-interactive-guide-to-the-fourier-transform/>
- <http://www.di.fc.ul.pt/~jpn/r/fourier/fourier.html#examples>
- <http://www.jezzamon.com/fourier/>
- <https://youtu.be/FCTVloBp7uw>
- <https://youtu.be/spUNpyF58BY>
- <https://youtu.be/qS4H6PEcCCA>

9. Документация по видео-писателям:

- https://matplotlib.org/3.1.1/api/animation_api.html#writer-classes

10. Telegram Bots:

- <https://habr.com/ru/post/448310/>

11. Документация pyTelegramBotAPI:

- <https://github.com/eteranno/pyTelegramBotAPI>
- <https://core.telegram.org/bots/api>

12. Руководство зависимостям Python:

- <https://medium.com/@boscacci/why-and-how-to-make-a-requirements-txt-f329c685181e>

13. Heroku Deployment:

- <https://devcenter.heroku.com/articles/getting-started-with-python?singlepage=true>

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ 1.

```
def create_contour(image_name, *, level=[200]):
    image = Image.open(image_name).convert('L')
    im = array(image)
    image.close()

    contour_plot = contour(im, levels=level, colors='black', origin='image')
    contour_path = contour_plot.collections[0].get_paths()[0]
    x_table, y_table = contour_path.vertices[:, 0], contour_path.vertices[:, 1]
    time_table = np.linspace(0, 2*pi, len(x_table))

    x_table = x_table - min(x_table)
    y_table = y_table - min(y_table)
    x_table = x_table - max(x_table) / 2
    y_table = y_table - max(y_table) / 2

    return time_table, x_table, y_table
```

ПРИЛОЖЕНИЕ 2.

```
def color_correction(image_name, *, ID=None, reverse=True, out_name='befor2'):
    green = np.uint8([[0, 255, 0]])
    green_hsv = cv2.cvtColor(green, cv2.COLOR_BGR2HSV)
    image = cv2.imread(image_name)
    image = ~image if reverse else image
    hsv_img = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    green_low = np.array([45, 100, 50])
    green_high = np.array([75, 255, 255])
    curr_mask = cv2.inRange(hsv_img, green_low, green_high)
    hsv_img[curr_mask > 0] = ([75, 255, 200])
```

```

RGB_again = cv2.cvtColor(hsv_img, cv2.COLOR_HSV2RGB)
gray = cv2.cvtColor(RGB_again, cv2.COLOR_RGB2GRAY)
ret, threshold = cv2.threshold(gray, 90, 255, 0)
cv2.imwrite(f'{ID}_ {out_name}.jpg', threshold)

```

ПРИЛОЖЕНИЕ 3.

```

from keras.models import load_model
from keras.utils import get_file
from keras import layers
from keras.backend import tf as ktf
import cv2
import numpy as np
import random
__all__ = ('load_PSPNet', 'start_neuro_segmentation')

class Interp(layers.Layer):
    def __init__(self, new_size, **kwargs):
        self.new_size = new_size
        super(Interp, self).__init__(**kwargs)

    def build(self, input_shape):
        super(Interp, self).build(input_shape)

    def call(self, inputs, **kwargs):
        new_height, new_width = self.new_size
        resized = ktf.image.resize_images(inputs, [new_height, new_width],
align_corners=True)
        return resized

    def compute_output_shape(self, input_shape):
        return tuple([None, self.new_size[0], self.new_size[1], input_shape[3]])

    def get_config(self):
        config = super(Interp, self).get_config()
        config['new_size'] = self.new_size

```

```

        return config

def get_image_arr(path, width, height):
    if type(path) is np.ndarray:
        img = path
    else:
        img = cv2.imread(path, 1)

    img = cv2.resize(img, (width, height))
    img = img.astype(np.float32)
    img[:, :, 0] -= 103.939
    img[:, :, 1] -= 116.779
    img[:, :, 2] -= 123.68
    img = img[:, :, ::-1]
    return img

def load_PSPNet():
    random.seed(0)
    class_colors =
    [(random.randint(0,255),random.randint(0,255),random.randint(0,255)) for _ in
    range(5000)]
    model_url =
    "https://getfile.dokpub.com/yandex/get/https://yadi.sk/d/BR1EAlZ-UQMzQQ"
    model_config = get_file('PSP-Net.h5', model_url)
    model = load_model(model_config, custom_objects={'Interp': Interp})
    global graph
    graph = ktf.get_default_graph()
    return model, class_colors

def start_neuro_segmentation(input_name, model, colors, *, ID=None):
    inp = cv2.imread(input_name)
    orininal_h = inp.shape[0]
    orininal_w = inp.shape[1]
    output_width = output_height = input_width = input_height = 473
    n_classes = 21

    x = get_image_arr(inp, input_width, input_height)
    with graph.as_default():
        pr = model.predict(np.array([x]))[0]
    pr = pr.reshape((output_height, output_width, n_classes)).argmax(axis=2)

```

```

seg_img = np.zeros((output_height, output_width, 3))

for c in range(n_classes):
    seg_img[:, :, 0] += ((pr[:, :] == c)*(colors[c][0])).astype('uint8')
    seg_img[:, :, 1] += ((pr[:, :] == c)*(colors[c][1])).astype('uint8')
    seg_img[:, :, 2] += ((pr[:, :] == c)*(colors[c][2])).astype('uint8')

seg_img = cv2.resize(seg_img, (orininal_w, orininal_h))
cv2.imwrite(f'{ID}_befor1.jpg', seg_img)

```

ПРИЛОЖЕНИЕ 4.

```

def get_contours(image_name, *, ID=None):
    color_correction(image_name, ID=ID, reverse=True)
    contour_1 = create_contour(image_name)
    contour_2 = create_contour(f'{ID}_before2.jpg')

    fig, ax = plt.subplots(1, 2, figsize=(10, 5))
    ax[0].set_aspect('equal', 'datalim')
    ax[1].set_aspect('equal', 'datalim')
    ax[0].set_title('Algorithm 1')
    ax[1].set_title('Algorithm 2')
    ax[0].plot(contour_1[1], contour_1[2], 'k-')
    ax[1].plot(contour_2[1], contour_2[2], 'k-')
    plt.savefig(f'{ID}_after.jpg')
    plt.close('all')

```

ПРИЛОЖЕНИЕ 5.

```

def DFT(t, coef_list, order=10):
    kernel = np.array([np.exp(-n*1j*t) for n in range(-order, order+1)])
    series = np.sum((coef_list[:, 0]+1j*coef_list[:, 1]) * kernel[:])
    return np.real(series), np.imag(series)

```

ПРИЛОЖЕНИЕ 6.

```
def visualize(x_DFT, y_DFT, coef, order, space, fig_lim):
    fig, ax = plt.subplots()
    lim = 3*max(fig_lim)/2
    ax.set_xlim([-lim, lim])
    ax.set_ylim([-lim, lim])
    ax.set_aspect('equal')

    line = plt.plot([], [], 'k-', linewidth=2)[0]
    radius = [plt.plot([], [], 'r-', linewidth=0.5)[0]
               for _ in range(2 * order + 1)]
    circles = [plt.plot([], [], 'r-', linewidth=0.5)[0]
                for _ in range(2 * order + 1)]

def update_c(c, t):
    new_c = []
    for i, j in enumerate(range(-order, order + 1)):
        theta = -j * t
        cos, sin = np.cos(theta), np.sin(theta)
        v = [cos * c[i][0] - sin * c[i][1], sin * c[i][0] + cos * c[i][1]]
        new_c.append(v)
    return np.array(new_c)

def circle_sorting(order):
    idx = []
    for i in range(1, order+1):
        idx.extend([order+i, order-i])
    return idx

def animate(i):
    line.set_data(x_DFT[:i], y_DFT[:i])
    r = [np.linalg.norm(coef[j]) for j in range(len(coef))]
    pos = coef[order]
    c = update_c(coef, i / len(space) * (2*pi))
    idx = circle_sorting(order)
    for j, rad, circle in zip(idx, radius, circles):
        new_pos = pos + c[j]
        rad.set_data([pos[0], new_pos[0]], [pos[1], new_pos[1]])
        theta = np.linspace(0, 2*pi, 50)
        x, y = r[j] * np.cos(theta) + pos[0], r[j] * np.sin(theta) + pos[1]
```

```

        circle.set_data(x, y)
        pos = new_pos

plt.title('t.me/FourierTransformBot')
ax.axis('off')
ani = animation.FuncAnimation(fig, animate, frames=len(space), interval=50)
return ani

```

ПРИЛОЖЕНИЕ 7.

```

def get_ani(image, *, ID=None, approx_level=60, frames=180):
    time_table, x_table, y_table = create_contour(image)
    coef = coef_list(time_table, x_table, y_table, approx_level)
    space = np.linspace(0, 2*pi, frames)
    x_DFT = [DFT(t, coef, approx_level)[0] for t in space]
    y_DFT = [DFT(t, coef, approx_level)[1] for t in space]

    plt.close('all')
    plt.plot(x_table, y_table)
    xmin, xmax = xlim()
    ymin, ymax = ylim()

    anim = visualize(x_DFT, y_DFT, coef, approx_level, space, [xmin, xmax, ymin,
ymax])
    FFwriter = animation.FFMpegWriter(fps=24, extra_args=['-vcodec', 'libx264'])
    anim.save(f'{ID}_Fourier.mp4', writer=FFwriter, dpi=150)
    plt.close('all')

```

ПРИЛОЖЕНИЕ 8.

```

def save_dataset(new_dataset):
    dataset = load_dataset()

```

```
dataset.update(new_dataset)
in_file = open('dataset.txt', 'wb')
pickle.dump(dataset, in_file)
in_file.close()

def load_dataset():
    out_file = open('dataset.txt', 'rb')
    dataset = pickle.load(out_file)
    out_file.close()
    return dataset
```