ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

КЫРГЫЗСКО-РОССИЙСКИЙ СЛАВЯНСКИЙ УНИВЕРСИТЕТ

ЕСТЕСТВЕННО-ТЕХНИЧЕСКИЙ ФАКУЛЬТЕТ

Кафедра информационных и вычислительных технологий

С. Н. ВЕРЗУНОВ, М. С. ОСМОНОВ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКИХ РАБОТ ПО КУРСУ «ЦИФРОВАЯ ОБРАБОТКА СИГНАЛОВ»

Посвящается
25-летию КРСУ

Бишкек 2018

УДК 621.391.083.92
ББК 32.811.3
В 31

Рецензенты:
И. В. Брякин – д-р техн. наук, проф.,
Н. М. Лыченко – д-р техн. наук, доц.

Рекомендовано к изданию
кафедрой информационных и вычислительных технологий,
Ученым советом Естественно-технического факультета КРСУ

Верзунов С. Н., Осмонов М. С.

В 31 УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКИХ РАБОТ ПО КУРСУ «ЦИФРОВАЯ ОБРАБОТКА СИГНАЛОВ». Бишкек: КРСУ, 2018. 72 с.: ил.

ISBN 978-9967-19-576-9

Учебно-методическое пособие охватывает основные разделы учебной программы по курсу «Цифровая обработка сигналов». Рассматриваются вопросы программирования цифровых фильтров обработки изображений с применением алгоритма свертки. Изучается сегментация изображений с помощью порогов и динамическая сегментация изображений. В пособии достаточно глубоко рассмотрены различные алгоритмы дискретного преобразования Фурье и их применение для спектрального анализа сигналов. Изучаются вопросы использования вейвлет-преобразования, в том числе применение алгоритма вейвлет-преобразования с помощью быстрого преобразования Фурье для вейлет-анализа временных рядов.

Пособие отличает большое количество примеров выполнения заданий на языке Python и его четко выверенная профессиональная направленность.

Учебно-методическое пособие для выполнения практических работ по курсу «Цифровая обработка сигналов» может быть рекомендовано для подготовки студентов по направлению магистратуры 710400 (09.04.04) Программная инженерия (Программа: Разработка программно-информационных систем).

В 2303040000-18

УДК 621.391.083.92
ББК 32.811.3

ISBN 978-9967-19-576-9

СОДЕРЖАНИЕ

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

БПФ – Быстрое преобразование Фурье (в англоязычной литературе FFT, Fast Fourier Transform) – алгоритм быстрого вычисления дискретного преобразования Фурье (ДПФ).

ДПФ – Дискретное преобразование Фурье (в англоязычной литературе DFT, Discrete Fourier Transform) – это одно из преобразований Фурье, широко применяемое в алгоритмах цифровой обработки сигналов.

MP3 – кодек третьего уровня, разработанный командой MPEG, формат файла для хранения аудиоинформации.

JPEG – один из популярных растровых графических форматов, применяемый для хранения фотоизображений и подобных им изображений.

RGB – аббревиатура английских слов red, green, blue – красный, зеленый, синий) или КЗС – аддитивная цветовая модель, как правило, описывающая способ кодирования цвета для цветовоспроизведения.

ВВЕДЕНИЕ

Учебно-методическое пособие включает все основные разделы учебной программы по курсу «Цифровая обработка сигналов». Задания, приведенные в пособии, имеют своей целью выработать практические навыки применения основных положений теории цифровой обработки сигналов и ее методов. Приведены также дополнительные задания для самостоятельной работы с целью углубленного изучения отдельных алгоритмов цифровой обработки сигналов.

Практическая работа № 1 посвящена изучению цифровых фильтров обработки изображений с применением алгоритма свертки.

В практической работе № 2 осуществляется сегментация изображений с помощью порогов и динамическая сегментация изображений.

Практическая работа № 3 включает различные алгоритмы дискретного преобразования Фурье и их применение для спектрального анализа сигналов.

Практическая работа № 4 включает исследование вейвлет-преобразования, в том числе использование алгоритма вейвлет-преобразования с помощью быстрого преобразования Фурье для вейлет-анализа временных рядов.

В пособие приведены примеры выполнения заданий на языке Python, с использованием расширений для матричных вычислений и вывода графики – NumPy и Matplotlib и интерактивной среды разработки Jupyter. Исходный код всех примеров можно найти по адресу https://github.com/verzunov/dsp.

ПРАКТИЧЕСКАЯ РАБОТА № 1.
АЛГОРИТМЫ ОБРАБОТКИ ИЗОБРАЖЕНИЙ

Операция свертки

Свертка – это алгоритм очень широкого применения, который можно использовать как для предварительной обработки изображения, так и для распознавания и идентификации объектов. Пусть изображение задается двумерной матрицей яркостей **F'**, а импульсная характеристика матрицей **H**. Математически свертку матрицы **F** с ядром **H** можно определить следующей формулой:

$$r(i,j) = \sum_{m=-\frac{M_2-1}{2}}^{\frac{M_2-1}{2}} \sum_{n=-\frac{N_2-1}{2}}^{\frac{N_2-1}{2}} f(i+m, j+n)h(m,n),$$

где $M_2 \times N_2$ – размер матрицы ядра свертки.

Размер матрицы F равен $(M_1 + M_2 - 1) \times (N_1 + N_2 - 1)$, где $M_1 \times N_1$ – размер исходной матрицы **F'**. Матрица **F** получается из исходной путем добавления элементов на краях матрицы по некоторому правилу с тем, чтобы привести ее к необходимому размеру. Обычно исходная матрица на краях дополняется нулями на половину ширины матрицы **H** влево и вправо и соответственно на половину высоты вверх и настолько же вниз. Тогда размер полученной матрицы **R** будет таким же, как и у матрицы **F'**.

Свертку можно вычислять непосредственно «пробеганием» одной матрицы по другой, как уже было показано выше. На рисунке 1.1 показана схема вычисления свертки (размер матрицы маски взят равным 3 × 3). Оператор свертки можно рассматривать как матрицу коэффициентов (маску), которая поэлементно умножается с выделенным фрагментом изображения с последующим суммированием для получения нового значения элемента отфильтрованного изображения. Эта матрица может быть произвольного размера, необязательно квадратная.

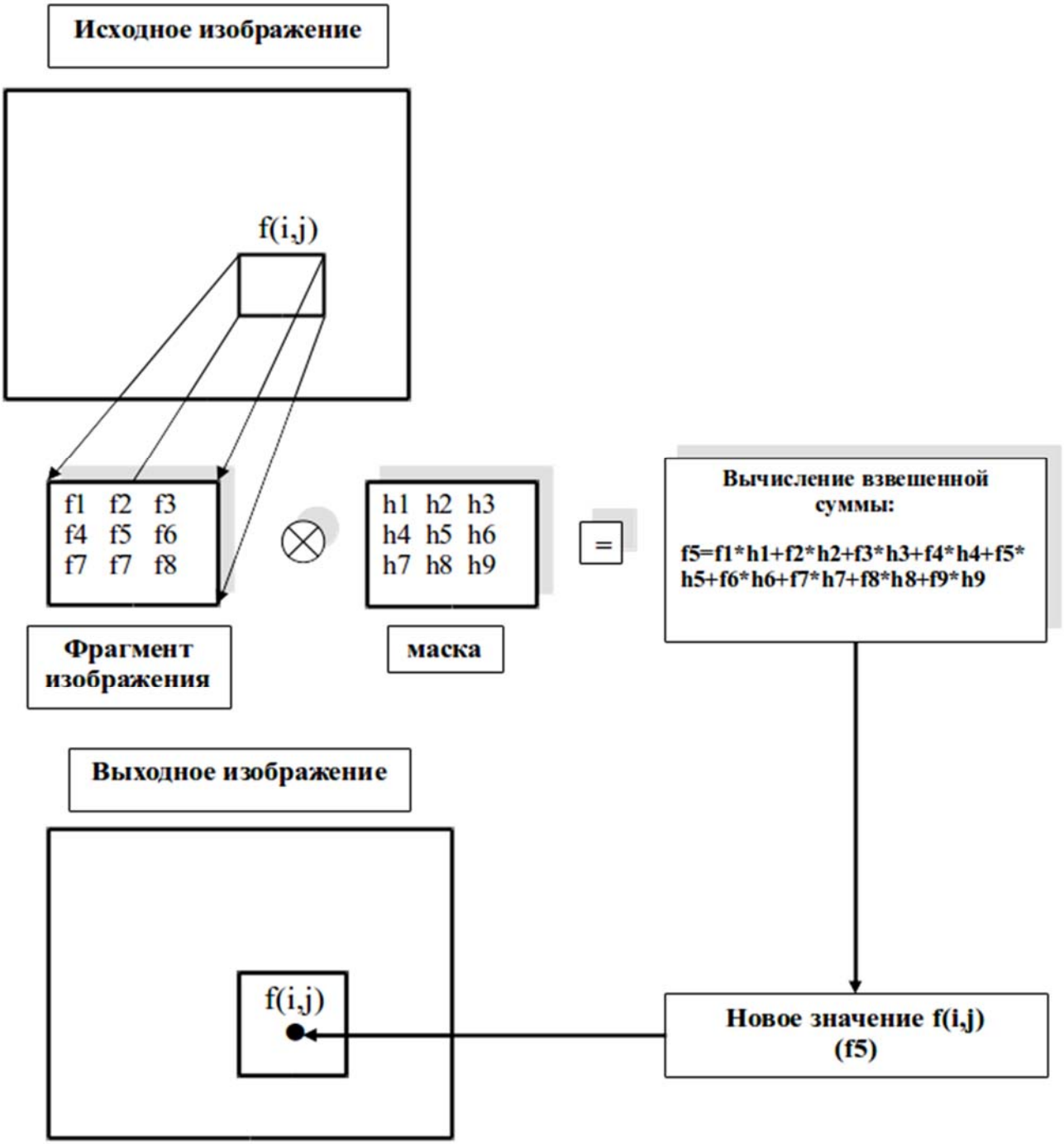


Рисунок 1.1 – Реализация операции свертки

Задание 1.1

Реализовать алгоритм, выполняющий операцию свертки исходного изображения с матрицей-маской. Размер и вид матрицы-маски задаются пользователем. Используйте следующие матрицы маски для реализации различных алгоритмов обработки изображений:

- ✓ для сглаживания и подавления шумов в изображении используют матрицу-маску размером 3x3 следующего вида:

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{1.1}$$

- ✓ для подчеркивания контуров используются матрицы-маски следующего вида:

$$\frac{1}{9}\begin{bmatrix} -1 & -1 & -1 \\ -1 & 17 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{1.2}$$

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{1.3}$$

- ✓ для выделения контуров используются маски следующего вида:

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{1.4}$$

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{1.5}$$

Пример выполнения задания 1.1

Ниже показан код для создания матриц свертки:

```
import numpy as np
kernel_blur = np.array([[1.,1.,1.],
                        [1.,1.,1.],
                        [1.,1.,1.]])/9.

kernel_sharpen1  = np.array([[-1.,-1.,-1.],
                             [-1.,17.,-1.],
                             [-1.,-1.,-1.]])/9.

kernel_sharpen2 = np.array([[0.,-1.,0.],
                            [-1.,5.,-1.],
                            [0.,-1.,0.]])

kernel_edge_detect1 = np.array([[-1.,-1.,-1.],
                           [-1.,8.,-1.],
                           [-1.,-1.,-1.]])

kernel_edge_detect2 = np.array([[0.,-1.,0.],
                            [-1.,4.,-1.],
                            [0.,-1.,0.]])
```

На рисунке 1.2 показано исходное изображение, загруженное с помощью matplotlib – пакета для работы с графикой в Python:

```
import matplotlib.image as imr
im = imr.imread('files1/test_image.jpg')
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams["figure.figsize"] = (15,10)
plt.rc('axes',facecolor=(1,1,1,0),edgecolor=(1,1,1,0));
plt.rc(('xtick','ytick'),color=(1,1,1,0));
plt.rcParams["image.cmap"] = "gray"
plt.imshow(im)
plt.show()
```

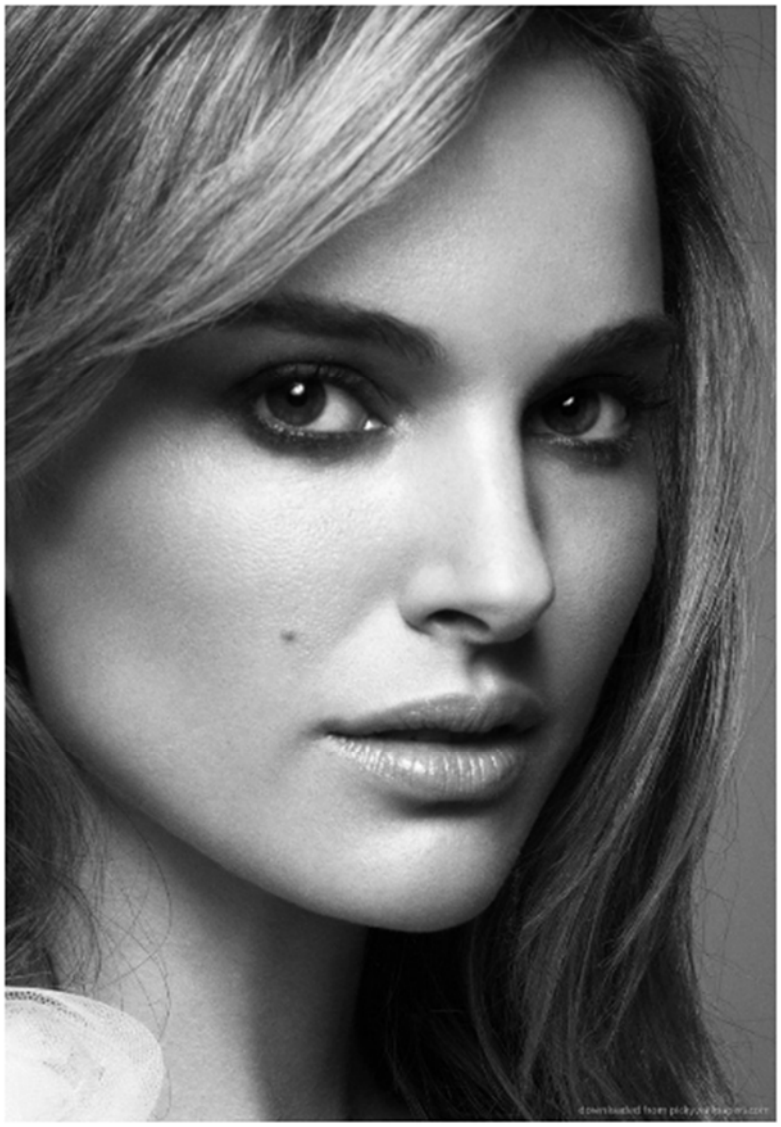


Рисунок 1.2 – Исходное изображение

Сглаживание и подавление шумов в изображение выполняется с помощью алгоритма, показанного на рисунке 1.1, с помощью маски (1.1).

```
def proc(image, kernel,const=0):
    kernel_sum = kernel.sum()
    image = image/255.
    const = const/255.
    # Получение размеров изображения и ядра для итерации по пикселам
и весам
    i_height, i_width = image.shape[0], image.shape[1]
    k_width, k_height = kernel.shape[0], kernel.shape[1]
```

```
    # создание пустого изображения.
    filtered = np.zeros_like(image)

    # Итерация по каждому (x, y) пикселу в изображении,
    for y in range(i_height):
        for x in range(i_width):
            weighted_pixel_sum = 0
            # итерация по каждому весу (kx, ky) в ядре, определенному
выше.
            # 'Центральный' вес в ядре интерпретируется как имеющий
координаты (0, 0);
            # тогда координаты остальных весов в ядре будут такими:
            #
            #  [ (-1,-1),  (0,-1),  (1,-1)
            #    (-1, 0),  (0, 0),  (1, 0)
            #    (-1, 1),  (0, 1),  (1, 1) ].
            #
            # Таким образом, пиксель изображения с координатами[y,x]
будет умножен на вес ядра[0,0]; аналогично,
            # пиксель[y-1,x] будет умножен на вес ядра[-1,0] и.т.д.
            # Значение отфильтрованного пикселя это сумма этих
произведений.
Итак
            #
            #   weighted_pixel_sum = image[y-1,x-1] * kernel[-1,-1] +
            #                        image[y-1,x  ] * kernel[-1, 0] +
            #                        image[y-1,x+1] * kernel[-1, 1] +
            #                        image[y,  x-1] * kernel[ 0, 1] +
            #                        image[y,  x  ] * kernel[ 0, 0] +
            #                        и.т.д.

            for ky in range(-k_height // 2+1, k_height // 2+1):
                for kx in range(-k_width // 2+1, k_width // 2+1):
                    pixel = 0
                    pixel_y = y - ky
                    pixel_x = x - kx
```

```
                    # Проверка: если пиксель выходит за край изображения,
то он равен нулю,
                    # а иначе он берется из изображения.
                    if (pixel_y >= 0) and (pixel_y < i_height) and
(pixel_x >= 0) and (pixel_x < i_width):
                        pixel = image[pixel_y,pixel_x]
                    # Текущая позиция в маске:
                    pos = (ky + k_height // 2, kx + k_width // 2)
                    # Получение веса ядра в текущей позиции:
                    weight = kernel[pos[1], pos[0]]
                    weighted_pixel_sum += pixel * weight
            weighted_pixel_sum+=const

            # наконец, пиксель с позицией (x, y) это сумма взвешенных
соседних пикселей:
            if weighted_pixel_sum > 1.:
                weighted_pixel_sum=1.
            elif weighted_pixel_sum < 0.:
                weighted_pixel_sum=0.
            #Нормализация.
            filtered[y, x] = weighted_pixel_sum

    return (filtered*255).astype('int')
plt.subplot(1,2,1)
plt.imshow(im)
plt.subplot(1,2,2)
plt.imshow(proc(im, kernel_blur))
plt.show()
```

В результате получается изображение, показанное на рисунке 1.3.

Рисунок 1.3 – Исходное и сглаженное изображение

Подчеркивание контуров выполняется с использованием масок (1.2), (1.3). В результате получаются изображения, показанные на рисунках 1.4 и 1.5:

```
im_out=proc(im, kernel_sharpen1)
plt.subplot(1,2,1)
plt.imshow(im)
plt.subplot(1,2,2)
plt.imshow(im_out)
plt.show()
```

Рисунок 1.4 – Подчеркивание контуров с помощью маски (1.2)

```
im_out=proc(im, kernel_sharpen2)
plt.figure(figsize=(15,10))
plt.subplot(1,2,1)
plt.imshow(im)
plt.subplot(1,2,2)
plt.imshow(im_out)
plt.show()
```

Рисунок 1.5 – Подчеркивание контуров с помощью маски (1.3)

Выделение контуров осуществляется с помощью масок (1.4), (1.5). В результате получаются изображения, показанные на рисунках 1.6 и 1.7:

```
im_out=proc(im, kernel_edge_detect1)
plt.subplot(1,2,1)
plt.imshow(im)
plt.subplot(1,2,2)
plt.imshow(im_out)
plt.show()
```

Рисунок 1.6 – Подчеркивание контуров с помощью маски (1.4)

```
im_out=proc(im, kernel_edge_detect2)
plt.subplot(1,2,1)
plt.imshow(im)
plt.subplot(1,2,2)
plt.imshow(im_out)
plt.show()
```

Рисунок 1.7 – Подчеркивание контуров с помощью маски (1.5)

Задание 1.2

Реализовать медианный фильтр, использующийся для подавления точечных и импульсных помех. Пиксел изображения и его соседи в рассматриваемой области выстраиваются в вариационный ряд (по возрастанию или убыванию значений пикселов) и отбирается центральное значение этого вариационного ряда как новое значение пиксела. Результатом усредненного фильтрования является то, что любой случайный шум, содержащийся в изображении, будет эффективно устранен (рисунок 1.7).

Это происходит потому, что любое случайное резкое изменение в интенсивности пиксела в пределах рассматриваемой области будет сортироваться, т. е. оно будет помещаться либо на вершину, либо на нижнюю часть сортированных значений этой области и не будет учитываться, так как для нового значения элементов всегда отбирается центральное значение.

Пример выполнения задания 1.2

Возможная реализация медианного фильтра показана ниже:

```
def median(image, size=(5,5)):
    # Получение размеров изображения и ядра для итерации по пикселам
и весам:
    i_height, i_width = image.shape[0], image.shape[1]
    k_width, k_height = size[0], size[1]
    # Создание пустого изображения:
    filtered = np.zeros_like(image)
    # Итерация по каждому (x, y) пикселу в изображении:
    for y in range(i_height):
        for x in range(i_width):
            weighted_pixel = []
            # Итерация по каждому значению в ядре, размеры которого
определенны выше:
            for ky in range(-(k_height / 2), k_height - 1):
                for kx in range(-(k_width / 2), k_width - 1):
                    pixel = 0
                    pixel_y = y - ky
                    pixel_x = x - kx
```

```
                    # проверка: если пиксель выходит за край изображения,
то он равен нулю,
                    # а иначе он берется из изображения
                    if (pixel_y >= 0) and (pixel_y < i_height) and
(pixel_x >= 0) and (pixel_x < i_width):
                        pixel = image[pixel_y,pixel_x]
                    # и добавляется во временный список.
                    weighted_pixel.append(pixel)
            # Наконец, находится медиана:
            filtered[y, x] = np.median(weighted_pixel)
    return filtered
plt.subplot(1,2,1)
plt.imshow(im)
plt.subplot(1,2,2)
plt.imshow(median(im))
plt.show()
```

В результате получается изображение, показанное на рисунке 1.8.

Рисунок 1.8 – Медианный фильтр

Задание 1.3

Реализовать алгоритм тиснения. Тиснение делается аналогично алгоритмам усреднения или подчеркивания контуров. Каждый пиксел в изображении обрабатывается ядром (матрицей-маской) тиснения размером 3 x 3:

$$\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$$

После того, как значение пиксела обработано ядром тиснения, к нему прибавляется 128. Таким образом, значением фоновых пикселов станет средний серый цвет (красный = 128, зеленый = 128, синий = 128). Суммы, превышающие 255, можно округлить до 255. В тисненом варианте изображения контуры кажутся выдавленными над поверхностью. Направление подсветки изображения можно изменять, меняя позиции 1 и –1 в ядре. Если, например, поменять местами значения 1 и –1, то реверсируется направление подсветки (рисунок 1.8):

```
kernel_stamping1= np.array([[0, -1,  0],
                            [1,  0, -1],
                            [0,  1,  0]])
kernel_stamping2= np.array([[0, 1,  0],
                            [-1,  0, 1],
                            [0,  -1,  0]])
im_out1=proc(im, kernel_stamping1, const=127)
im_out2=proc(im, kernel_stamping2, const=127)
plt.subplot(1,3,1)
plt.imshow(im)
plt.subplot(1,3,2)
plt.imshow(im_out1)
plt.subplot(1,3,3)
plt.imshow(im_out2)
plt.show()
```

В результате получается изображение, показанное на рисунке 1.9.

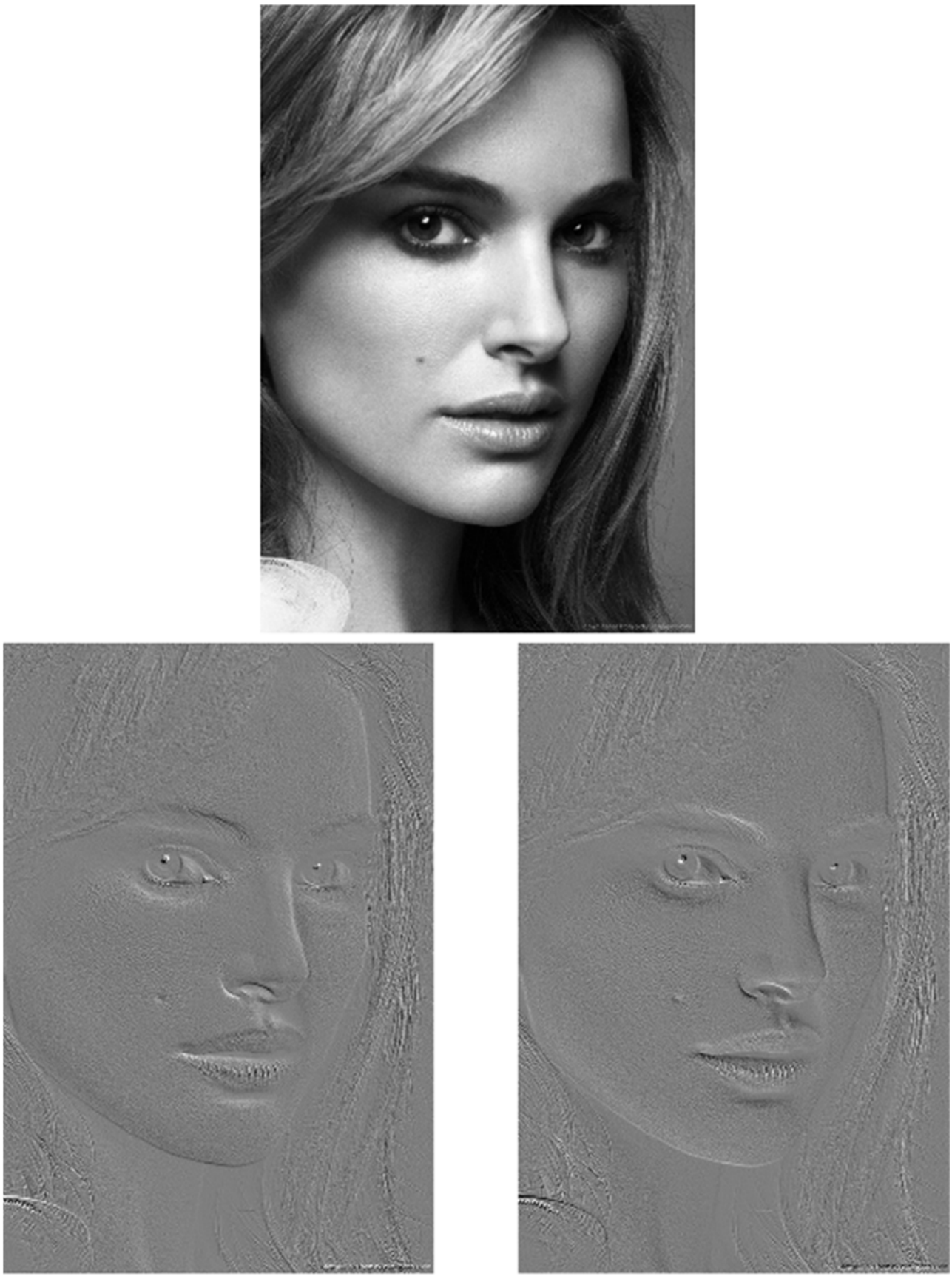


Рисунок 1.9 – Тиснение

Задание 1.4

Акварелизация изображения. Акварельный фильтр преобразует изображение, и после обработки оно выглядит так, как будто написано акварелью (рисунок 1.9).

Первый шаг в применении акварельного фильтра – сглаживание цветов в изображении. Одним из способов сглаживания является применение медианнного усреднения цвета в каждой точке. Значение цвета каждого пиксела и его 24 соседей (размер матрицы-маски равен 10 x 10) выстраиваются в вариационный ряд по убыванию или возрастанию. Медианное (тринадцатое) значение цвета в вариационном ряде присваивается центральному пикселу.

После сглаживания цветов необходимо применить фильтр подчеркивания контуров, чтобы выделить границы переходов цветов:

```
im_out=proc(median(im, size=(5,5)), kernel_sharpen1)
plt.subplot(1,2,1)
plt.imshow(im)
plt.subplot(1,2,2)
plt.imshow(im_out)
plt.show()
```

В результате получается изображение, показанное на рисунке 1.10.

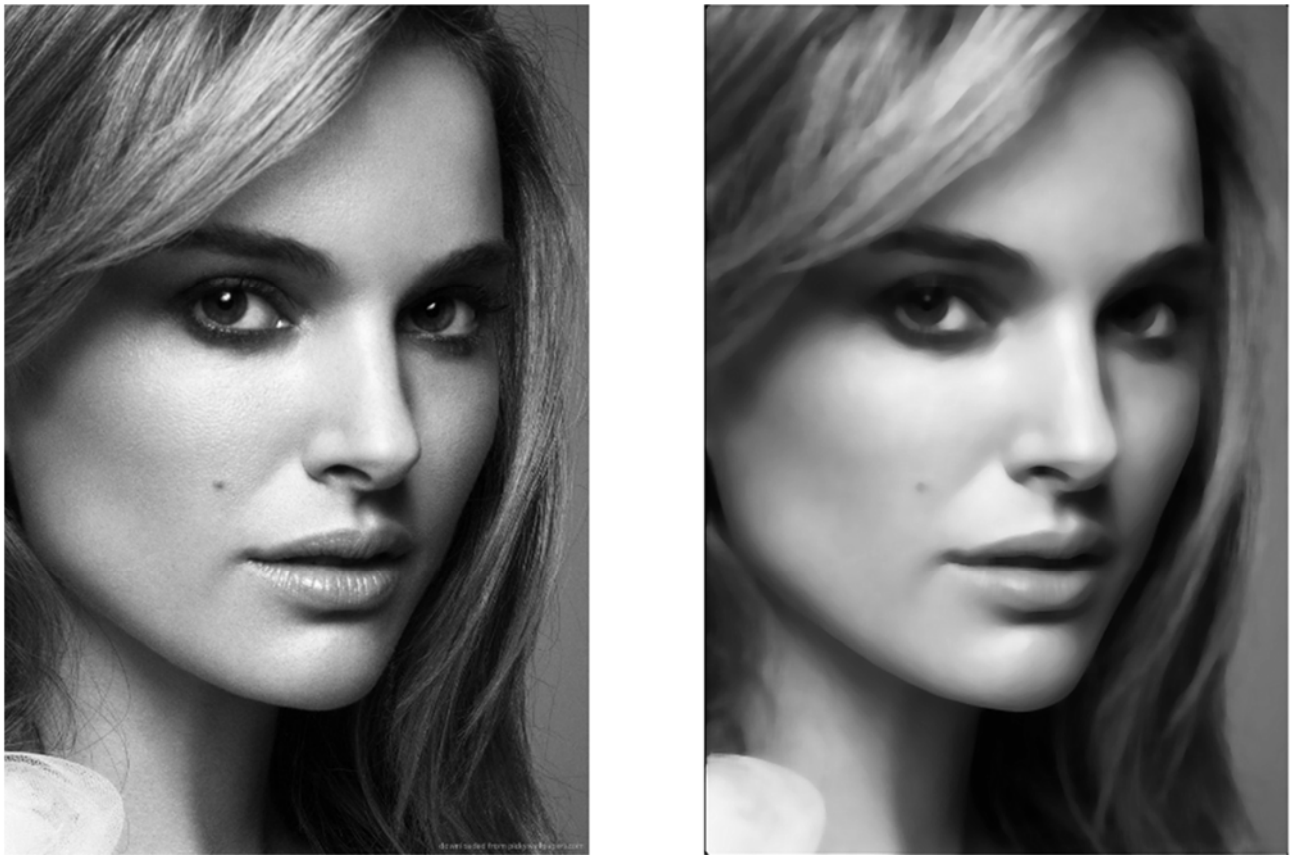


Рисунок 1.10 – Фильтр «Акварель»

Фильтр Гаусса. Фильтр размытия по Гауссу (широко известный как “gaussian blur”) достаточно часто применяется сам по себе или как часть других алгоритмов обработки изображений.

В этом случае матрица коэффициентов определяется по формуле нормального распределения Гаусса. Маска размером 5 x 5 с делителем 256:

$$\frac{1}{256}\begin{bmatrix} 1 & 4 & 6 & 4 & 1 \\ 4 & 16 & 24 & 16 & 4 \\ 6 & 24 & 36 & 24 & 6 \\ 4 & 16 & 24 & 16 & 4 \\ 1 & 4 & 6 & 4 & 1 \end{bmatrix}.$$

Результат применения такого фильтра показан на рисунке 1.10:

```
kernel_gaussian = np.array([[1, 4,  6,  4,  1],
                            [4, 16, 24, 16, 4],
                            [6, 24, 36, 24, 6],
                            [4, 16, 24, 16, 4],
                            [1, 4,  6,  4,  1]])/256.
```

```
im_out=proc(im, kernel_gaussian)
plt.subplot(1,2,1)
plt.imshow(im)
plt.subplot(1,2,2)
plt.imshow(im_out)
plt.show()
```

В результате получается изображение, показанное на рисунке 1.11.

Рисунок 1.11 – Фильтр Гаусса

Обработка цветных изображений. Цветные изображения обрабатываются по тому же самому алгоритму, что и черно-белые, который применятся к RGB каналам изображения:

```
im_clr = imr.imread('files1/test_image1.jpg')
im_out = np.zeros_like(im_clr)
im_out[:,:,0]=proc(im_clr[:,:,0], kernel_sharpen2)
im_out[:,:,1]=proc(im_clr[:,:,1], kernel_sharpen2)
im_out[:,:,2]=proc(im_clr[:,:,2], kernel_sharpen2)
plt.subplot(1,2,1)
plt.imshow(im_clr)
```

```
plt.subplot(1,2,2)
plt.imshow(im_out)
plt.show()
```

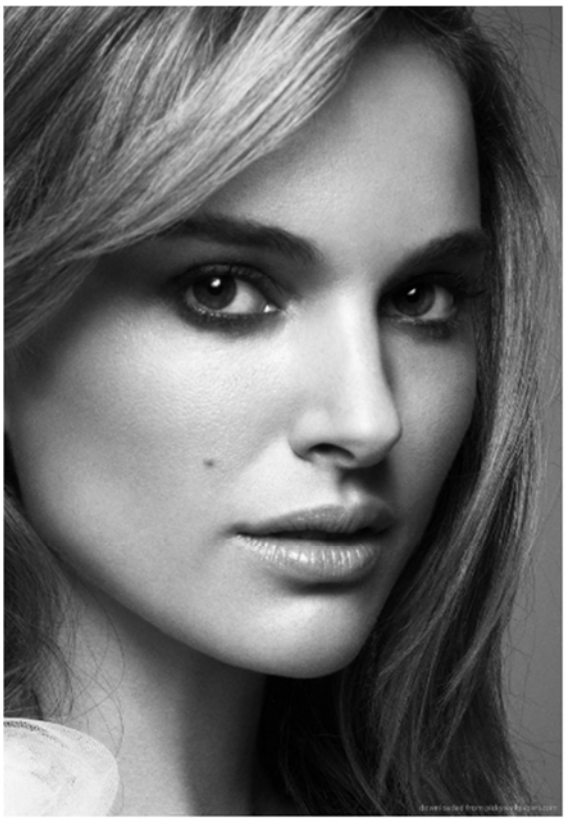
Результат обработки цветного изображения на примере подчеркивания контуров показан на рисунке 1.11.

Рисунок 1.12 – Обработка цветного изображения

Фильтры выделения и подчеркивания контуров произвольного размера. В общем случае маска свертки может быть произвольного размера, необязательно квадратная. На рисунках 1.12, 1.13 продемонстрированы результаты обработки изображения с прямоугольными масками выделения и подчеркивания контуров:

```
height=3
width=9
size=height*width
k=np.zeros((width, height))
for h in range(height):
    for w in range(width):
        if w == width//2 and h == height//2:
```

```
            v=size-1
        else:
            v=-1
        k[w,h]=v
print(k)
im_out=proc(im, k)
plt.subplot(1,2,1)
plt.imshow(im)
plt.subplot(1,2,2)
plt.imshow(im_out)
plt.show()
[[ -1.  -1.  -1.]
 [ -1.  -1.  -1.]
 [ -1.  -1.  -1.]
 [ -1.  -1.  -1.]
 [ -1.  26.  -1.]
 [ -1.  -1.  -1.]
 [ -1.  -1.  -1.]
 [ -1.  -1.  -1.]
 [ -1.  -1.  -1.]]
```



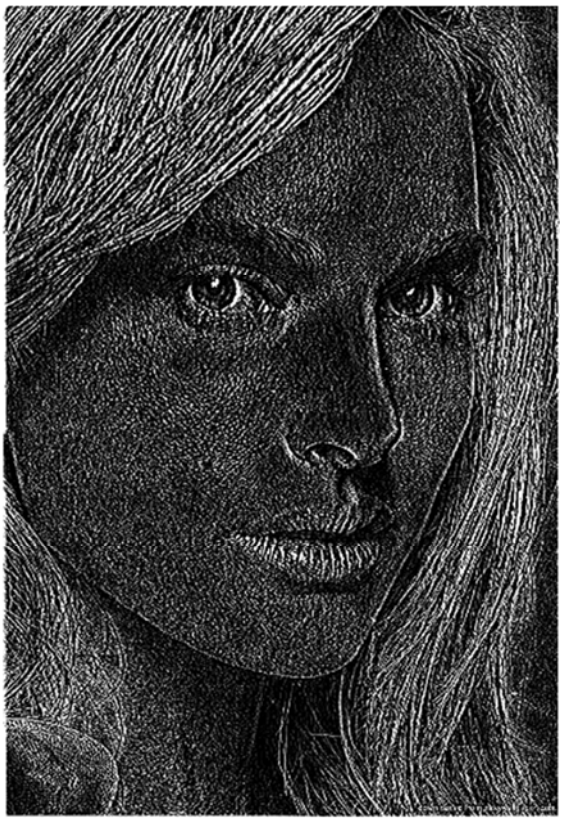


Рисунок 1.13 – Фильтр выделения контуров размером 3 x 9

```
height=9
width=3
size=height*width
k=np.zeros((width, height))
for h in range(height):
    for w in range(width):
        if w == width//2 and h == height//2:
            v=size
        else:
            v=-1
        k[w,h]=v
print(k)
im_out=proc(im, k)
plt.subplot(1,2,1)
plt.imshow(im)
plt.subplot(1,2,2)
plt.imshow(im_out)
plt.show()
[[ -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.]
 [ -1.  -1.  -1.  -1.  27.  -1.  -1.  -1.  -1.]
 [ -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.  -1.]]
```

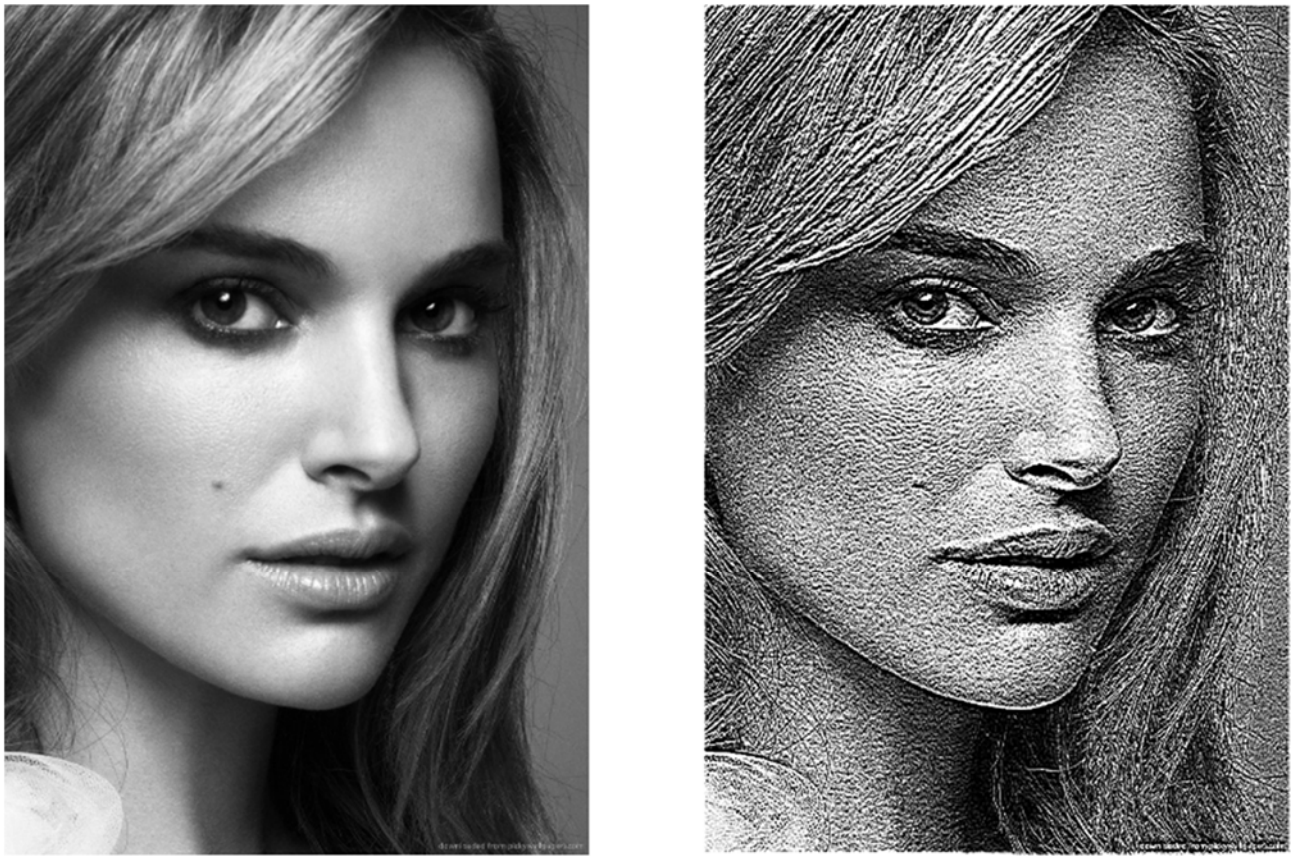


Рисунок 1.14 – Фильтр подчеркивания контуров размером 9 x 3

Задания для самостоятельной работы

1. Реализуйте фильтр тиснения произвольного размера.
2. Какой вид должна иметь матрица свертки, чтобы не изменять изображение? Приведите пример.

ПРАКТИЧЕСКАЯ РАБОТА № 2.
СЕГМЕНТАЦИЯ ИЗОБРАЖЕНИЙ

Сегментация изображений с помощью порогов

Задание 2.1

1. Построить гистограммы по трем (RGB) составляющим изображения. По оси абсцисс откладывается значение составляющей, а по оси ординат количество пикселов, имеющих соответствующее значение составляющей. При необходимости провести сглаживание гистограммы:

$$\hat{g}_i = \frac{1}{n}\sum_{-n/2}^{n/2} g_{i+j},$$

где $\hat{g}_i$ – новое значение столбца гистограммы;

n – ширина фильтра сглаживания.

1. По гистограмме определить значение порогов сегментации: значения порогов по каждой составляющей R, G и B определяются по локальным минимумам гистограмм; значения составляющих изображения, лежащие в пределах между двумя минимумами гистограммы, можно отнести к одной области (сегменту) изображения.
2. Однородные области изображения окрасить в псевдоцвета.

Пример выполнения задания

Загрузка исходного изображения (рисунок 2.1):

```
import matplotlib.image as imr
im = imr.imread('files2/test_image1.jpg')
import matplotlib.pyplot as plt
%matplotlib inline
plt.rcParams["figure.figsize"] = (10,7)
plt.imshow(im)
plt.show()
```

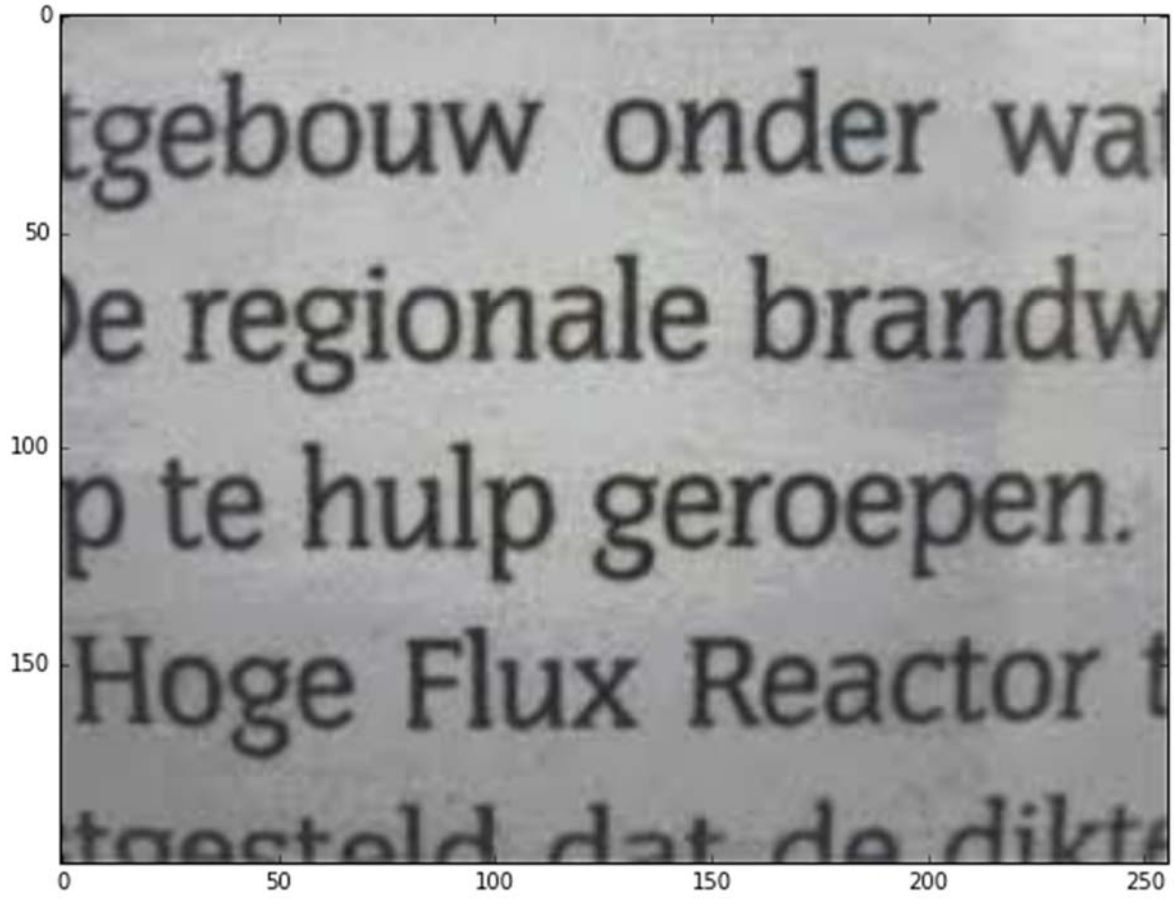


Рисунок 2.1 – Исходное изображение

Результаты построения гистограмм, согласно пункту 1 задания показаны на рисунке 2.2.

```
import numpy as np
def histogram(im, bins):
    res = np.zeros(bins)
    for i in im: res[i] = res[i] + 1
    return res

h1 = histogram(im[:,:,0], 256)
h2 = histogram(im[:,:,1], 256)
h3 = histogram(im[:,:,2], 256)
plt.subplot(131)
plt.scatter(np.arange(256), h1)
plt.subplot(132)
plt.scatter(np.arange(256), h2)
plt.subplot(133)
_=plt.scatter(np.arange(256), h3)
```

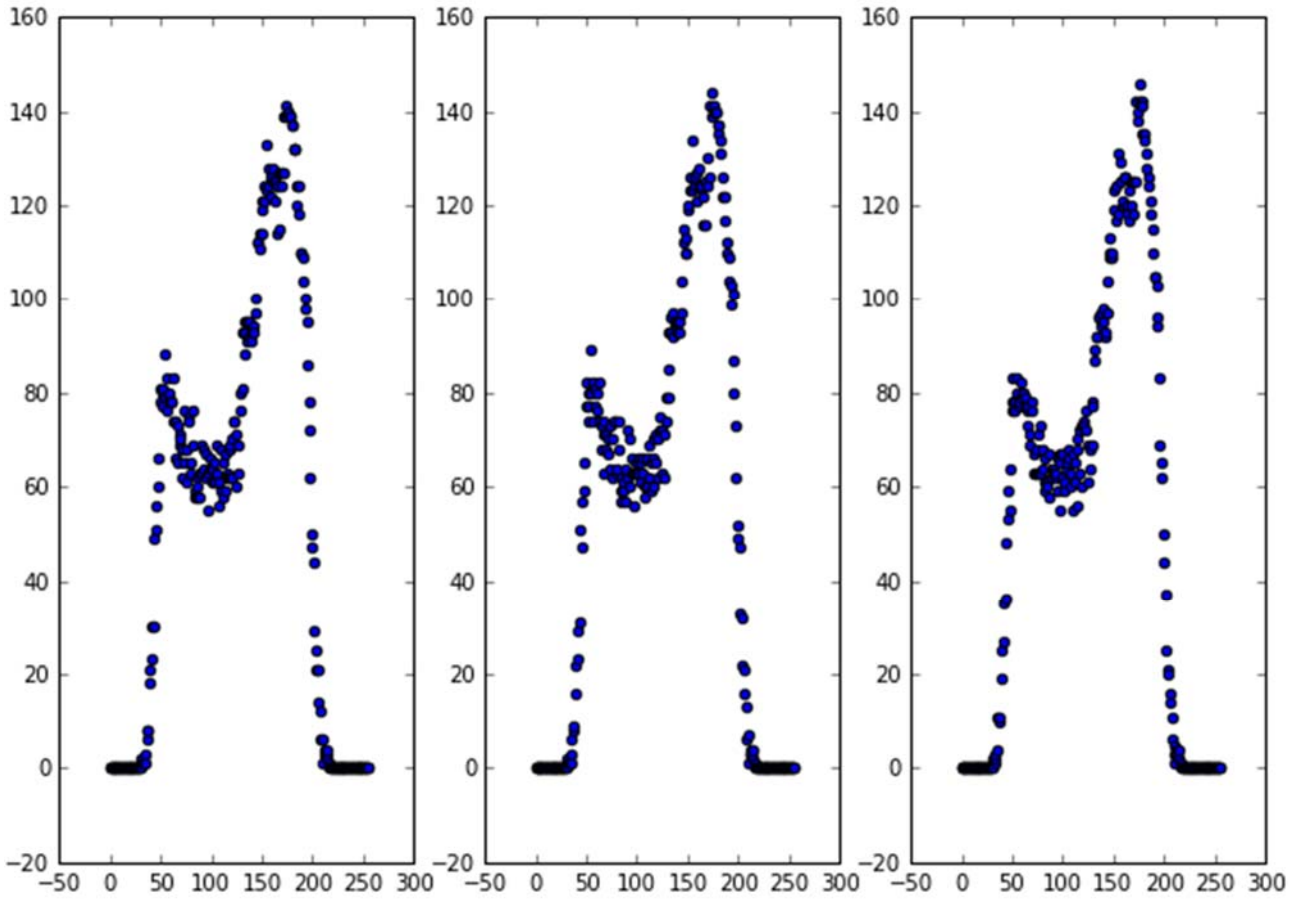


Рисунок 2.2 – Гистограммы изображения по каналам RGB

Сглаженные гистограммы показаны на рисунке 2.3. Локальные минимумы находятся исходя из выполнения условия $\hat{g}_{i-1} > \hat{g}_i \quad и \quad \hat{g}_i < \hat{g}_{i+1}$:

```
def smooth (g, n):
    res=np.zeros_like(g)
    for i in range(g.shape[0]):
        s=0
        for j in range(-n//2, n//2):
            inx = i+j
            if inx >= 0 and inx<g.shape[0]:
                s=s+g[inx]
        res[i]=s//n
    return res

n=40 # ширина окна сглаживания
_h1 = smooth(h1,n)
_h2 = smooth(h2,n)
_h3 = smooth(h3,n)
```

```
plt.subplot(131)
plt.scatter(np.arange(256),_h1)

plt.subplot(132)
plt.scatter(np.arange(256),_h2)
plt.subplot(133)
_ = plt.scatter(np.arange(256),_h3)

def locmin(g):
    res=[]
    for i in range(1, g.shape[0]-1):
        if g[i-1]>g[i] and g[i]<g[i+1]:
            res.append(i)
    res.append(255)
    return res
# поиск локальных минимумов
l1 = locmin(_h1)
l2 = locmin(_h2)
l3 = locmin(_h3)
```

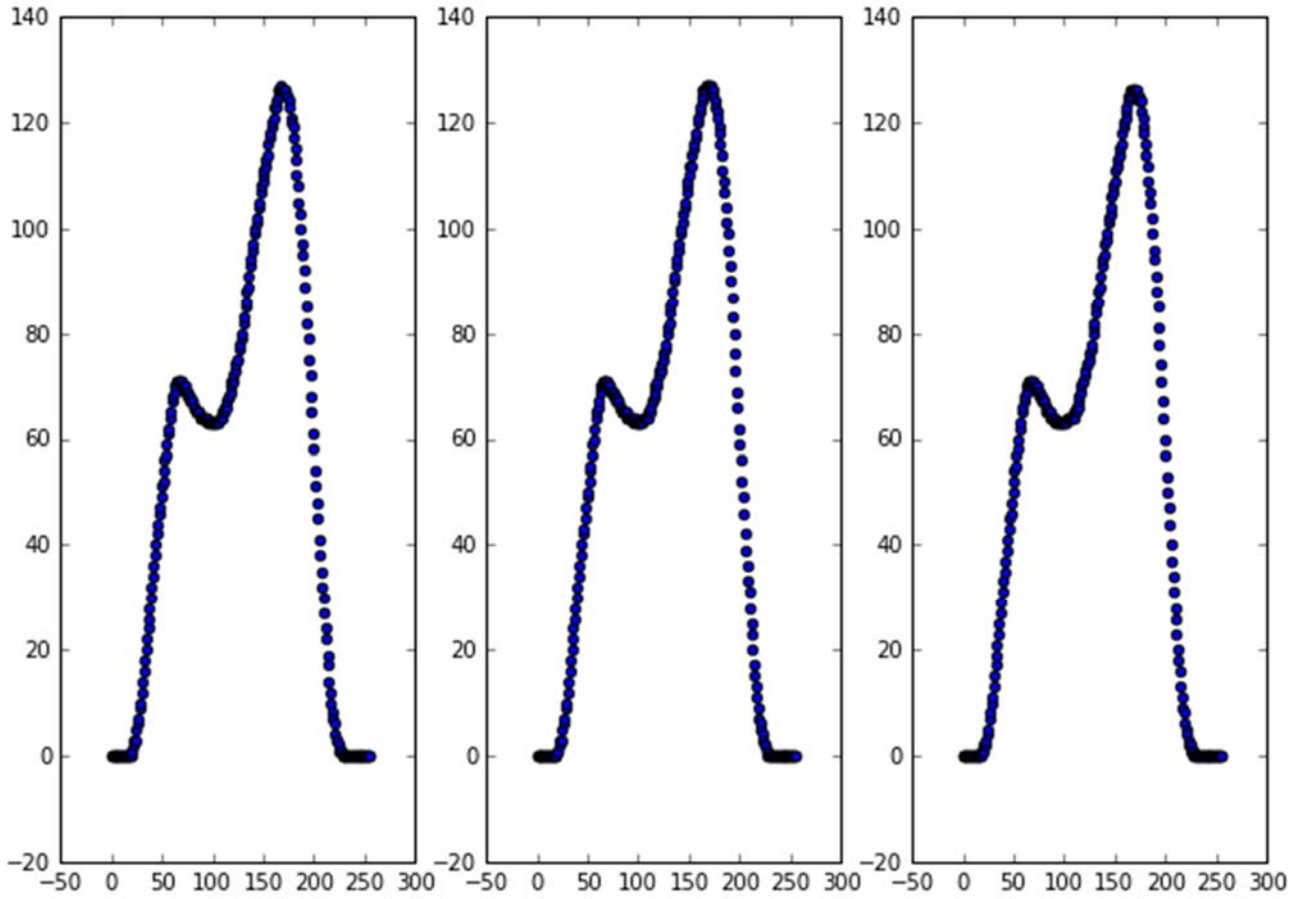


Рисунок 2.3 – Сглаженные гистограммы

Раскрашенное в псевдоцвета изображение показано на рисунке 2.4:

```
def colored(h, lm):
    for i in range(h.shape[0]):
        for j in range(h.shape[1]):
            for lc in lm:
                if h[i,j]<lc:
                    h[i,j]=lc
                    break
    return h
im.setflags(write=1)
im[:,:,0]=colored(im[:,:,0],l1)
im[:,:,1]=colored(im[:,:,1],l2)
im[:,:,2]=colored(im[:,:,2],l3)
plt.imshow(im)
plt.show()
```

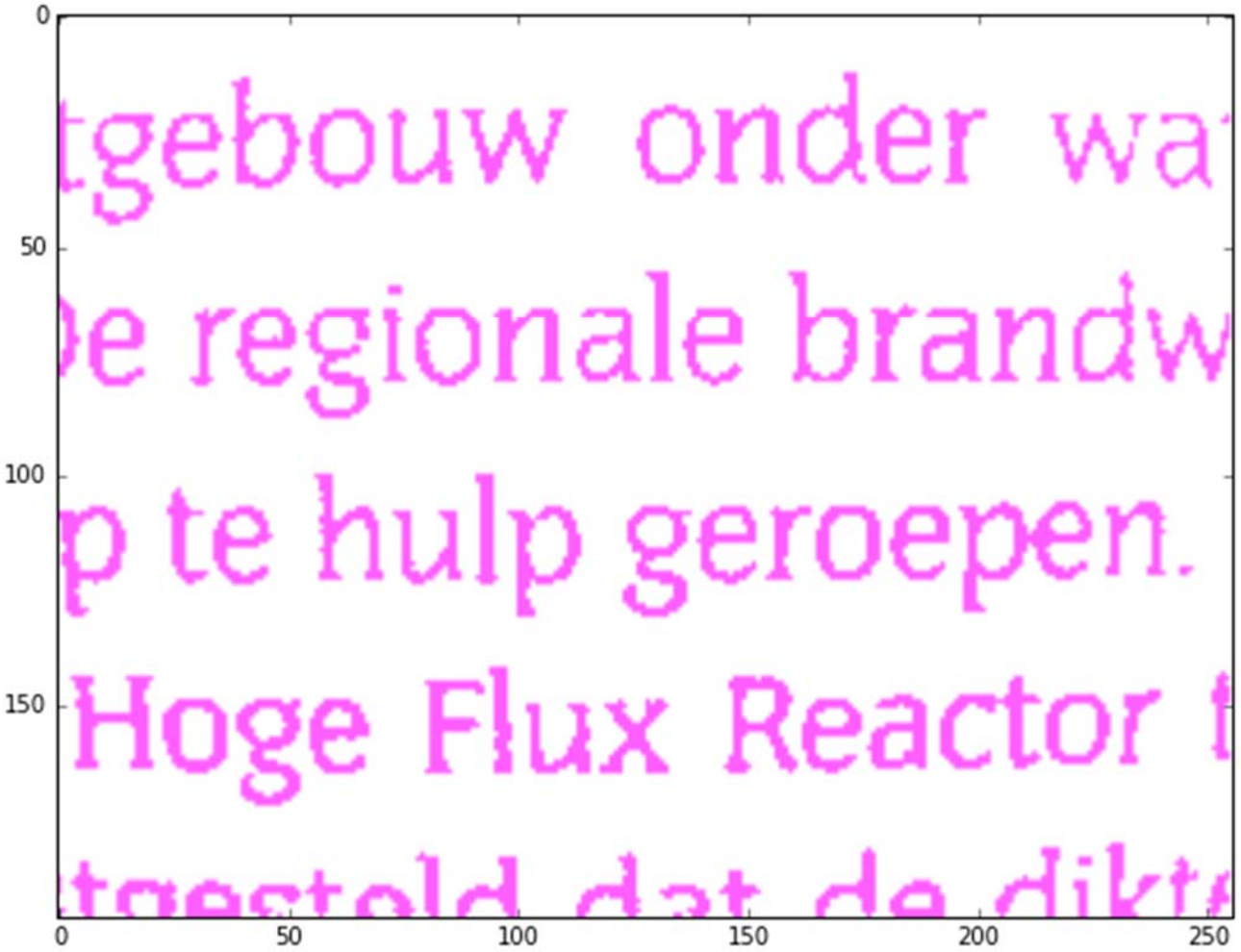


Рисунок 2.4 – Изображение, раскрашенное в псевдоцвета

Динамическая сегментация изображений

Задание 2.2

Реализовать метод сегментации изображения с динамически изменяющимся порогом, суть которого заключается в следующем:

1. В первую очередь необходимо определить приемлемый диапазон изменений P цвето-яркостных характеристик элементов изображения (пикселов), который будет служить критерием принадлежности пикселов одному сегменту.
2. Разбиение на сегменты начинается с того, что фиксируется определенный пиксел, который рассматривается как элемент сегмента.
3. Затем рассматриваются элементы, граничащие с сегментом (на первом проходе сегмент состоит из единственного фиксированного пиксела).
4. Вычисляется так называемая «базовая яркость» или «базовый цвет» по формуле:

$$I_i^b = \frac{1}{i}\sum_{j=1}^{i} I_j,$$

где I_j – яркость (цветовая характеристика) пиксела j.

На первом шаге – это значение яркости (цветовой характеристики) первого пиксела.

5. Вычисляется критерий принадлежности очередного граничного элемента к сегменту:

$$\left| I_i^b - I_i \right| \le P. \qquad (2.1)$$

Если неравенство выполняется, то рассматриваемый элемент изображения приписывается сегменту.

6. Пункты 3–5 выполняются до тех пор, пока существуют граничащие с сегментом элементы изображения, удовлетворяющие критерию (2.1).

Формирование сегмента завершается, если не окажется ни одного такого элемента.

7. Затем начинается формирование нового сегмента с фиксации пиксела, не принадлежащего ни одному из вновь образованных сегментов.
   Предыдущие шаги выполняются до тех пор, пока не будут рассмотрены все пикселы изображения.
8. Для цветного изображения может быть применен следующий критерий объединения пикселов в сегменты:

$$\sqrt{(R_c - R)^2 + (G_c - G)^2 + (B_c - B)^2} \le P,$$

где $R_cG_cB_c$ – базовый цвет сегмента и RGB – цвет тестируемого пиксела.

Пример выполнения задания

Ниже показан пример динамической сегментации изображения, реализованной с помощью рекурсии:

```
# Функция, возвращающая случайный цвет
def random_color():
    h,s,l = random.random(), 0.5 + random.random()/2.0, 0.4 +
random.random()/5.0
    r,g,b = [int(256*i) for i in colorsys.hls_to_rgb(h,l,s)]
    return [r, g, b]

# Метрика (пункт 8)
def metric(x1, x2):
    return np.sqrt(np.sum((x1-x2)**2))

# Рекурсивная функция добавления пикселя в сегмент:
# Pc - приемлемый диапазон, x - координаты добавляемого пикселя,
# seg - матрица сегментов, im - изображение,
# P - сумма яркостей пикселей в сегменте,
# n - количество пикселей в сегменте, n - номер сегмента.
def append(Pc, x, seg, im, P, n, i):
    # Добавляемый пиксель помечается принадлежащим текущему сегменту,
    seg[x[0],x[1]]=i
```

```
    n=n+1 # увеличивается количество пикселей в сегменте,
    P=P+im[x[0],x[1],:] #яркость добавленного пикселя добавляется к сумме
яркостей.
    for w in range(-1,2): # Циклы, проходящие
        for h in range(-1,2): # восемь соседей добавленного пикселя
            if w==0 and h==0: continue # Если сам пиксел не является
своим соседом, то находим
            cx=x[0]+w; cy=x[1]+h # координаты n-го соседа
            # и если они выходят за пределы изображения,
            # то переходим к следующему соседу.
            if cx<0 or cy<0 or cx>=im.shape[0] or cy>= im.shape[1]:
continue
            #Если n-й сосед уже входит в какой-либо сегмент,
            if seg[cx, cy] !=0: continue # то он пропускается.
            #Находим расстояние между базовой яркостью сегмента
            Pi=metric(P/n, im[cx, cy, :]) # и яркостью n-го соседа
            if Pi<=Pc: # и если оно меньше или равно приемлемому,
                try:
                    # то добавляем n-ный сосед в текущий сегмент
                    seg, P, n = append(Pc, [cx, cy], seg, im, P, n, i)
                except Exception:
                    print("Достигнут лимит вызова рекурсий!")
    # возвращается матрица сегментов, сумма яркостей пикселей в сегменте
    return(seg, P, n) # и количество пикселей в сегменте.

# Функция динамической сегментации
# im - изображение, Pc - приемлемый диапазон (пункт 1)
def din_seg(im, Pc):
    # Создается матрица сегментов
    seg = np.zeros([im.shape[0], im.shape[1]])
    si=1 # и счетчик сегментов.
    while True:
        #Ищется первый пиксель, не принадлежащий
        x=np.where(seg==0) # какому-либо сегменту.
        # Если такого нет
        if len(x[0])==0 or len(x[1]) == 0: break # то выходим из цикла.
        # Если есть, то добавляем его в текущий сегмент.
```

```
        seg, _, _ = append(Pc,[x[0][0],x[1][0]], seg, im,[0.,0.,0.], 0,
si)
        cl = random_color() # Случайный цвет
        for i in range (im.shape[0]): # которым мы раскрашиваем
            for j in range (im.shape[1]): # каждый пиксель изображения
                # который принадлежит
                if seg[i,j]==si: im[i,j]=cl # текущему сегменту.
        si=si+1
    # Возвращаем количество сегментов
    return si

import resource, sys
import random
import colorsys
im = imr.imread('files2/test_image1.jpg')
im.setflags(write=1)
# Запрашивается большой стек
resource.setrlimit(resource.RLIMIT_STACK, (2**29,-1))
# и большой лимит вызова рекурсий.
sys.setrecursionlimit(10**6)
# Динамическая сегментация
P = 90. # с приемлемым диапазоном равным 90.
seg_count = din_seg(im, P)
print ("Всего сегментов: "+str(seg_count))
plt.imshow(im)
plt.show()
```

Всего сегментов: 145

В результате получается изображение, показанное на рисунке 2.5.

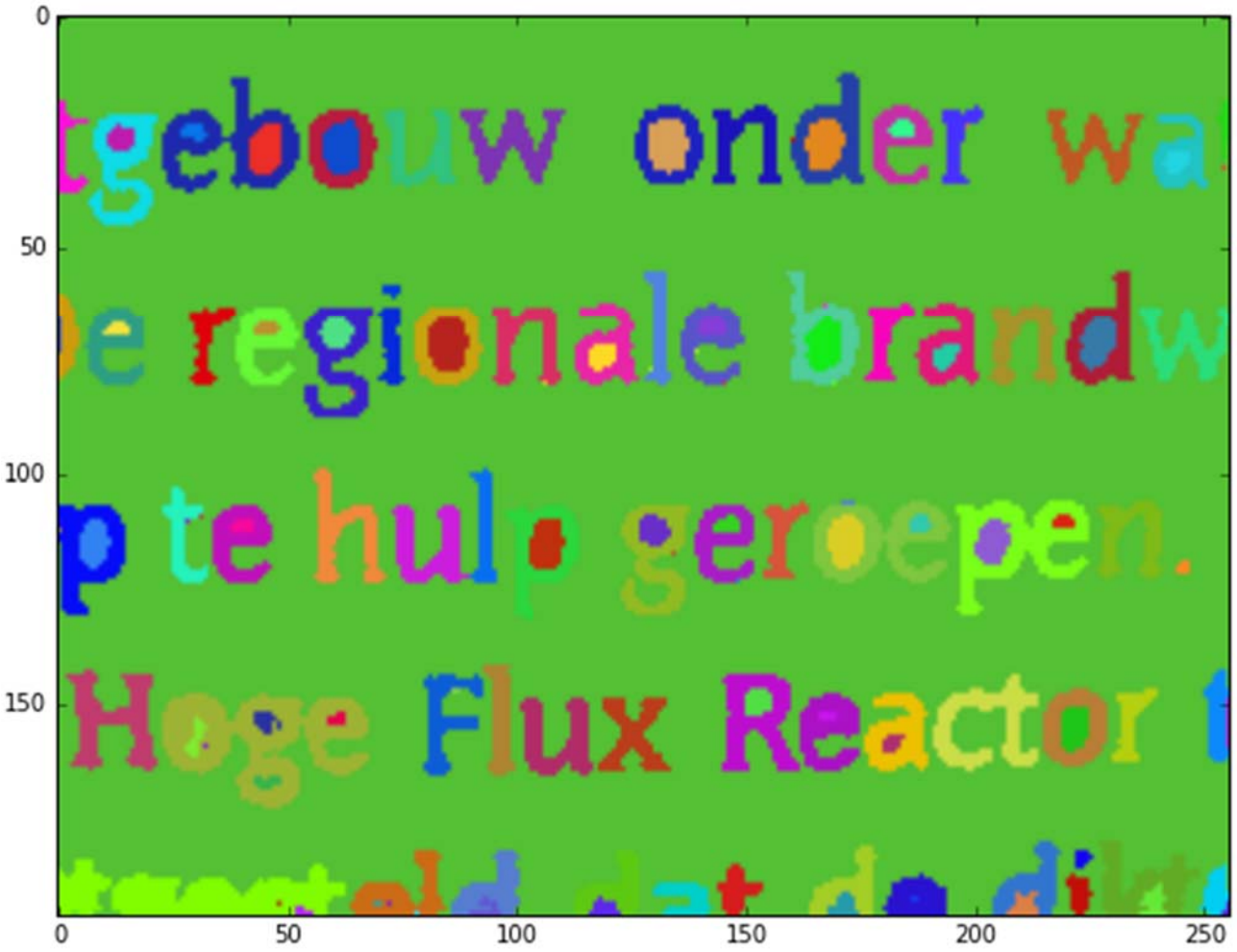


Рисунок 2.5 – Результат динамической сегментации изображения

Задания для самостоятельной работы

1. Реализуйте динамическую сегментацию без использования рекурсии, только с помощью циклов.
2. Реализуйте алгоритм, объединяющий смежные сегменты в один сегмент.
3. Реализуйте алгоритм, берущий в прямоугольную рамочку каждый сегмент.

ПРАКТИЧЕСКАЯ РАБОТА № 3.
ПРЕОБРАЗОВАНИЕ ФУРЬЕ

Дискретное преобразование Фурье

Дискретное преобразование Фурье (Discrete Fourier Transform) – это одно из преобразований Фурье, широко применяемых в алгоритмах цифровой обработки сигналов (его модификации применяются в сжатии звука в MP3, сжатии изображений в JPEG и др.), а также в других областях, связанных с анализом частот в дискретном (к примеру, оцифрованном аналоговом) сигнале. Дискретное преобразование Фурье требует в качестве входа дискретную функцию. Такие функции часто создаются путем дискретизации (выборки значений из непрерывных функций). Дискретные преобразования Фурье помогают решать дифференциальные уравнения в частных производных и выполнять такие операции, как свертки. Дискретные преобразования Фурье также активно используются в статистике, при анализе временных рядов.

Формулы преобразований

Прямое преобразование:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn}, k = 0, \ldots, N-1. \quad (3.1)$$

Обратное преобразование:

$$x_k = \frac{1}{N} \sum_{n=0}^{N-1} X_n e^{\frac{2\pi i}{N} kn}, k = 0, \ldots, N-1, \quad (3.2)$$

где N – количество значений дискретного сигнала;

x_n, $n = 0, \ldots, N-1$ – измеренные значения сигнала (в дискретных временных точках с номерами $n = 0, \ldots, N-1$, которые являются входными данными для прямого преобразования и выходными для обратного;

$X_k, k = 0, \ldots, N-1$ – N комплексных амплитуд синусоидальных сигналов, слагающих исходный сигнал; являются выходными данными для прямого

преобразования и входными для обратного; поскольку амплитуды комплексные, то по ним можно вычислить одновременно и модуль и фазу.

Тогда $|X_n|$ – обычная (вещественная) амплитуда *k*-го синусоидального сигнала;

$arg(X_n)$ – фаза *n*-го синусоидального сигнала (аргумент комплексного числа);

n – индекс частоты. Частота *n*-го сигнала равна n/T, *T* – период времени, в течение которого брались входные данные.

Из последнего видно, что преобразование раскладывает сигнал на синусоидальные составляющие (которые называются гармониками) с частотами от *N* колебаний за период до одного колебания за период, графическое изображение которых называется периодограммой. Поскольку частота дискретизации сама по себе равна *N* отсчетов за период, то высокочастотные составляющие не могут быть корректно отображены – возникает муаровый эффект. Это приводит к тому, что вторая половина из *N* комплексных амплитуд фактически является зеркальным отображением первой и не несет дополнительной информации.

Задание 3.1

Построить периодограммы модельного ряда, заданного формулой:

$$X(t) = \cos(2\pi t / 10 + 1) + \\ + \cos(2\pi t / 40 + \pi / 2), t = 0, 0.7, \ldots t < 480. \tag{3.3}$$

Пример выполнения задания

На рисунке 3.1 показан график и периодограммы указанного выше модельного ряда (3.3), полученные с помощью ДПФ:

```
def dft(x, inv=False):
    X=np.empty_like(x, dtype=np.complex128)
    N=len(x)
    i=1j
    for k in range(N):
        S=0+0.*i
        for n in range(N):
            omega=2*np.pi/N*k*n
            if not inv:
                omega=-omega
            S=S+(np.cos(omega)+np.sin(omega)*i)*x[n]
        if inv: S=S/N
        X[k]=S
    return X

import numpy as np
import matplotlib.pyplot as pl
%matplotlib inline
pl.rcParams["figure.figsize"] = (15,7)
omega=2*np.pi
dt=0.7
t=np.arange(0, 480, dt)
x=np.cos(t*omega/10+1)+np.cos(t*omega/40+np.pi/2)
N=len(x)
X= dft(x)
nu=np.arange(N)/dt/N
np.seterr(divide='ignore')
T=1/nu
```

```
fig = pl.figure()
ax = fig.add_subplot(3,1,1)
pl.plot(t,x)
pl.grid()
ax = fig.add_subplot(3,1,2)
A=np.sqrt(np.real(X)**2+np.imag(X)**2)/N
pl.semilogy(T[0:N//2], A[0:N//2])
pl.xlim([0, 100])
pl.xticks(np.arange(0,100,step=10) )
pl.grid()
P=np.arctan2(np.imag(X),np.real(X))
ax = fig.add_subplot(3,1,3)
pl.plot(T[0:N//2], P[0:N//2])
pl.xlim([0, 100])
pl.xticks(np.arange(0,100,step=10) )
pl.grid()
pl.show()
```

В результате получается изображение, показанное на рисунке 3.1.

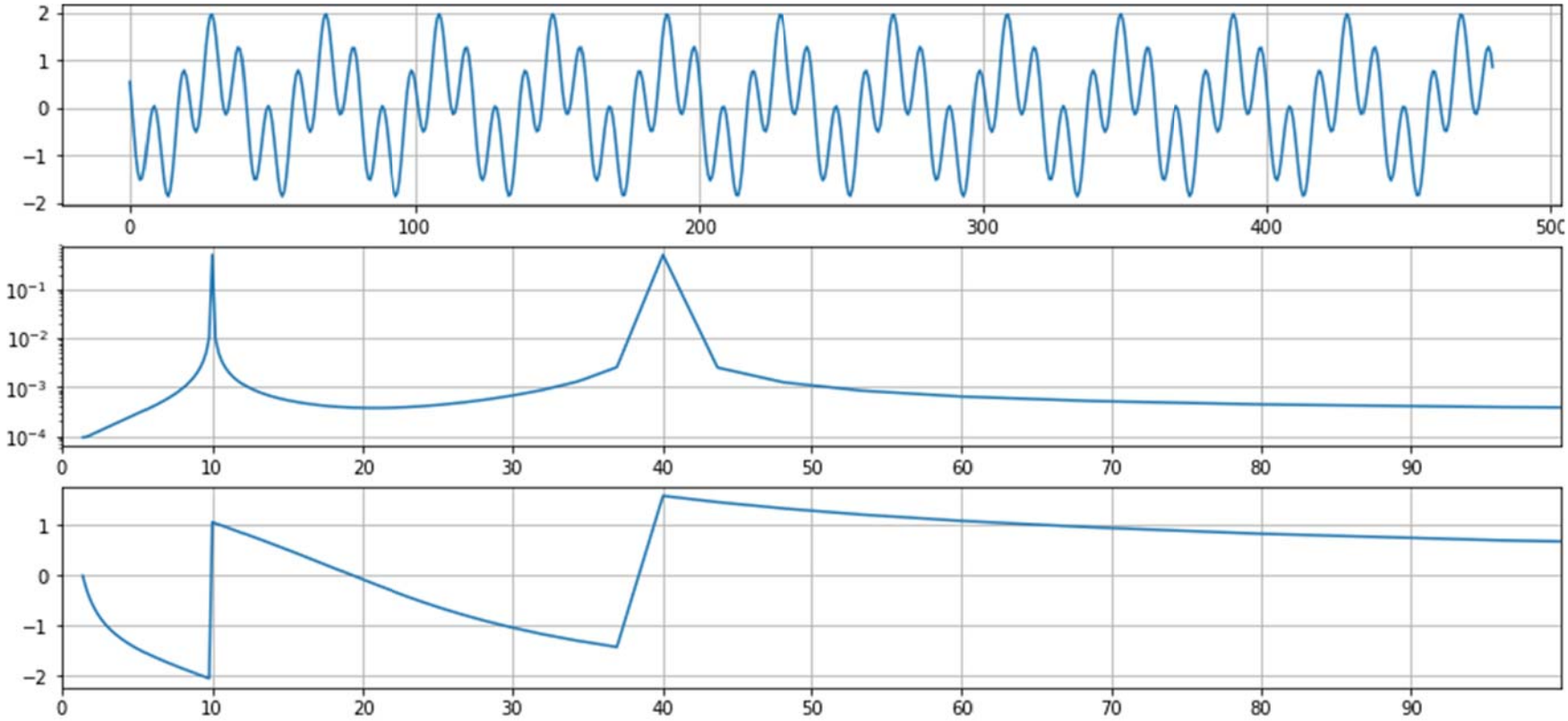


Рисунок 3.1 – Результат ДПФ модельного временного ряда

Обратное преобразование Фурье

Обратное ДПФ выполняется по формуле (3.2). Результат обратного ДПФ показан на рисунке 3.2:

```
X= dft(x, inv=True)
pl.rcParams["figure.figsize"] = (15,2)
_=pl.plot(t,x)
pl.grid()
```

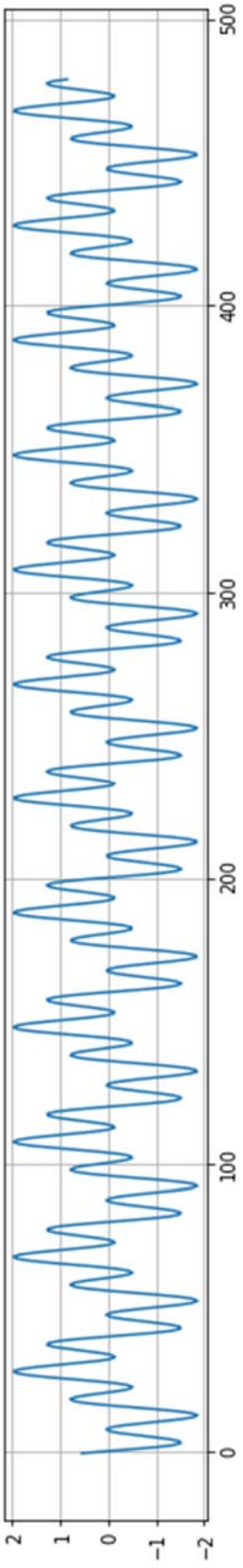


Рисунок 3.2 – Результат обратного ДПФ модельного временного ряда

Быстрое преобразование Фурье

Из соотношения (3.1) следует, что если последовательность x_k является комплексной, то при прямом вычислении требуется N^2 комплексных умножений и сложений. Основная идея БПФ состоит в том, чтобы исходную последовательность разбить на две более короткие последовательности, ДПФ которых могут быть скомбинированы таким образом, чтобы объединение их дало исходную N точечную ДПФ. Так, например, если N четное, то исходную последовательность можно разбить на две $N/2$ точечные последовательности, то для вычисления N точечную ДПФ потребуется $N^2/2$ комплексных умножений, т. е. в двое меньше, чем раньше. Эту операцию можно повторить, если $N/2$ является четным.

Алгоритм БПФ с прореживанием по времени

Считаем, что N равно степени 2-ки. Введем две последовательности $x_{1,n}$ и $x_{2,n}$, состоящие из четных и нечетных членов x_n:

$$x_{1,n} = x_{2n};\ x_{2,n} = x_{2n+1}, n = 0,\ \ldots,\ N\ /\ 2 - 1 \tag{3.4}$$

Для удобства можно ввести определение поворотного множителя:

$$W_N^{nk} = e^{\frac{\pm 2\pi kn}{N}}.$$

```
def W(k,N, n=1):
    return np.exp(-2*np.pi*1j*k*n/N)
```

Как видно из рисунка 3.3 $W^{nk}{}_N$ является периодичной функцией с периодом N., т. е.

$$W_N^{(n+mN)(k+1N)} = W_N^{kn}, l, m = 0, \pm 1,\ \ldots \tag{3.5}$$

```
pl.rcParams["figure.figsize"] = (15,4)
k = range(100)
w = [W(ki, 20) for ki in k]
pl.subplot(2,1,1)
pl.plot(k,np.real(w))
pl.subplot(2,1,2)
_=pl.plot(k,np.imag(w))
```

$$X_k = \sum_{n=0,чет.}^{N-1} x_n W_N^{nk} + \sum_{n=0,нечет.}^{N-1} x_n W_N^{nk} = \\ = \sum_{n=0}^{N/2-1} x_{2n} W_N^{2nk} + \sum_{n=0}^{2N-1} x_{2n+1} W_N^{(2n+1)k}. \quad (3.6)$$

Выражение (3.6) получилось из (3.1), где отделены слагаемые с четными номерами от слагаемых с нечетными номерами исходной последовательности.

Заметим, что $W_N^2 = W_{N/2}$, перепишем (3.6) с учетом (3.4) в виде:

$$X_k = \sum_{n=0}^{N/2-1} x_{1,n} W_{N/2}^{nk} + W_N^k \sum_{n=0}^{N/2-1} x_{2,n} W_{N/2}^{nk}. \quad (3.7)$$

Или

$$X_k = X_{1,k} + W_N^k X_{2,k}, \quad (3.8)$$

где $X_{1,k}$, $X_{2,k}$, равны $N/2$ точечному ДПФ последовательностей
$x_{1,n}$ и $x_{2,n}$ соответственно.

Из (3.7) следует, что N точечное ДПФ может быть разложено на два $N/2$ точечных ДПФ. Если бы $N/2$ точечное ДПФ вычислялось бы обычным образом, то потребовалось бы $N^2/2+N$ комплексных умножений.

При больших N (когда $N^2/2 >> N$) это позволяет сократить вычисления на 50 %:

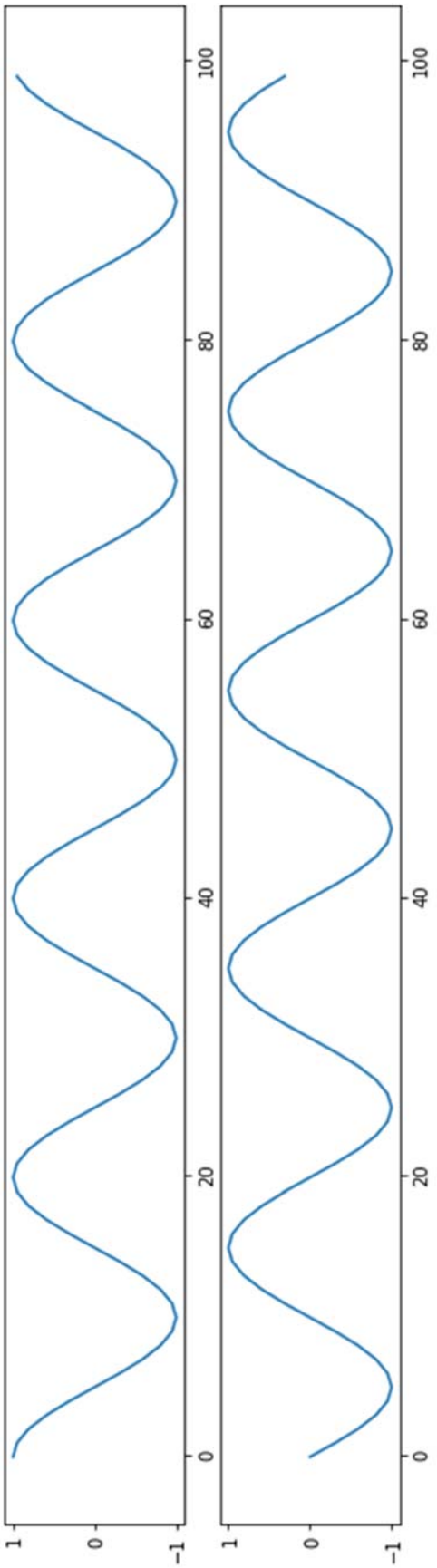


Рисунок 3.3 – Поворотный множитель

```
def dft2(x):
    N=len(x)
    if N % 2 > 0:
        raise ValueError("Количество отсчетов должно быть четным!")
    x1=x[::2]  #четные
    x2=x[1::2] #нечетные
    X1 = dft(x1)
    X2 = dft(x2)
    return [X1[k]+W(k,N)*X2[k]for k in range(N/2)]

import timeit
x2 = x[:500]
t1 = timeit.timeit('dft(x2)',
                   setup="from __main__ import dft, x2",
                   number=1)
print(t1)
t2 = timeit.timeit('dft2(x2)',
                   setup="from __main__ import dft2, x2",
                   number=1)
print(t2)
1.21888303757
0.611738204956
```

Как видно из примера выше, временные затраты сократились в два раза. Можно сравнить результаты двух преобразований, чтобы убедится в том, что они идентичны (рисунок 3.4):

```
X=dft2(x)
A=np.abs(X)
P=np.angle(X)
N=len(x)
nu=np.arange(N)/dt/N
T=1/nu
pl.subplot(2,1,1)
pl.semilogy(T[0:N//2], A[0:N//2])
pl.xlim([0, 100])
pl.xticks(np.arange(0,100,step=10))
pl.grid()
pl.subplot(2,1,2)
pl.plot(T[0:N//2], P[0:N//2])
pl.xlim([0, 100])
pl.xticks(np.arange(0,100,step=10))
pl.grid()
```

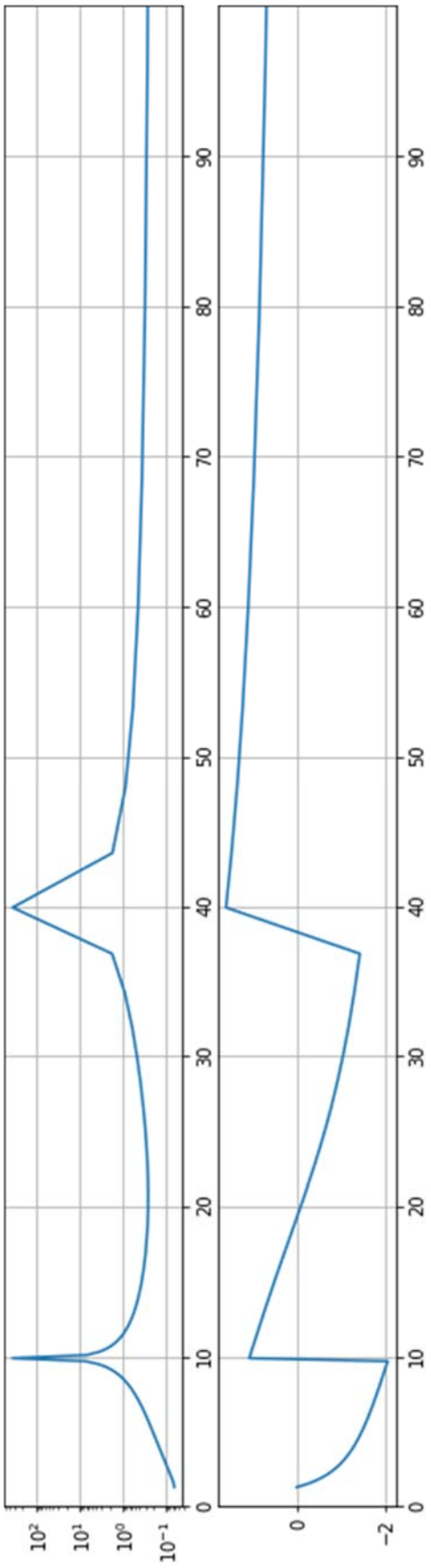


Рисунок 3.4 – Периодограммы модельного временного ряда, полученные с помощью ускоренного алгоритма

Поскольку *Xk* определено при $0 \le k < N$, а $X_{1,k}$, $X2,k$ при $0 \le k < N/2$, необходимо доопределить (3.8) при $k > N/2$.

Сделаем это следующим образом, используя периодичность ДПФ и тот факт, что $W_N^{k+N/2} = e^{-\frac{j2\pi(k+N/2)}{N}} = e^{-\frac{j2\pi k}{N}} e^{-j\pi} = -W_N^k$.

$$X_k = \begin{cases} X_{1,k} + W_N^k X_{2,k}, 0 \le k \le N/2 - 1 \\ X_{1,k-N/2} + W_N^{k-N/2} X_{2,k-N/2}, N/2 \le k \le N \end{cases}. \quad (3.9)$$

Выражение (3.9) описывает получение *N*-точечного Фурье преобразования из двух *N/2* точечных Фурье. Видно, что для получения каждого коэффициента Фурье *X(k)* необходимо выполнить одно умножение и одно сложение (или вычитание).

Здесь необходимо пояснить, почему выполняется только *N*/2 умножений на каждом шаге, а не *N* умножений? Необходимо рассмотреть выражение (3.9). Запишем вычисление X_0 и $X_{N/2}$:

$$\begin{aligned} X_0 &= X_{1,0} = W_N^0 X_{2,0} \\ X_{N/2} &= X_{1,N/2-N/2} - W_N^{N/2-N/2} X_{2,N/2-N/2} = X_{1,0} - W_N^0 X_{2,0}. \end{aligned} \quad (3.10)$$

Как видно из (3.10) X_0 и $X_{N/2}$ вычисляются почти одинаково, за исключением того, что в X_0 надо сложить, а в $X_{N/2}$ отнять одно и тоже произведение. Это произведение можно посчитать один раз, а затем добавить и отнять *к* и от $X_{1,0}$ для получения соответственно X_0 и $X_{N/2}$. Таким образом, чтобы вычислить два коэффициента Фурье, необходимо вычислить только одно произведение.

Если мы продолжим разбиение последовательности на две последовательности и применим тот же механизм, то мы еще в два раза уменьшим количество умножений. Таким образом, на каждом шаге мы выполняем *N*/2 умножений, а таких шагов log_2N. Число умножений равно $N log_2 N/2$, что значительно меньше N^2 (при $N = 1024$ число умножений меньше в 100 раз). Рассмотрим алгоритм на примере 8- точечной последовательности, т. е. $N = 8$. Выражение (3.9) можно иллюстрировать схемой на рисунке 3.5.

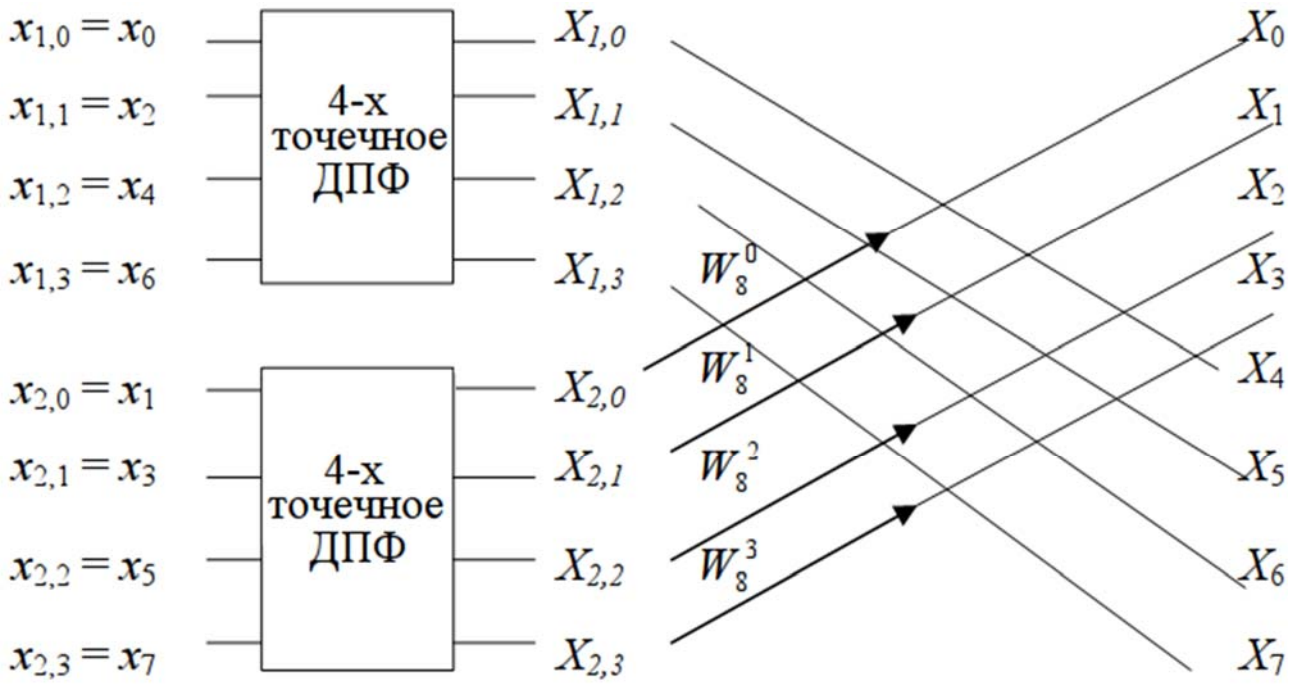


Рисунок 3.5 – Алгоритм Кули-Тюки для вычисления БПФ
(здесь стрелка вверх означает сложение; линия, направленная вниз – вычитание)

Описанный алгоритм известен как простой алгоритм Кули-Тьюки для степеней двойки.

Задание 3.2

Построить с помощью БПФ периодограммы модельного ряда, заданного формулой (3.3).

Пример выполнения задания

Ниже приведена реализация схемы вычисления БПФ, показанной на рисунке 3.5:

```
def fft(x):
    N=len(x)
    if N % 2 > 0:
        raise ValueError("Количество отсчетов должно быть степенью 2-ки")
    Y1=[]
    Y2=[]
    if N == 2: #условие окончания рекурсии
        return [x[0]+x[1], x[0]-x[1]]
    else:
        x1=x[::2]  #четные
        x2=x[1::2] #нечетные
        X1 = fft(x1)
        X2 = fft(x2)
    for k in range(N/2):
        tmp=W(k,N)*X2[k]
        Y1.append(X1[k]+ tmp)
        Y2.append(X1[k]- tmp)
    return Y1+Y2 #объединение двух списков


xp2 = x[:512]
t1 = timeit.timeit('dft(xp2)',
                   setup="from __main__ import dft, xp2",
                   number=1)
print(t1)
t2 = timeit.timeit('fft(xp2)',
                   setup="from __main__ import fft, xp2",
                   number=1)
print(t2)
1.3117249012
0.00572085380554
```

При $N = 2$ выполняется операция, которая носит название «Бабочка»:

$$X_0 = x_0 + x_1,$$
$$X_1 = x_0 - x_1,$$

т. к. при $k = 0$, $W^0_N = 1$. Можно снова сравнить на рисунках 3.6, 3.7 результаты двух преобразований, чтобы убедится в том, что они идентичны:

```
pl.rcParams["figure.figsize"] = (15,4)
X=dft(xp2)
A=np.abs(X)
P=np.angle(X)
N=len(xp2)
nu=np.arange(N)/dt/N
T=1/nu
pl.subplot(2,1,1)
pl.semilogy(T[0:N//2], A[0:N//2])
pl.xlim([0, 100])
pl.xticks(np.arange(0,100,step=10))
pl.grid()
pl.subplot(2,1,2)
pl.plot(T[0:N//2], P[0:N//2])
pl.xlim([0, 100])
pl.xticks(np.arange(0,100,step=10) )
pl.grid()
```

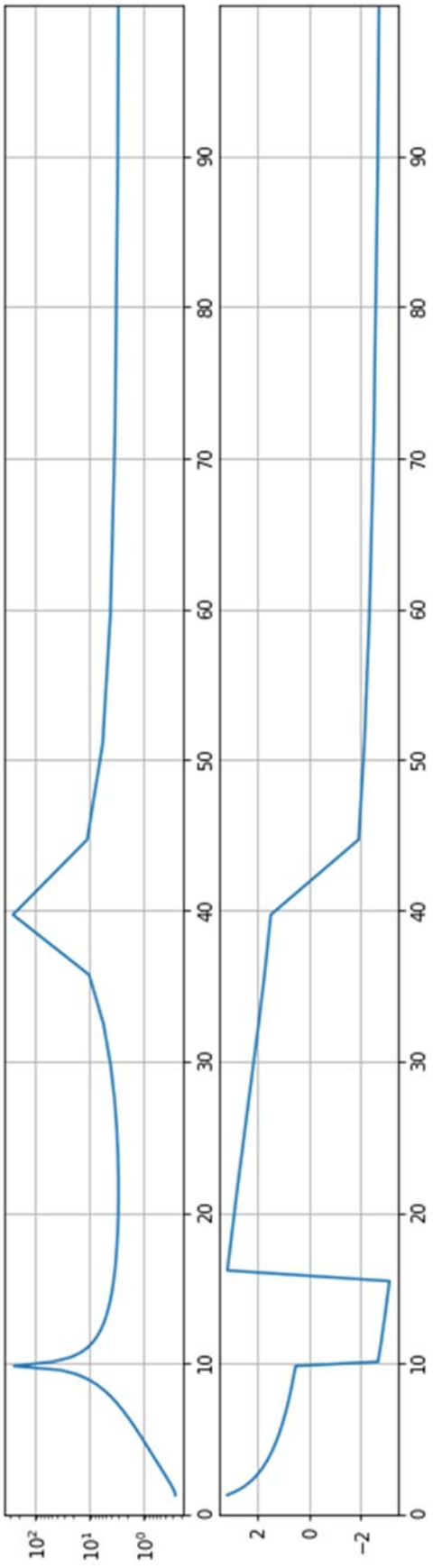


Рисунок 3.6 – Периодограммы модельного временного ряда, полученные с помощью ДПФ

```
pl.rcParams["figure.figsize"] = (15,4)
X=fft(xp2)
A=np.abs(X)
P=np.angle(X)
N=len(xp2)
nu=np.arange(N)/dt/N
T=1/nu
pl.subplot(2,1,1)
pl.semilogy(T[0:N//2], A[0:N//2])
pl.xlim([0, 100])
pl.xticks(np.arange(0,100,step=10))
pl.grid()
pl.subplot(2,1,2)
pl.plot(T[0:N//2], P[0:N//2])
pl.xlim([0, 100])
pl.xticks(np.arange(0,100,step=10) )
pl.grid()
```

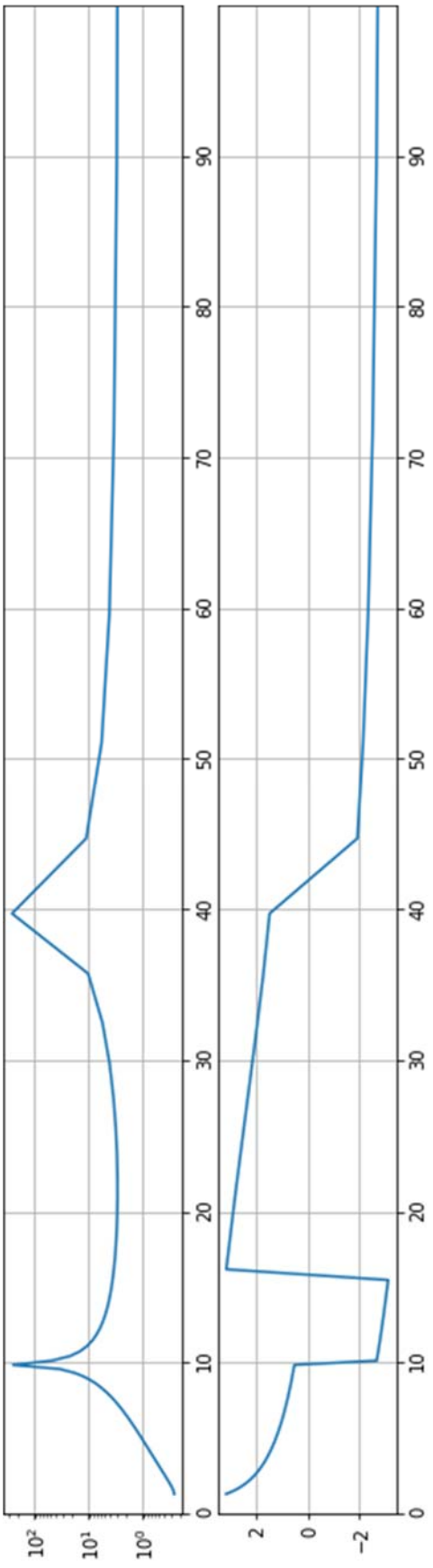


Рисунок 3.7 – Периодограммы модельного временного ряда, полученные с помощью БПФ

Задания для самостоятельной работы

1. Выполните исследование зависимости времени выполнения от количества отсчетов для алгоритмов ДПФ и БПФ и постройте графики.
2. Реализуйте алгоритм БПФ без использования рекурсии.
3. Реализуйте обратное БПФ.

ПРАКТИЧЕСКАЯ РАБОТА № 4.
ВЕЙВЛЕТ-ПРЕОБРАЗОВАНИЕ

Обычно для построения оценок спектра мощности временных рядов используют преобразование Фурье. Оно обладает способностью фокусировать в точку «размазанную» по времени информацию о периодичности функции при переходе из временной области в частотную. Достигается это за счет того, что ядро преобразования Фурье не локализовано во времени, но имеет предельную локализацию в частотной области. Это обстоятельство и делает преобразование Фурье хорошим инструментом для изучения процессов, характеристики которых не меняются со временем. Напротив, вейвлет-анализ основан на использовании локализованных во времени ядер преобразования, размеры которых согласованы с масштабом изучаемых компонентов ряда. Основная идея вейвлет-преобразования отвечает специфике многих временных рядов, демонстрирующих эволюцию во времени своих основных характеристик – среднего значения, дисперсии, периодов, амплитуд и фаз гармонических компонентов.

Непрерывное вейвлет-преобразование

Предположим, что имеется некоторый временной ряд x_n, заданный с равным шагом по времени $\delta t = 1$, где $n = 0 \ldots N - 1$, т. е. $N = 1024$ (рисунок 4.1).

$$X(t) = \begin{cases} \sin(2\pi t / 150), 0 \le t < 512, 768 \le t < 1024 \\ \sin(2\pi t / 150) + \sin(2\pi (t - 512) / 50), 512 \le t < 768 \end{cases}, \quad (4.1)$$

где $t = 0, 1, 2, \ldots, 1023$:

```
import numpy as np
import pylab as pl
%matplotlib inline
pl.rcParams["figure.figsize"] = (15,4)
# Тестовый временной ряд
Ns=1024
Nlo=0
Nhi=Ns
dt =1.
#Синусоиды с периодами 50 и 150
t=np.arange(0.0,dt*Ns,dt)
A=np.sin(2.0*np.pi*t/150.)
B=np.sin(2.0*np.pi*t/50.)
A[512:768]+=B[0:256]
x=A[:]
pl.plot(t,x)
_=pl.xlim([0, Ns])
```

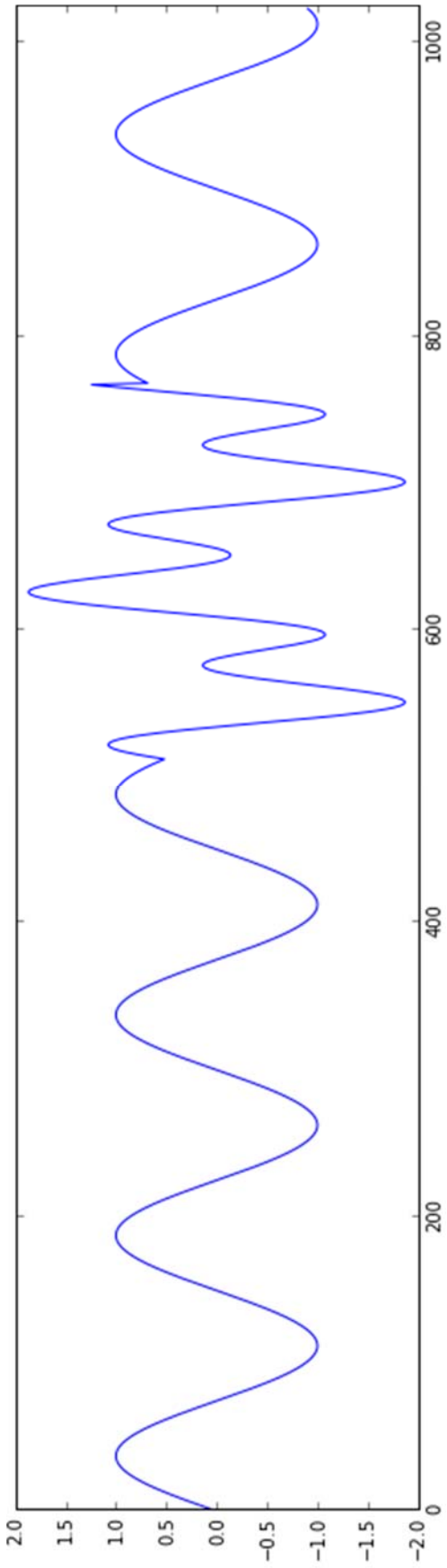


Рисунок 4.1 – Модельный временной ряд

Можно задать некоторую «волновую» вейвлет-функцию $\psi_0(\eta)$, хорошо локализованную во временном и частотном пространстве, зависящую от безразмерного параметра времени η. Например, это может быть вейвлет Морле, показанный на рисунке 4.2 и представляющий собой косинусоиду и синусоиду, модулированные Гауссианной:

$$\psi_0(\eta) = \pi^{-1/4} e^{i\omega_0 \eta} e^{-\eta^2/2},$$

где ω_0 – безразмерная частота вейвлета, часто используется значение $\omega_0 = 6$:

```
pl.rcParams["figure.figsize"] = (15,7)
def morlet(x, w0=6.0):
    return np.pi**-0.25 * np.exp(1j*w0*x)*np.exp(-0.5*x**2)

m_x = np.linspace(-2 * np.pi, 2 * np.pi, 1024)
m = morlet(m_x)
pl.plot(m_x,np.real(m), label="Re")
pl.plot(m_x,np.imag(m), label="Im")
_ = pl.legend()
```

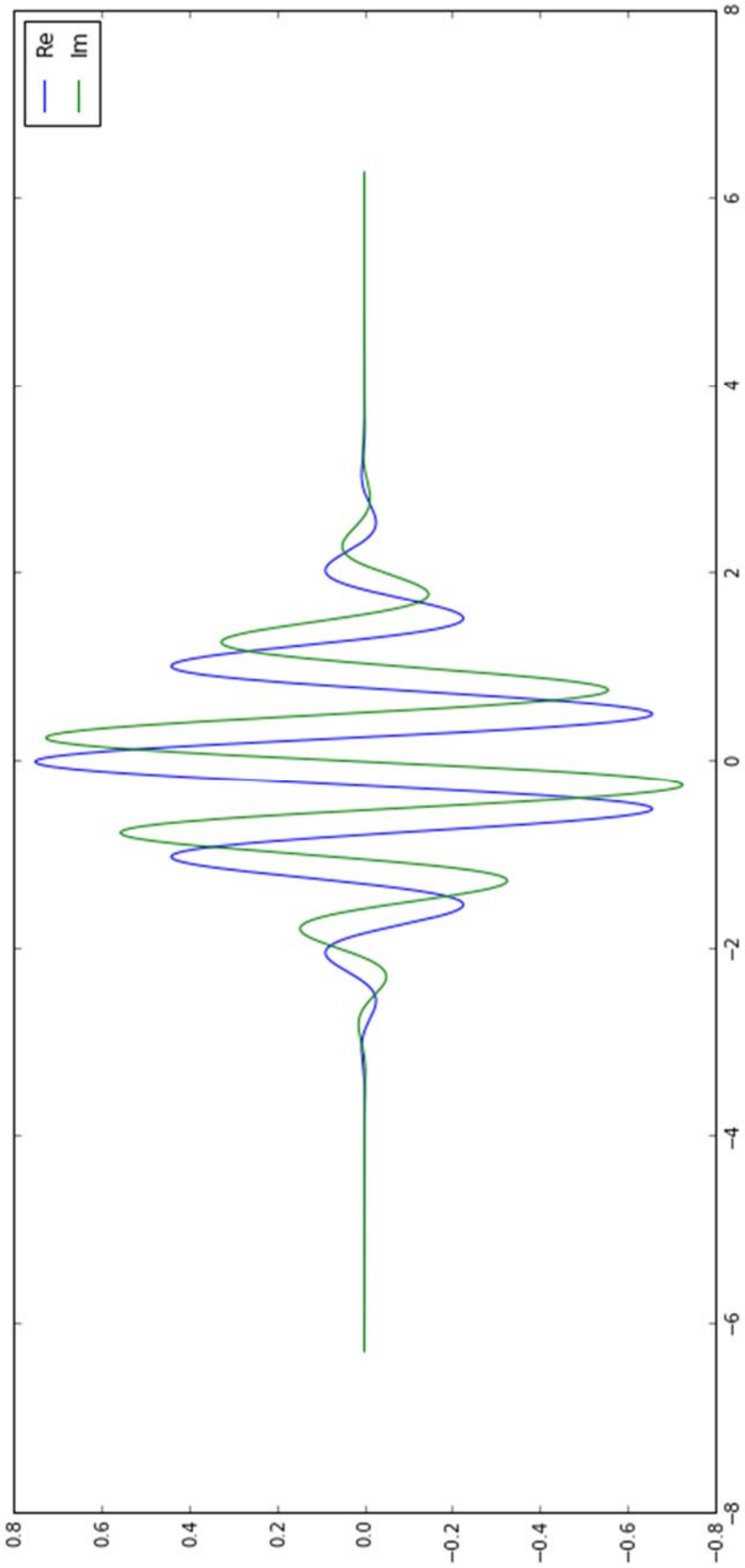


Рисунок 4.2 – Вейвлет Морле

Тогда непрерывное вейвлет-преобразование определяется как свертка временного ряда x_n с масштабированным и смещенным вейвлетом $\psi(\eta)$:

$$W_n(s) = \sum_{n'=0}^{N-1} x_{n'} \psi^* \left[\frac{(n'-n)\delta t}{s} \right], \tag{4.2}$$

где звездочкой обозначена процедура комплексного сопряжения. Если комплексное число $z = x + iy$, то число $z^* = x - iy$ называется сопряженным (или комплексно сопряженным) к z.

Для нормализации параметров масштабов вейвлета используется параметр длины волны Фурье вейвлета. Для вейвлета Морле:

$$\lambda = \frac{4\pi}{\omega_0 + \sqrt{2 + \omega_0^2}}. \tag{4.3}$$

Если визуализировать результат вейвлет преобразования в виде двухмерной карты, то получается т.н. скалограмма, показывающая как изменятся со временем частота анализируемого временного ряда.

Задание 4.1

Реализовать алгоритм вейвлет-преобразования с помощью (4.2) и построить скалограмму (рисунок 4.3) временного ряда, заданного формулой 4.1.

Пример выполнения задания

Ниже показан возможный вариант выполнения задания 4.1:

```
#Функция комплексного сопряжения
def conj(a):
    return np.real(a)-1j*np.imag(a)
#Параметры смещений
n=np.arange(Nlo , Nhi)
#Параметры масштабов
s=np.array([1, 50, 100, 150, 200])
w0=6. #Частота вейвлета
fwl=4* np.pi/(w0 + np.sqrt(2.0+w0**2)) # длина волны вейвлета, формула
(4.3)
sf=s/fwl #Нормализация масштабов
#вычисление по формуле (4.2)
W=np.zeros([len(s), len(n)],dtype=np.complex64)
for i in range(len(s)):
    for j in range(len(n)):
        for n_ in range (Ns):
            W[i,j]= W[i,j]+x[n_]*conj(morlet((n_-n[j])*dt/sf[i],w0=w0))
pl.rcParams["figure.figsize"] = (15,4)
im=pl.imshow(np.abs(W),
             cmap=pl.cm.jet,
             extent=[n[0],n[-1],s[-1],s[0]],
             aspect='auto',
             interpolation="nearest")
pl.ylim(s[0],s[-1])
_=pl.yticks(s)
```

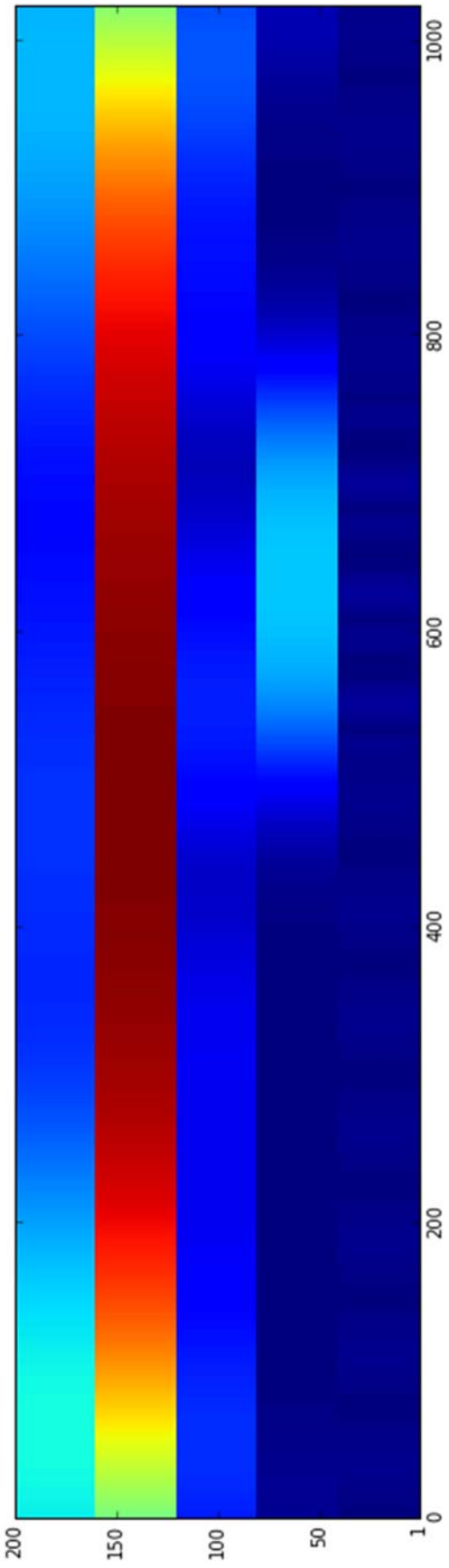


Рисунок 4.3 – Скалограмма модельного временного ряда, полученная с помощью формулы 4.2

Вейвлет-преобразование с использованием БПФ

Согласно теореме о свертке вейвлет-преобразование равно:

$$W_n(s) = \sum \hat{x}_k \hat{\psi}^*(s\omega_k) e^{i\omega_k n\delta t}, \tag{4.4}$$

где угловая частота ω_k определяется как:

$$\omega_k = \begin{cases} \dfrac{2\pi k}{N\delta t}, k \le N/2 \\ \dfrac{-2\pi k}{N\delta t}, k > N/2 \end{cases}. \tag{4.5}$$

$\hat{x}_k$ – дискретное преобразование Фурье исходного временного ряда x_k, $\hat{\psi}(s\omega)$ – непрерывное Фурье преобразование вейвлета $\psi(\eta)$. В частности для вейвлета Морле (рисунок 4.4):

$$\hat{\psi}_0(s\omega) = \pi^{-1/4} H(\omega) e^{-(s\omega-\omega_0)^2/2},$$

где $H(\omega)$ – функция Хэвисайда, $H(\omega) = 1$, если $\omega > 0$, $H(\omega) = 0$ в других случаях.

```
def hat_morlet(s_omega, w0=5.):
    H= np.ones(len(s_omega))
    n=len(s_omega)
    H[s_omega < 0.0] = 0.0 #функция Хэвисайда
    xhat=0.75112554*( np.exp(-(s_omega-w0)**2/2.0))*H
    return xhat

s_omega = np.linspace(0, 10, 1024)
m = hat_morlet(s_omega)
_ = pl.plot(s_omega, m)
```

Используя формулу (4.5) и реализованную ранее процедуру быстрого преобразования Фурье (рисунок 3.5), можно эффективно вычислять вейвлет-преобразование для заданных масштабов и одновременно для всех смещений.

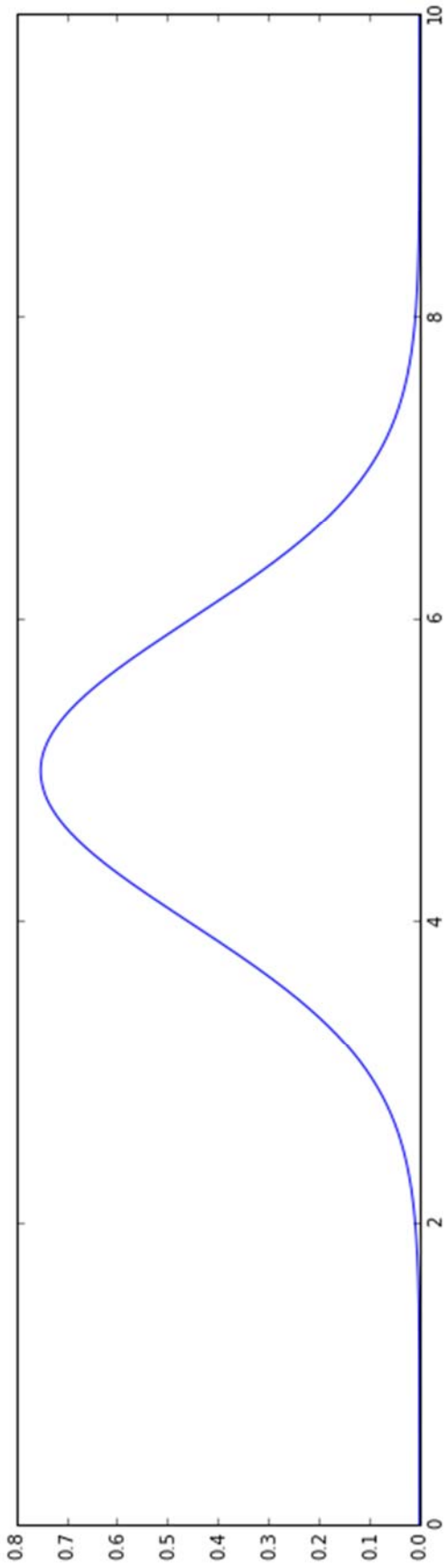


Рисунок 4.4 – Фурье-преобразование вейвлета Морле

Задание 4.2

Реализовать алгоритм вейвлет-преобразования с помощью БПФ и построить скалограмму временного ряда, заданного формулой (4.1).

Пример выполнения задания

```
# Функции, вычисляющие БПФ
def _W(k,N, n=1, inv=False):
    w=-2*np.pi*1j*k*n/N
    if not inv: return np.exp(w)
    else: return np.exp(-w)

def fft(x, inv=False):
    N=len(x)
    if N % 2 > 0:
        raise ValueError("Количество отсчетов должно быть степенью 2-ки")
    Y1=[]
    Y2=[]
    if N == 2: #условие окончания рекурсии
        return [x[0]+x[1], x[0]-x[1]]
    else:
        x1=x[::2]  #четные
        x2=x[1::2] #нечетные
        X1 = fft(x1, inv=inv)
        X2 = fft(x2, inv=inv)
    for k in range(N/2):
        tmp=_W(k,N, inv=inv)*X2[k]
        Y1.append(X1[k]+ tmp)
        Y2.append(X1[k]- tmp)
    return Y1+Y2 #объединение двух списков
```

```
s=np.arange(1,250, step=1) #масштабы вейвлета
hat_x =fft(x)
omega_ = [2* np.pi * k/ Ns/dt for k in range(0, Ns/2)] #формула (5)
_omega = [-2* np.pi * k/ Ns/dt for k in range(Ns/2, Ns)]
omega = np.array(omega_+_omega)
w0=5.
fwl=4* np.pi/(w0 + np.sqrt(2.0+w0**2))
sf=s/fwl
# Реализация формулы (4), но с учетом того, что затем будет применено
обратное БПФ
Ws=np.zeros([len(s), Ns],dtype=np.complex64)
for scaleindex in range(len(s)):
    currentscale=sf[scaleindex]
    s_omega = omega*currentscale
    psihat=hat_morlet(s_omega,w0)
    convhat = psihat * hat_x
    W = np.array(fft(convhat, inv=True))
    Ws[scaleindex,:]=W[:]/Ns
```

Используя БПФ можно построить скалограмму для 250 различных значений масштабов (рисунок 4.5). Кроме того, это позволяет анализировать достаточно большие временные ряды.

```
pl.rcParams["figure.figsize"] = (15,8)
pl.subplot(2,1,1)
pl.plot(t,x)
_=pl.xlim([0, Ns])
pl.subplot(2,1,2)
im=pl.imshow(np.abs(Ws),
             cmap=pl.cm.jet,
             extent=[n[0],n[-1],s[-1],s[0]],
             aspect='auto',
             interpolation="nearest")
_=pl.ylim(s[0],s[-1])
```

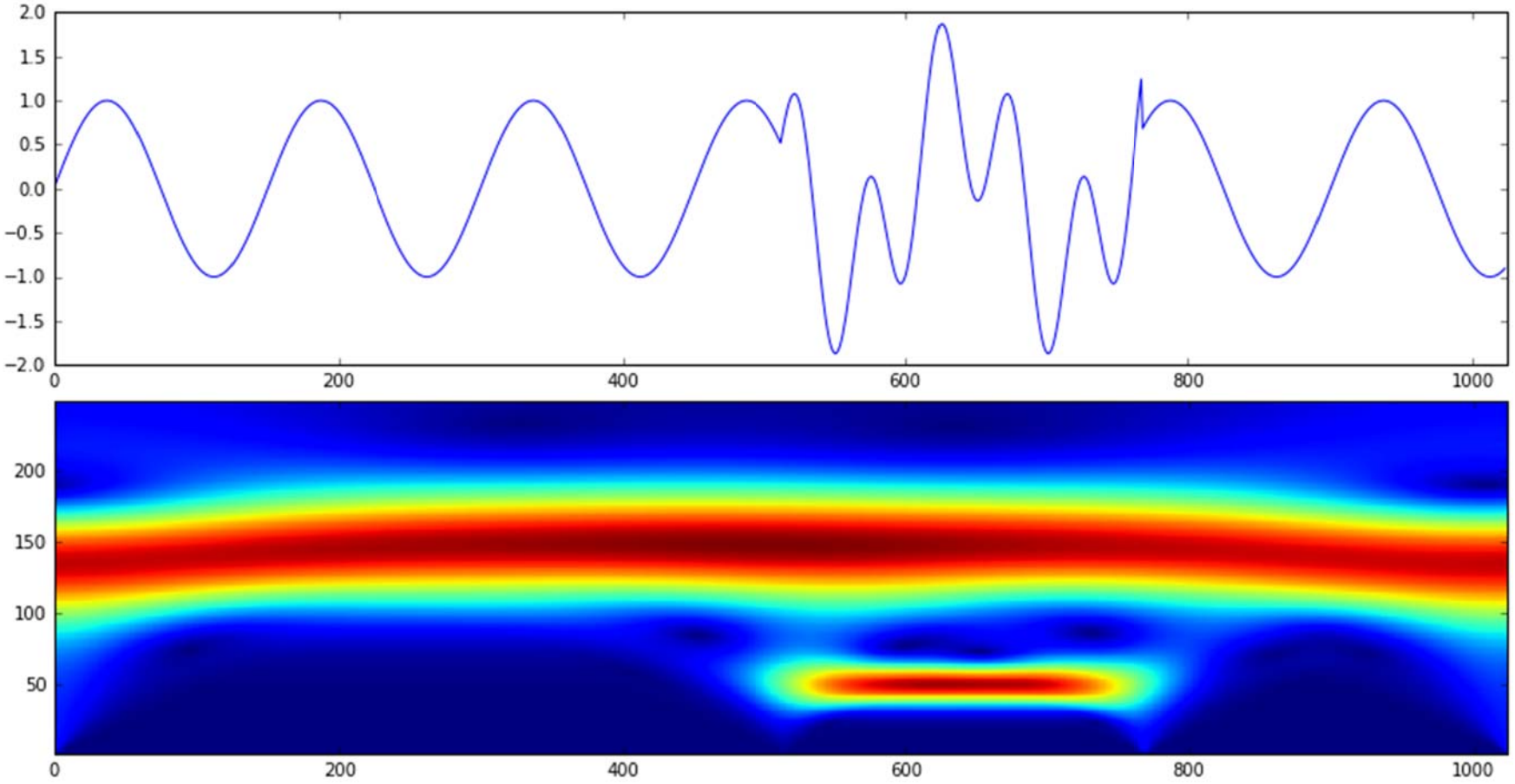


Рисунок 4.5 – Детализированная скалограмма модельного временного ряда, полученная с помощью БПФ

Для сравнения, на рисунке 4.6 показаны периодограммы тестового временного ряда, полученные в результате обычного БПФ:

```
pl.rcParams["figure.figsize"] = (15,4)
X=fft(x)
A=np.abs(X)
P=np.angle(X)
N=len(A)
nu=np.arange(N)/dt/N
np.seterr(divide='ignore')
T=1/nu
pl.subplot(2,1,1)
pl.semilogy(T[0:N//2], A[0:N//2])
pl.xlim([0, 400])
pl.grid()
pl.subplot(2,1,2)
pl.plot(T[0:N//2], P[0:N//2])
pl.xlim([0, 400])
pl.grid()
```

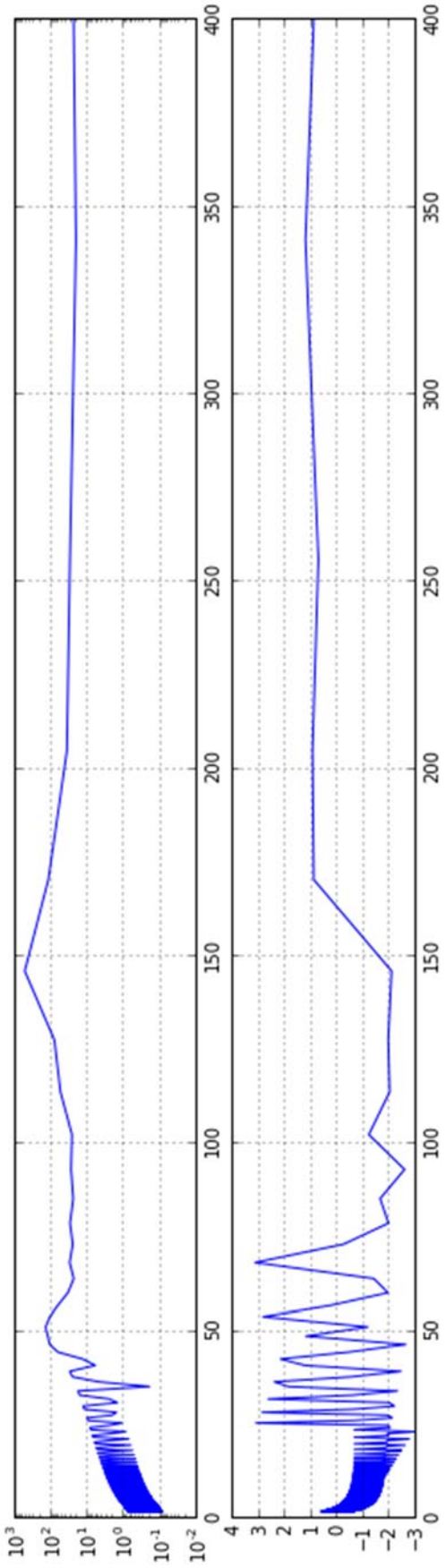


Рисунок 4.6 – Периодограмма модельного временного ряда, полученная с помощью БПФ

Таким образом, используя вейвлет-преобразование можно не только уставить наличие тех или иных частот в исследуемом временном ряду, но и проследить за их эволюцией во времени.

Задания для самостоятельной работы

1. Выясните, как влияет значение параметра ω_0 на результат вейвлет-преобразования.
2. Выполните Фурье и вейвлет-анализ среднегодового количества солнечных пятен (чисел Вольфа, рисунок 4.7). Выясните, изменяется ли со временем период колебаний количества солнечных пятен?

```
pl.rcParams["figure.figsize"] = (15,4)
url = 'http://www.sidc.be/silso/DATA/SN_y_tot_V2.0.txt'
data = np.genfromtxt(url)
y=data[:,0]
N=data[:,1]
_=pl.plot(y,N)
```

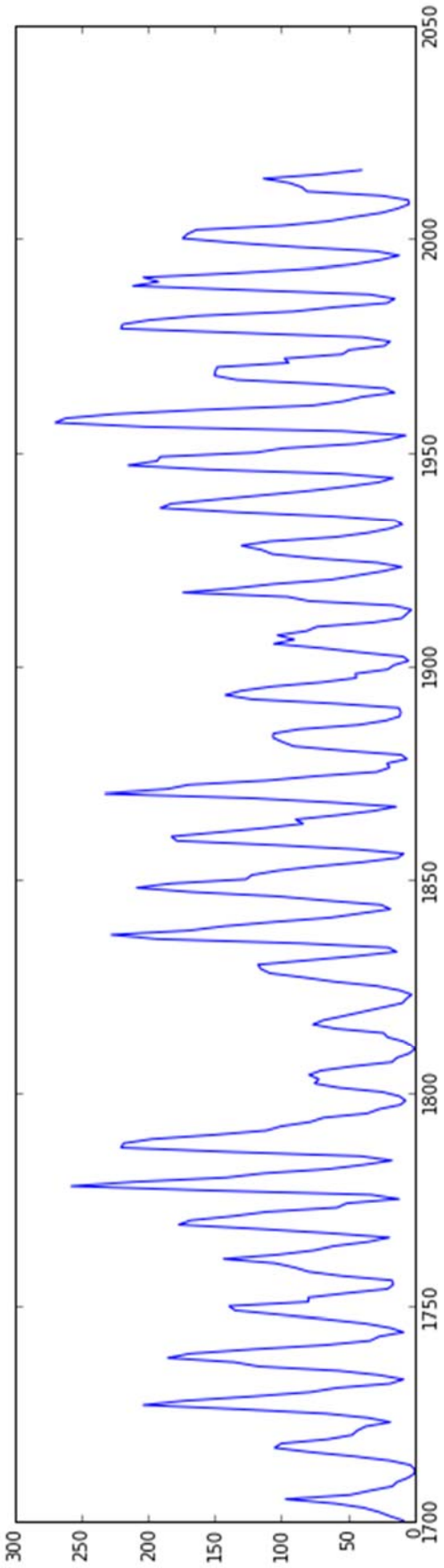


Рисунок 4.7 – Числа Вольфа

ЛИТЕРАТУРА

1. Блаттер К. Вейвлет-анализ. Основы теории // М.: Техносфера, 2006. 279 с.
2. Брякин И. В., Верзунов С. Н., Лыченко Н. М. Методы предварительной обработки результатов физического эксперимента. Бишкек: ИАИТ, 2017. 92 с.
3. Верзунов С. Н. Вейвлет-преобразование как инструмент анализа магнитовариационных данных // Проблемы автоматики и управления. Бишкек: Илим, 2014. № 2. С. 52–61.
4. Витязев В. В. Вейвлет-анализ временных рядов: учеб.пособие. СПб.: Изд-во СПб. ун-та, 2001. 65 с.
5. Гонсалес Р., Вудс Р. Цифровая обработка изображений // пер. с англ. М.: Техносфера, 2006. 1072 с.
6. Марпл С. Л. Цифровой спектральный анализ. М.: Мир, 1990.
7. Старовойтов В. В., Талеб М. А. Методы сегментации цветных изображений. Минск: Ин-т техн. кибернетики НАН Беларуси, 1999. 44 с.
8. Тропченко А. Ю. Методы вторичной обработки изображений и распознавания объектов: учеб. пособие. СПб.: СПбГУ ИТМО, 2012. 52 с.
9. Compo Gilbert P. A Practical Guide to Wavelet Analysis // Amer. Meteor. Soc. In Bulletin of the American Meteorological Society. 1998, vol. 79, no. 1. P 61–78.
10. Hayes M. H. Statistical Digital Signal Processing and Modeling. New York: John Wiley & Sons, 1996.
11. Orfanidis S. J. Introduction to Signal Processing. Upper Saddle River, NJ: Prentice Hall, 1996.

Сергей Николаевич Верзунов,
Мелис Садыкбекович Осмонов

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ
ДЛЯ ВЫПОЛНЕНИЯ ПРАКТИЧЕСКИХ РАБОТ
ПО КУРСУ «ЦИФРОВАЯ ОБРАБОТКА СИГНАЛОВ»

Редактор *А. А. Матвиенко*
Компьютерная верстка – *Ю. Ф. Атаманов*

Подписано в печать 11.07.18.
Формат 60x84 1/8
Офсетная печать. Объем 9,0 п. л.
Тираж 100 экз. Заказ 194

Отпечатано в типографии КРСУ
720048, г. Бишкек, ул. Горького, 2