

Содержание

Введение	3
1 Методы детектирование кисти человека в системах человеко-машинного взаимодействия	4
1.1 Непосредственное отделение фона изображения	4
1.2 Метод распознавания кожи в HSV и YCbCr цветовых моделях	7
1.2.1 Цветовая модель HSV	7
1.2.2 Цветовая модель YCbCr	10
1.2.3 Использование цветовых моделей RGB и YCrCb для распознавания кожи.	11
1.3 Метод Оцу	15
1.4 Рекурсивные методы построения модели фона	19
1.4.1 Visual Background Extractor.	19
1.4.2 Mixture of Gaussians.	21
2 Методы построения контура объекта на изображении	24
2.1 Выделение границ с помощью оператора Кэнни	24
2.2 Топологический структурный анализ цифрового бинарного изображения с помощью отслеживания границ.	26
2.2.1 Обзор функции <code>findContours()</code>	27
2.2.2 Обзор функции <code>drawContours()</code>	28
2.2.3 Формула площади Гаусса.	29
3 Методы нахождения ключевых точек на кисти.	32
3.1 Нахождение точек путём определения дефектов выпуклости.	32
3.1.1 Алгоритм Грэхема.	32
3.1.2 Алгоритм Джарвиса.	35
3.1.3 Алгоритм Киркпатрика.	36
3.1.4 Сравнение алгоритмов.	37
3.1.5 Обзор функции <code>convexHull()</code>	38
3.1.6 Обзор функции <code>convexityDefects()</code>	40
3.2 Локализация ключевых точек кисти руки на изображении на основе непрерывного скелета.	44
Список литературы	47

Введение

Развитие технологий, продолжающееся уже более полувека, приводит к тому, что нижняя граница размеров процессоров уменьшается, в то время как их производительность продолжает увеличиваться. Для рядового пользователя результатом этого является разнообразие созданных интеллектуальных систем, используемых в повседневной жизни: от смартфонов до планшетов, от бытовой техники до домашних роботов. Камнем преткновения становится обмен информацией между компьютером и пользователем, растёт потребность в исследовании новых способов, более естественных, чем ввод данных с клавиатуры, эмулирующих бытовое общение между людьми. Активно развивающимся направлением решения проблемы является управление компьютером с помощью голосовых команд. Тем не менее, несмотря на значительные успехи в области, этот способ применим далеко не во всех ситуациях. Другим исследуемым способом ввода данных является использование визуальных систем: передача информации посредством мимики и жестов. Для реализации последнего метода можно воспользоваться многими способами.

В данной работе идёт речь о поиске так называемых особых точек на кисти руки для дальнейшей их обработки. Поиск таких точек позволяет извлекать конфигурацию кисти и давать её описание, что позволяет не только задавать исполнение определённых команд по заранее определённым жестам, но и выполнять совершенно не характерные для обычной реализации данного метода действия.

В ходе исследования разрабатывается алгоритм детектирования кисти руки человека и её описания с помощью "особых" точек. Важным требованием для реализации является минимизация времени задержки на обработку каждого кадра для комфорtnого использования в прикладных задачах.

Поставленной **целью** является создание программы для детектирования и извлечения конфигурации кисти руки человека с помощью языка высокого уровня Python.

Для достижения цели необходимо решить ряд **задач**:

1. Сделать обзор теоретического материала по детектированию кисти руки человека и её последующего описания с помощью "особых" точек.
2. Найти и описать наиболее популярные методы обнаружения кисти человека и выделить из них самые эффективные.
3. Ознакомиться с методами извлечения конфигураций кисти человека.
4. Реализовать несколько алгоритмов данной задачи и провести их сравнение.

1 Методы детектирование кисти человека в системах человеко-машинного взаимодействия

Человеко-машинное взаимодействие (Human-computer interaction - HCI) - это междисциплинарное научное направление, изучающее взаимодействие между людьми и машинами. Предметом HCI является изучения, планирование и разработка методов взаимодействия человека с машиной, где в роли машины может выступать персональный компьютер, компьютерная система больших масштабов, система управления процессами и т.д. [1]. Под взаимодействием понимается любая коммуникация между человеком и машиной. Одним из методов HCI, получившим широкое распространение в последние годы, является взаимодействие, основанное на жестах человека [2, 3].

Задачу распознавания жестов руки можно разделить на подзадачи:

1. Отделение кисти руки от остальной части изображения.
2. Построение контура кисти.
3. Нахождение ключевых точек на кисти.
4. Классификация жеста исходя из статического или динамического расположения точек.

Первая подзадача, а именно детектирование кисти человека в кадре является ключевой, поскольку от качества её решения зависит качество выполнения остальных подзадач. Рассмотрим первую подзадачу, а именно детектирование кисти человека в кадре.

Существует множество решений этой подзадачи. Наиболее популярными из них являются непосредственное отделение фона изображения, распознавание цвета кожи в кадре и метод Оцу. Подробно рассмотрим каждое из них.

1.1 Непосредственное отделение фона изображения

В данном решении принимается, что в кадре движется только рука, а остальные части тела, включая фон, остаются неподвижными. Таким образом, если вначале инициализировать фон как $bgr(x, y)$, а новое изображение с жестом рассматривать как $fgr(x, y)$, то изолированный жест можно принять как разность между этими изображениями:

$$gst_i(x, y) = fgr_i(x, y) - bgr(x, y). \quad (1)$$

Полученный разностный жест переднего плана преобразуется в бинарное изображение, устанавливая соответствующий порог. Поскольку фон не является полностью статичным, например, если камера удерживается в руках оператора, то добавляется шумовая часть. Чтобы получить изображение руки без шумов, этот метод сочетается с распознаванием кожи человека. Чтобы удалить этот шум, применяется анализ связанных компонентов, чтобы заполнить пустоты, если применяется заливка какой-либо

области, а также для получения чётких краёв применяется морфологическая обработка.

Недостаток данного решения состоит в том, что довольно сложно отделить фон, даже если он задан, поскольку не ясно какие из пикселей изменились, а какие остались прежними из-за тени, смещения фокуса, изменения экспозиции и т.д. Даже если зафиксировать все параметры камеры, то тень так или иначе испортит качество решения.

Приведём два примера отделения кисти от фона изображения. На рис. 1 изображены два изображения, одно из которых является инициализированным фоном, а второе – кисть на этом фоне.



а)



б)

Рис. 1. Инициализированный фон (а) и изображение кисти на нём (б).

Для отделения кисти первым способом воспользуемся встроенным в библиотеку OpenCV методом `absdiff()`, код для которого представлен ниже:

```
1 bg = cv2.imread('BG_result.png') # фон
2 fg = cv2.imread('FG_result.png') # фон с кистью
3 diff = cv2.absdiff(fg, bg) # вычисляем разность изображений
4 mask = cv2.cvtColor(diff, cv2.COLOR_BGR2GRAY) # переводим в ЧБ
5 th = 30
6 imask = mask > th
7 canvas = np.zeros_like(fg, np.uint8)
8 canvas[imask] = fg[imask]
9 cv2.imwrite('result.png', canvas) # сохраняем кисть
```

Второй способ будет заключаться в следующем. Пусть даны два пикселя

$$p_1 = (r_1, g_1, b_1) \quad p_2 = (r_2, g_2, b_2), \quad (2)$$

тогда различие между ними будем определять как

$$D = \sqrt{(r_2 - r_1)^2 + (g_2 - g_1)^2 + (b_2 - b_1)^2}. \quad (3)$$

Затем создаём битовую маску M , в которой значения определяются как

$$M_{(i,j)} = \begin{cases} 1, & D > Threshold \\ 0, & D \leq Threshold \end{cases} \quad (4)$$

Код для этого метода представлен ниже:

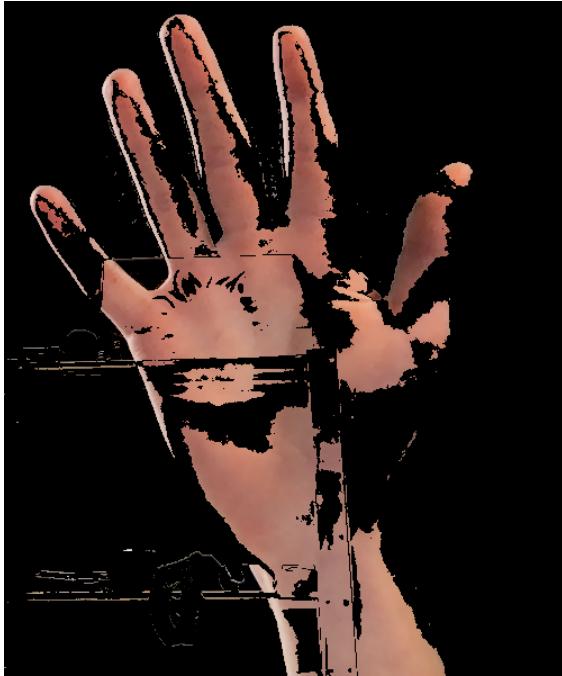
```

1  bg = cv2.imread('BG_result.png')
2  fg = cv2.imread('FG_result.png')
3  def difference(pixel1, pixel2):
4      return math.sqrt((int(pixel1[0]) - int(pixel2[0]))**2 +
5                      (int(pixel1[1]) - int(pixel2[1]))**2 +
6                      (int(pixel1[2]) - int(pixel2[2]))**2)
7
8  THRESHOLD = 60
9  mask = np.zeros_like(bg, np.uint8)
10 for i in range(len(bg)):
11     for j in range(len(bg[i])):
12         mask[i][j] = 1 if \
13             difference(bg[i][j], fg[i][j]) > THRESHOLD else (0, 0, 0)
14 cv2.imwrite('result1.png', mask)

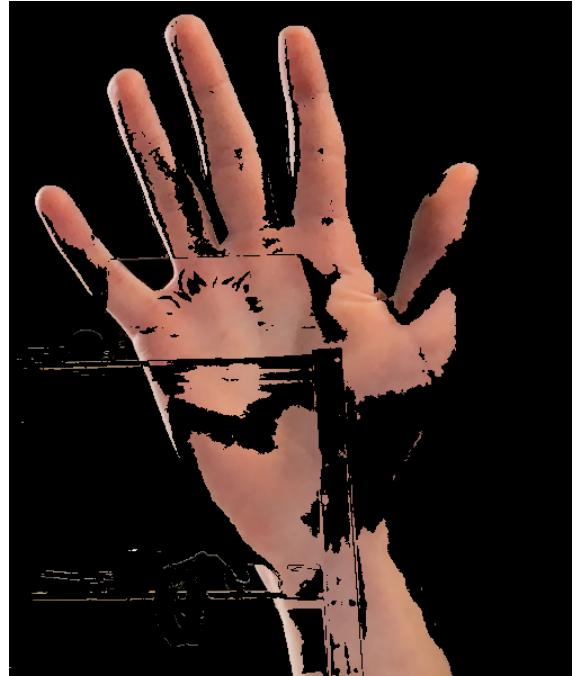
```

Результаты работы первого и второго метода представлены на рис. 2.

Как можно видеть, оба метода не совсем точно отделяют нужный объект от фона, поэтому рассмотрим метод распознавания кожи в HSV и YCbCr цветовых моделях.



а)



б)

Рис. 2. Результат первого метода (а) и второго (б).

1.2 Метод распознавания кожи в HSV и YCbCr цветовых моделях

Для того, чтобы исследовать соответствующие методы следует познакомиться с цветовыми моделями, которые они используют, поподробнее.

1.2.1 Цветовая модель HSV

HSV (*Hue, Saturation, Value*) или **HSB** (*Hue, Saturation, Brightness*) – цветовая модель, в которой координатами цвета являются:

1. **Hue** – цветовой тон, (например, красный, зелёный или сине-голубой). Варьируется в пределах 0-360°, однако иногда приводится к диапазону 0-100 или 0-1.
2. **Saturation** – насыщенность. Варьируется в пределах 0-100 или 0-1. Чем больше этот параметр, тем "чище" цвет, поэтому иногда называют *чистотой цвета*. А чем ближе этот параметр к нулю, тем ближе цвет к нейтральному серому.
3. **Value** или **Brightness** – яркость. Также задаётся в пределах 0-100 или 0-1.

Простейший способ отобразить HSV в трёхмерное пространство – воспользоваться цилиндрической системой координат (рис. 3). Здесь координата H определяется полярным углом, S – радиус-вектором, а V – Z-координатой. То есть, оттенок изменяется при движении вдоль окружности цилиндра, насыщенность – вдоль радиуса, а яркость – вдоль высоты. Несмотря на "математическую" точность, у такой модели есть существенный недостаток: на практике количество различимых глазом уровней насыщенности

сти и оттенков уменьшается при приближении яркости (V) к нулю. Также на малых S и V появляются существенные ошибки округления при переводе RGB в HSV и наоборот. Поэтому чаще применяется коническая модель (рис. 4).

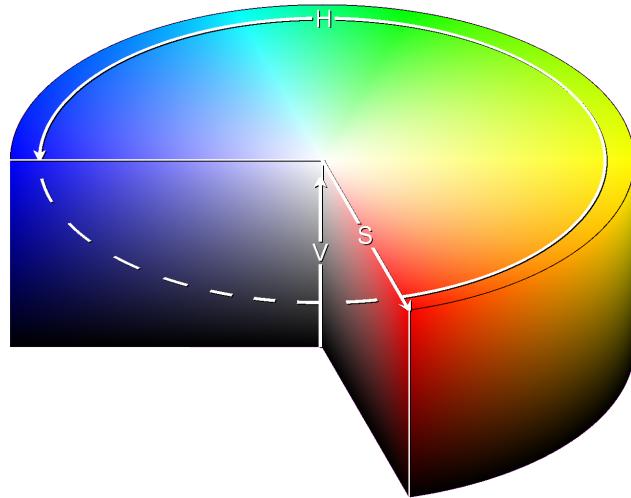


Рис. 3. HSV в цилиндрической системе координат.

Как и в цилиндре, в конической модели оттенок изменяются по окружности конуса. Насыщенность цвета возрастает с удалением от оси конуса, а яркость – с приближением к его основанию. Иногда вместо конуса используется шестиугольная правильная пирамида.

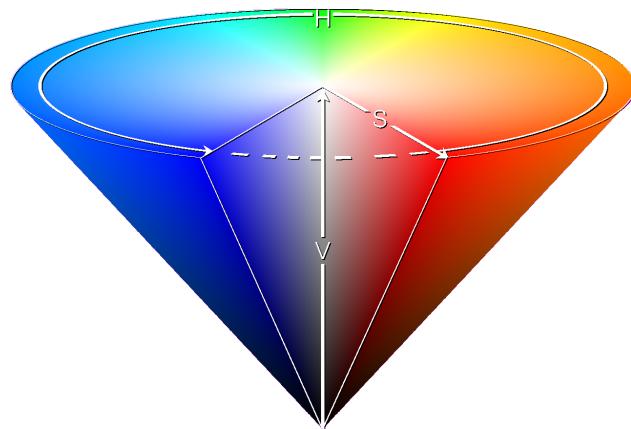


Рис. 4. HSV в конической модели.

Для перевода $\text{RGB} \Rightarrow \text{HSV}$ будем считать, что

$$H \in [0, 360],$$

$$S, V, R, G, B \in [0, 1].$$

Пусть $MAX = \max(R, G, B)$, а $MIN = \min(R, G, B)$. Тогда

$$H = \begin{cases} 60 \cdot \frac{G - B}{MAX - MIN} + 0, & MAX = R \quad G \geq B \\ 60 \cdot \frac{G - B}{MAX - MIN} + 360, & MAX = R \quad G < B \\ 60 \cdot \frac{B - R}{MAX - MIN} + 120, & MAX = G \\ 60 \cdot \frac{R - G}{MAX - MIN} + 240, & MAX = B \end{cases} \quad (5)$$

$$S = \begin{cases} 0, & MAX = 0, \\ 1 - \frac{MIN}{MAX}, & \end{cases} \quad (6)$$

$$V = MAX \quad (7)$$

Можно заметить, что уравнение (5) не определено, когда $MAX = MIN$, поэтому существуют другие уравнения:

$$\begin{aligned} H &= \arccos \frac{0.5 \cdot ((R - G) + R - B)}{\sqrt{(R - G)^2 + (R - B) + (G - B)}} \\ S &= 1 - 3 \frac{\min(R, G, B)}{R + G + B} \\ V &= \frac{1}{3}(R + G + B) \end{aligned} \quad (8)$$

Для перевода HSV \Rightarrow RGB будем считать, что $H \in [0, 360]$, $S \in [0, 100]$ и $V \in [0, 100]$.

Если

$$\begin{aligned} H_i &= \left\lfloor \frac{H}{60} \right\rfloor \bmod 6, \\ V_{min} &= \frac{(100 - S)V}{100}, \\ a &= (V - V_{min}) \frac{H \bmod 60}{60}, \\ V_{inc} &= V_{min} + a, \\ V_{dec} &= V - a, \end{aligned} \quad (9)$$

тогда по таблице 1 выбираем нужные значения. Полученные значения красного, зелёного и синего каналов RGB исчисляются в процентах. Чтобы привести их в соответствие распространённому представлению COLORREF необходимо умножить каждое из них на $\frac{255}{100}$. При целочисленном кодировании для каждого цвета в HSV есть соответствующий цвет в RGB. Однако обратное утверждение не является верным: некоторые цвета в RGB нельзя выразить в HSV так, чтобы значение каждого компонента было целым. Фактически, при таком кодировании доступна только $\frac{1}{256}$ часть цветового пространства RGB. Пример изображения в HSV изображён на рисунке 5.

H_i	R	G	B
0	V	V_{inc}	V_{min}
1	V_{dec}	V	V_{min}
2	V_{min}	V	V_{inc}
3	V_{min}	V_{dec}	V
4	V_{inc}	V_{min}	V
5	V	V_{min}	V_{dec}

Таблица 1. Таблица значений RGB для перевода из HSV.

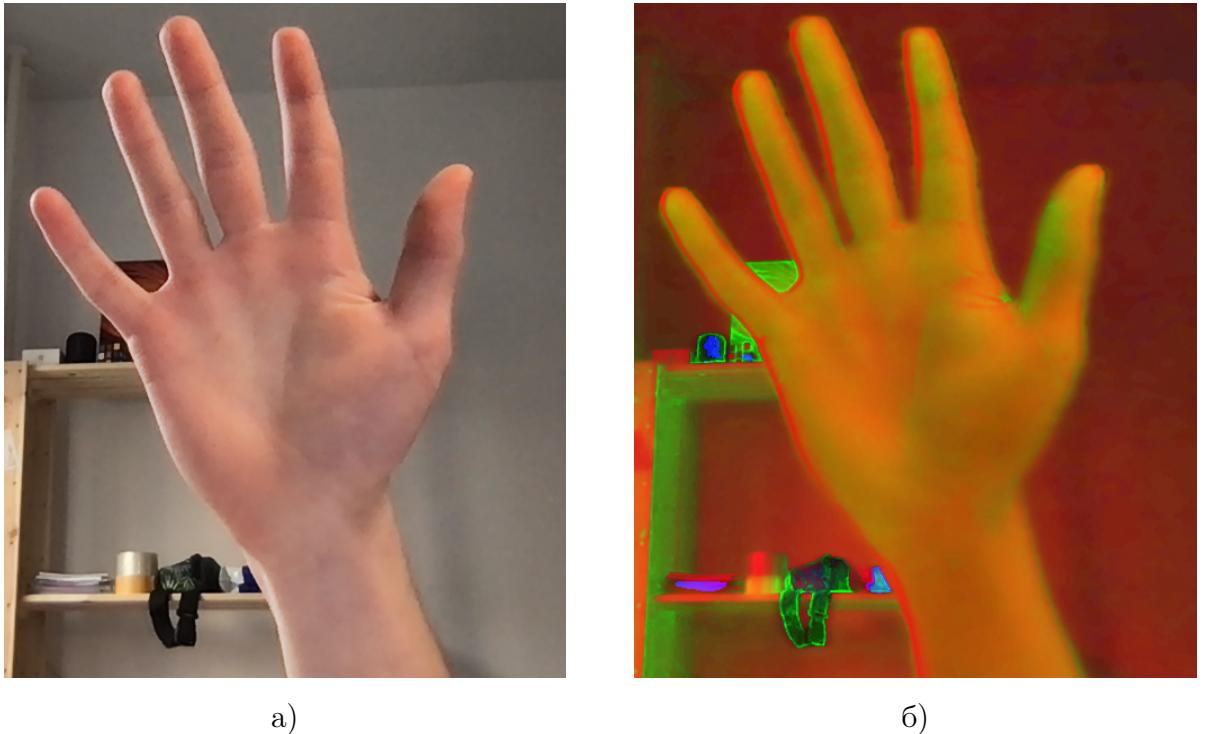


Рис. 5. Изображение в RGB (а) и HSV (б).

1.2.2 Цветовая модель YCbCr

Известно, что органы зрения человека менее чувствительны к цвету предметов, чем к их яркости. В цветовом пространстве RGB все три цвета считаются одинаково важными, и они обычно сохраняются с одинаковым разрешением. Однако можно отобразить цветное изображение более эффективно, отделив светимость от цветовой информации и представив её с большим разрешением, чем цвет. Цветовое пространство YC_BC_R и его вариации является популярным методом эффективного представления цветных изображений.

YC_BC_R – семейство цветовых пространств, которые используются для передачи цветных изображений в компонентном видео и цифровой фотографии. Является ча-

стью рекомендации МСЭ-R BT.601 при разработке стандарта видео Всемирной цифровой организации и фактически является масштабированной и смещённой копией YUV.

Y – компонента яркости, C_B – синий компонент цветности, C_R – красный компонент цветности.

Преобразование RGB $\Rightarrow YC_BC_R$ можно описать следующей формулой:

$$\begin{pmatrix} Y \\ C_B \\ C_R \end{pmatrix} = \begin{pmatrix} 16 \\ 128 \\ 128 \end{pmatrix} + \begin{pmatrix} 65.481 & 128.553 & 24.996 \\ -37.797 & -74.203 & 112 \\ 112 & -93.786 & -18.214 \end{pmatrix} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} \quad (10)$$

Пример изображения в YC_BC_R изображен на рис. 6.

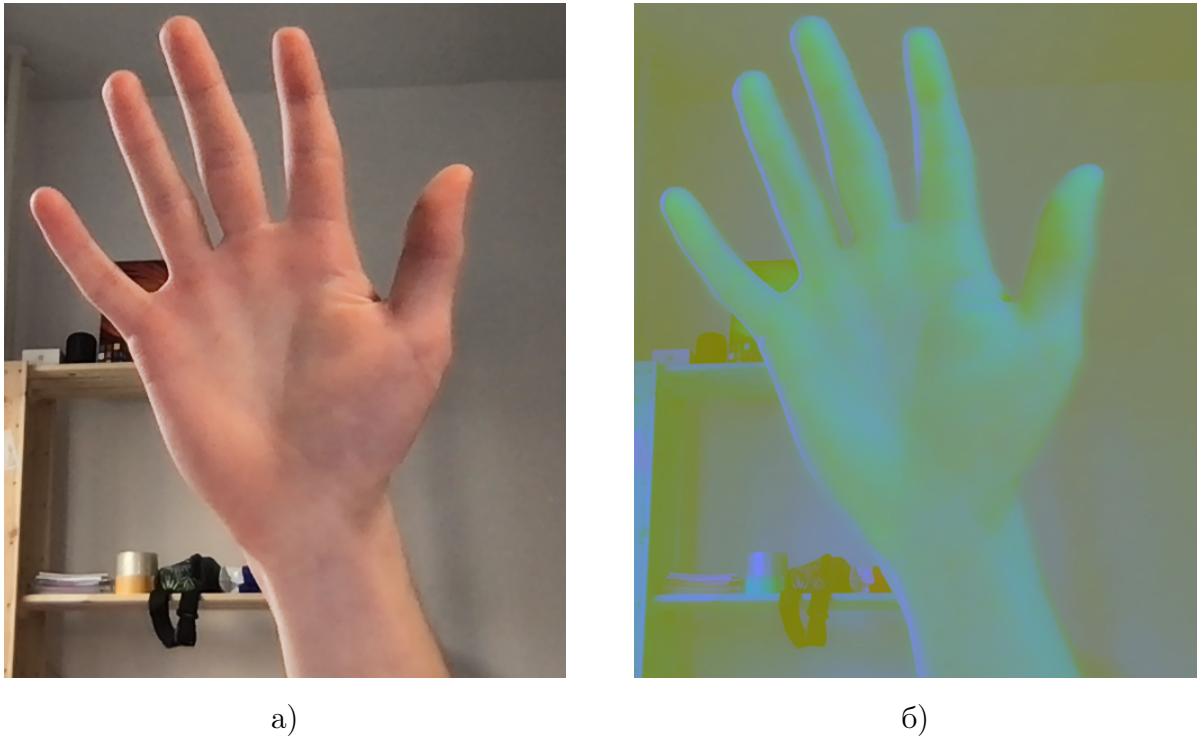


Рис. 6. Изображение в RGB (а) и YC_BC_R (б).

1.2.3 Использование цветовых моделей RGB и YCrCb для распознавания кожи.

Для того чтобы отделить кожу от остальной части изображения используют определённые диапазоны составляющих цветовых моделей, которые находятся эмпирически или итеративно. В первом случае путём проб и ошибок подбираются значения составляющих. Можно заметить, что при данном подходе кожа будет иметь разные цветовые диапазоны при разном освещении. Покажем это. Зададим диапазон для изображения в HSV цветовой модели как

$$0 \leq H \leq 200,$$

$$15 \leq S \leq 255,$$

$$80 \leq V \leq 255,$$

и, запустив код ниже с двумя разными изображениями одной и той же кисти, получим рис. 7.

```

1 import cv2
2 import numpy as np
3
4 LOWER, UPPER = np.array([0, 15, 80], dtype='uint8'), \
5                         np.array([200, 255, 255], dtype='uint8')
6
7 INPUT_TO_OUTPUT = {'image3.jpg': 'hsv_del1.png',
8                     'image7.jpg': 'hsv_del2.png'}
9
10 def HSV_skin_detection_mask_first(image):
11     converted = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
12     skinMask = cv2.inRange(converted, LOWER, UPPER)
13     kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (11, 11))
14     skinMask = cv2.erode(skinMask, kernel, iterations=2)
15     skinMask = cv2.dilate(skinMask, kernel, iterations=2)
16     skinMask = cv2.GaussianBlur(skinMask, (3, 3), 0)
17     return skinMask
18
19
20 for filename_input, filename_output in INPUT_TO_OUTPUT.items():
21     img = cv2.imread(filename_input)
22     hsv_mask = HSV_skin_detection_mask_first(img)
23     cv2.imwrite(filename_output, cv2.bitwise_and(img, img, mask=hsv_mask))

```

Можно видеть, что на первом изображении фон отделился практически идеально, но на втором видны фрагменты фона.

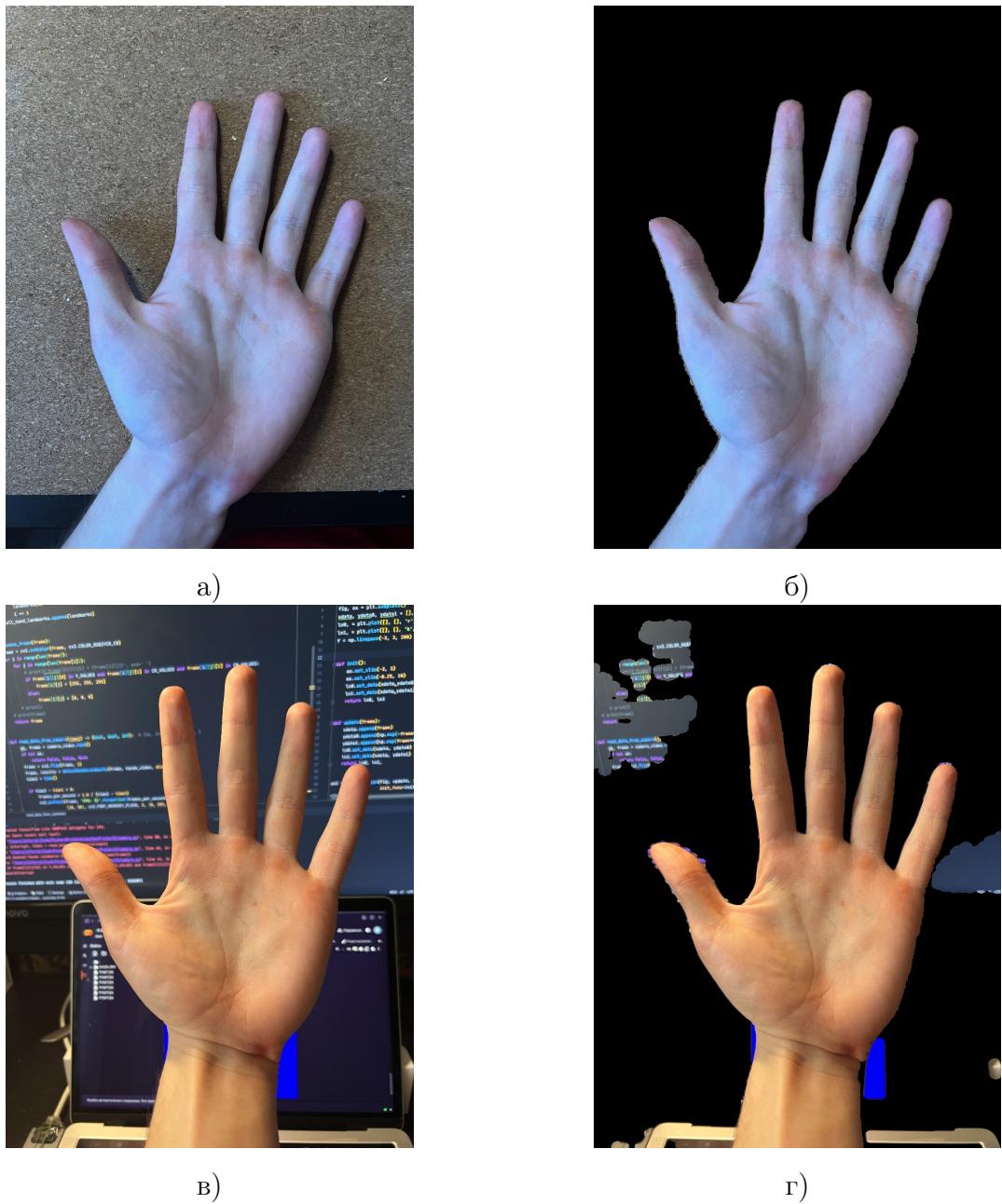


Рис. 7. Изображения с удалённым фоном в HSV.

Для цветовой модели YCbCr в статье [4] предложили два варианта диапазона компонент (11) и (12):

$$\begin{aligned} 80 < Y \leq 255, \\ 85 < C_b < 135, \\ 135 < C_r < 180 \end{aligned} \quad (11)$$

$$\begin{aligned} Y \in \forall, \\ 77 \leq C_b \leq 127, \\ 133 \leq C_r \leq 173 \end{aligned} \quad (12)$$

Выполнив код ниже, получаем сравнение двух методов на рисунке 8.

```
1 import cv2
2
3 LOWER_YCr_Cb1, UPPER_YCr_Cb1 = (79, 89, 134), \
4                                 (255, 134, 179)
5 LOWER_YCr_Cb2, UPPER_YCr_Cb2 = (0, 77, 133), \
6                                 (255, 127, 173)
7 INPUT_TO_OUTPUT = {'image3.jpg': ['ycbcr_del1_1.png', 'ycbcr_del1_2.png'],
8                    'image7.jpg': ['ycbcr_del2_1.png', 'ycbcr_del2_2.png']}
9
10 def YCrCb_skin_detection_mask(image, ex: int):
11     img_YCrCb = cv2.cvtColor(image, cv2.COLOR_BGR2YCR_CB)
12     img_YCrCb = cv2.GaussianBlur(img_YCrCb, (5, 5), 5)
13     if ex == 1:
14         skinMask = cv2.inRange(img_YCrCb, LOWER_YCr_Cb1, UPPER_YCr_Cb1)
15     else:
16         skinMask = cv2.inRange(img_YCrCb, LOWER_YCr_Cb2, UPPER_YCr_Cb2)
17     kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (11, 11))
18     skinMask = cv2.erode(skinMask, kernel, iterations=3)
19     skinMask = cv2.dilate(skinMask, kernel, iterations=3)
20     skinMask = cv2.GaussianBlur(skinMask, (3, 3), 0)
21     return skinMask
22
23 for filename, outputs in INPUT_TO_OUTPUT.items():
24     img = cv2.imread(filename)
25     for i, output in enumerate(outputs):
26         ycrcb_mask = YCrCb_skin_detection_mask(img, i)
27         image_skin = cv2.bitwise_and(img, img, mask=ycrcb_mask)
28         cv2.imwrite(output, image_skin)
```

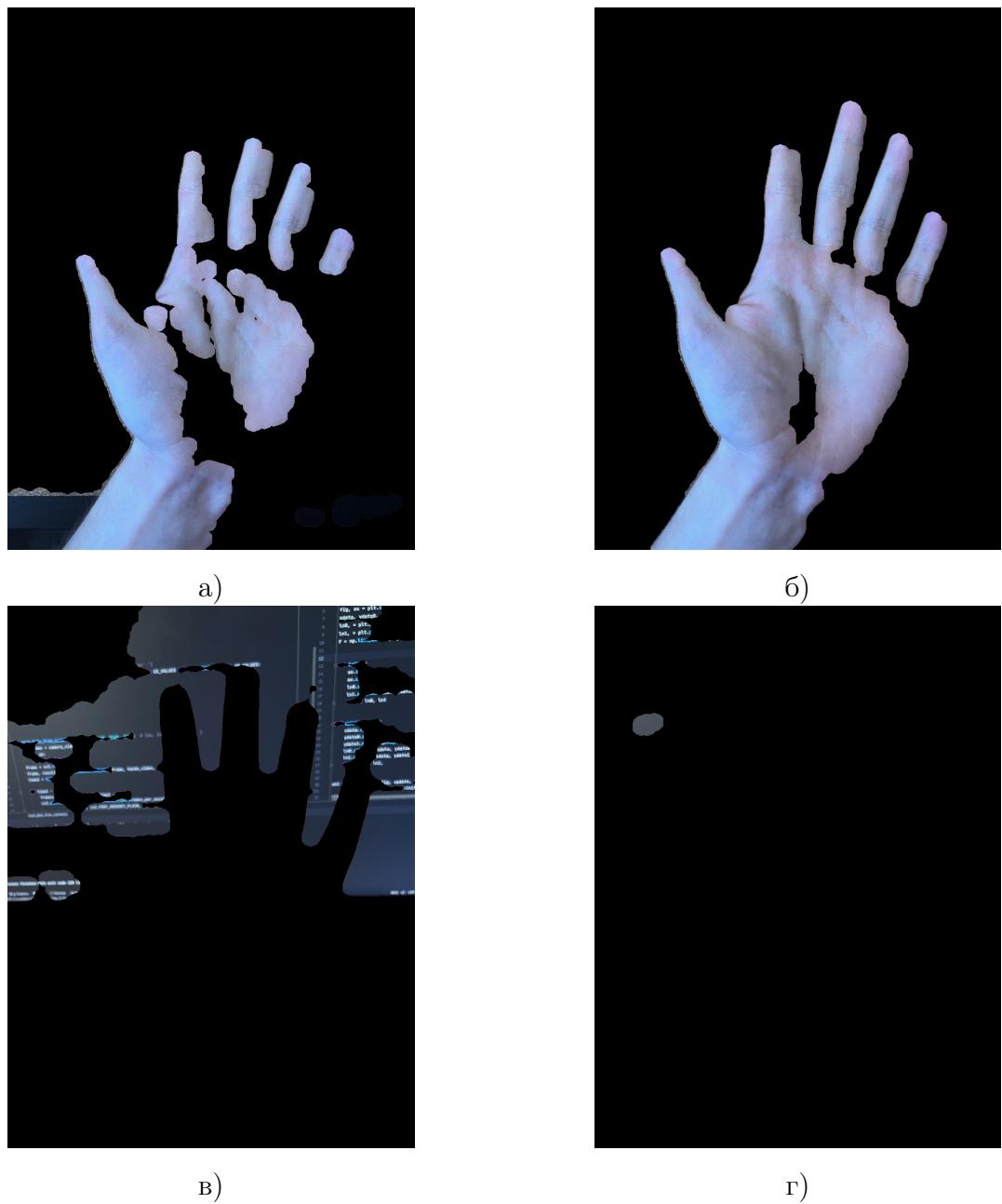


Рис. 8. Изображения с удалённым фоном в YCbCr.

Легко видеть, что оба диапазона работают не идеально, но первый показал себя намного лучше.

Таким образом, можно сделать вывод, что для детектирования цвета кожи лучше брать изображение в цветовой модели HSV.

Рассмотрим следующий метод отделения фона изображения, а именно метод Оцу.

1.3 Метод Оцу

В 1979 году Нобуюки Оцу опубликовал статью [5] метода порогового разделения, основываясь на гистограмме серых цветов изображения.

Метод Оцу – это алгоритм вычисления порога бинаризации для полутонаового изоб-

ражения, используемый в области компьютерного распознавания образов и обработки изображений для получения чёрно-белых изображений. Алгоритм позволяет разделить пиксели двух классов ("полезные" и "фоновые"), рассчитывая такой порог, чтобы внутриклассовая дисперсия была минимальной.

Пусть пиксели изображения представлены в L уровнях серого $[1, 2, \dots, L]$. Количество пикселей уровня i обозначим как n_i , а количество всех пикселей как

$$N = n_1 + n_2 + \dots + n_L.$$

Для того чтобы упростить алгоритм гистограмма монохромного изображения нормализована и рассматривается как распределение вероятностей:

$$p_i = n_i/N, \quad p_i \geq 0, \quad \sum_{i=1}^L p_i = 1. \quad (13)$$

Теперь предположим, что мы разделили пиксели на два класса C_0 и C_1 (фон и объект) по пороговому значению k . C_0 обозначает пиксели с уровнями $[1, \dots, k]$, а C_1 – пиксели с уровнями $[k+1, \dots, L]$. Следовательно, вероятности класса и средний уровень, соответственно, задаются как

$$\omega_0 = \sum_{i=1}^k p_i = \omega(k) \quad (14)$$

$$\omega_1 = \sum_{i=k+1}^L p_i = 1 - \omega(k) \quad (15)$$

и

$$\mu_0 = \sum_{i=1}^k \frac{ip_i}{\omega_0} = \frac{\mu(k)}{\omega(k)} \quad (16)$$

$$\mu_1 = \sum_{i=k+1}^L \frac{ip_i}{\omega_1} = \frac{\mu_T - \mu(k)}{1 - \omega(k)}, \quad (17)$$

где

$$\omega(k) = \sum_{i=1}^k p_i \quad (18)$$

и

$$\mu(k) = \sum_{i=1}^k ip_i. \quad (19)$$

$\omega(k)$ и $\mu(k)$ являются кумулятивными моментами нулевого и первого порядка соответственно, а

$$\mu_T = \mu(L) = \sum_{i=1}^L ip_i \quad (20)$$

является общим средним уровнем изначального изображения. Легко можно доказать, что следующие уравнения справедливы для любого k :

$$\omega_0\mu_0 + \omega_1\mu_1 = \mu_T, \quad \omega_0 + \omega_1 = 1. \quad (21)$$

Дисперсии классов определяются как

$$\sigma_0^2 = \sum_{i=1}^k \frac{(i - \mu_0)^2 p_i}{\omega_0}, \quad (22)$$

$$\sigma_1^2 = \sum_{i=k+1}^L \frac{(i - \mu_1)^2 p_i}{\omega_1}. \quad (23)$$

Оцу показал, что минимизация дисперсии *внутри* класса равносильна максимизации дисперсии *между* классами:

$$\sigma_W^2 = \omega_0 \sigma_0^2 + \omega_1 \sigma_1^2, \quad (24)$$

$$\sigma_B^2 = \omega_0 (\mu_0 - \mu_T)^2 + \omega_1 (\mu_1 - \mu_T)^2 = \omega_0 \omega_1 (\mu_1 - \mu_0)^2, \quad (25)$$

и

$$\sigma_T^2 = \sum_{i=1}^L (i - \mu_T)^2 p_i \quad (26)$$

являющееся внутриклассовой дисперсией, дисперсия между классами и общая дисперсия всех уровней соответственно. Таким образом, алгоритм можно записать как:

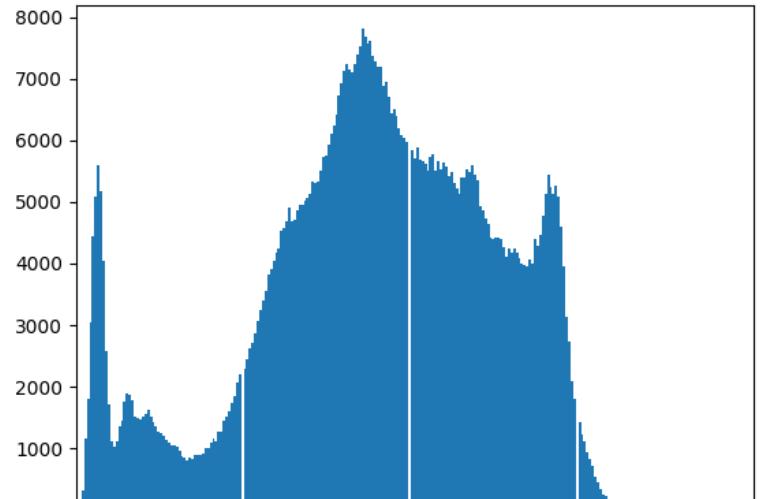
Пусть дано монохромное изображение $G(i, j)$, $i = \overline{1, Height}$, $j = \overline{1, Width}$. Счётчик повторений $k = 0$.

1. Вычислить гистограмму $p(l)$ изображения и частоту $N(l)$ для каждого уровня интенсивности изображения G .
2. Вычислить начальные значения для $\omega_0(0), \omega_1(0)$ и $\mu_1(), \mu_2(0)$.
3. Для каждого значения $t = \overline{1, max(G)}$ – полутона – горизонтальная ось гистограммы:
 - (a) Обновляем ω_0, ω_1 и μ_0, μ_1 .
 - (b) Вычисляем $\sigma_B^2(t) = \omega_0(t)\omega_1(t)[\mu_0(t) - \mu_1(t)]^2$.
 - (c) Если $\sigma_B^2(t)$ больше, чем имеющееся, то запоминаем σ_B^2 и значение порога t .
4. Искомый порог соответствует максимуму $\sigma_B^2(t)$.

Диаграммы для изображений изображены на рис. 9. Как видно из рисунков, на данных изображениях довольно проблематично отделить фон от изображения какой-то единственной пороговой величиной, поэтому и алгоритм Оцу на таких изображениях сработает не очень хорошо, как это показано на рис. 10.



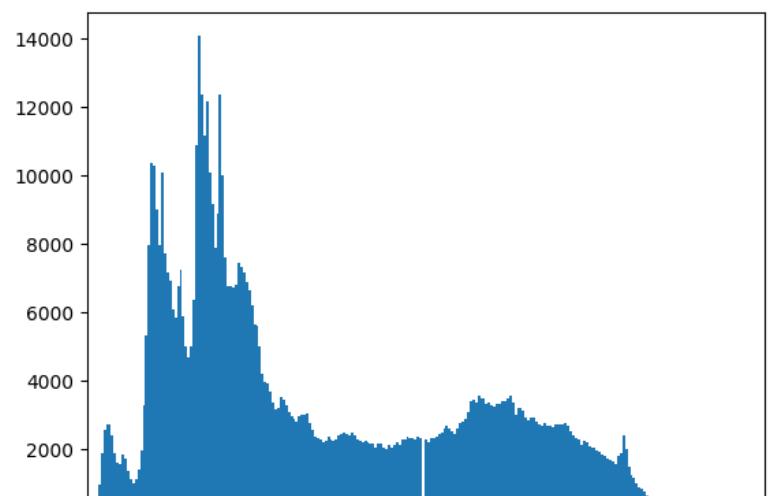
a)



б)



в)



г)

Рис. 9. Изображение 1 и 2 и их гистограммы.

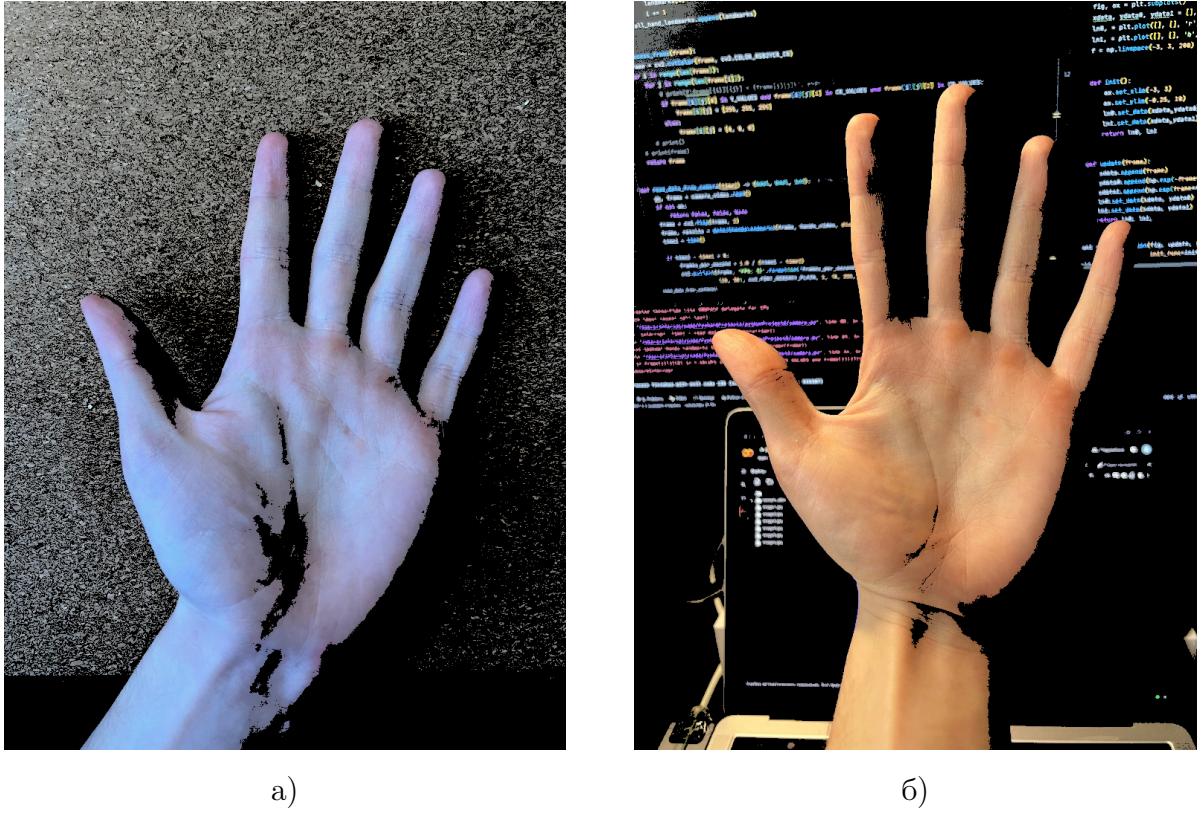


Рис. 10. Результат работы программы с алгоритмом Оцу

1.4 Рекурсивные методы построения модели фона

1.4.1 Visual Background Extractor.

Универсальным и наиболее совершенным на данный момент адаптивным методом, показывающим очень хорошие результаты практически для любых ситуаций и освещений, а также значительно выигрывающим в скорости работы по сравнению с другими алгоритмами, показывающими приблизительно такое же качество поиска движущихся объектов, считается *Visual Background Extractor* [6].

Классическим принципом работы большинства активно использующихся современных адаптивных методов является построение для каждого пикселя кадра функции плотности вероятности. ViBe основывается на принципиально другой идеи: к вопросу об интенсивности (в общем случае о цвете) пикселя $p = (x, y)$ подходят как к вопросу классификации. Для каждого пикселя сохраняется некоторое количество N его предыдущих значений $\nu(p)$, не обязательно последовательных (чаще всего полагают $N = 20$). Тогда можно сказать, что для каждого пикселя текущего кадра будет иметься своя модель C , которая представляется следующим множеством:

$$C(p) = \{\nu_1(p), \dots, \nu_N(p)\} \quad (27)$$

После этого осуществляется классификация пикселя с целью либо выявить движение, либо отнести данный пиксель к фону. В общем случае для этого в цветовом про-

пространстве для пикселя p , значение которого на текущем кадре обозначим через $\nu_n(p)$, где n – номер текущего кадра, строится сфера $S_R(\nu_n(p))$ заранее определённого радиуса R (рис. 11) и определяется количество значений K_n значений из $C_n(p)$, попадающих в эту сферу:

$$K_n(p) = |S_R(\nu_n(p)) \cap C_n(p)|. \quad (28)$$

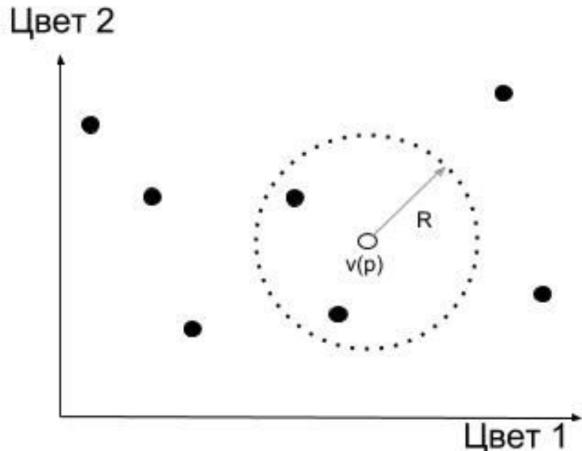


Рис. 11. Сфера в цветовом пространстве

В случае, когда изображения предварительно переводятся в градации серого, значения $\nu(p)$ представляют характеризующий интенсивность серого цвета скаляр, а не вектор, и для вычисления количества K_n "близких" значению $\nu_n(p)$ элементов из $C_n(p)$ необходимо просто осуществить следующую проверку для каждого из данных элементов:

$$|\nu(p) - \nu_n(p)| < R. \quad (29)$$

Решающее правило для текущего кадра строится следующим образом: если $K_n > K_{min}$, где K_{min} обычно принимается много меньше, чем $N/2$, пиксель p на текущем кадре принимается за фоновый, иначе предполагают, что он принадлежит движущемуся объекту. Стоит отметить, что в формулах (??), (29) можно осуществлять проверку не для всех элементов $C_n(p)$, а только до тех пор, пока не найдём K_{min} попадающих в сферу (в общем случае).

После этого для каждого пикселя необходимо обновить его модель. Этот процесс представляет из себя два последовательных шага:

1. Если p был отнесён к фону, из $C_n(p)$ случайным образом выбирается элемент, который будет заменён на $\nu_n(p)$.
2. Из окрестности $O(p)$ размера 3×3 (9 пикселей с учётом p) случайным образом выбирается пиксель, случайный элемент модели которого также будет заменён на $\nu_n(p)$.

Кроме самого процесса обновления модели фона отличительной особенностью ViBe также является процесс её инициализации. Для многих алгоритмов она производится

случайным образом, однако ViBe старается построить наиболее точную модель как можно раньше. С этой целью инициализация модели каждого пикселя производится так:

$$C_0(p) = \{\nu(z), z \in O(p)\}. \quad (30)$$

Здесь пиксель z выбирается N раз случайным образом. Такой процесс позволяет получить достаточно хорошую модель фона уже для второго кадра видео.

Несмотря на все достоинства и простоту реализации, достаточно часто приходится искать альтернативу данному алгоритму, так как он защищен множеством патентов.

1.4.2 Mixture of Gaussians.

Стандартным подходом к построению модели фона, использующимся для многих прикладных задач, является смесь гауссовых распределений (Mixture of Gaussians, MOG) [7, 8]. Как упоминалось выше, чаще всего для каждого пикселя текущего кадра с номером n строится функция плотности вероятности $P_n = P(\nu_n(p))$, и MOG используется именно этот подход. Предполагается, что для каждого пикселя текущего изображения она может быть представлена смесью нормальных распределений, где G – их число в смеси. Обозначим за w_i^n вес распределения Гаусса с номером i , E_i^n – его математическое ожидание и \sum_i^n – среднеквадратичное отклонение. Тогда для P_n получим:

$$P_n = \sum_{i=1}^G w_i^n \cdot N(\nu_n(p)|E_i^n, \sum_i^n). \quad (31)$$

Здесь в общем случае $N(\nu_n(p)|E_i^n, \sum_i^n)$ – многомерное нормальное распределение, имеющее вид:

$$N(\nu_n(p)|E_i^n, \sum_i^n) = \frac{1}{(2\pi)^{H/2} |\sum_i^n|^{1/2}} \exp\left[-\frac{1}{2}(\nu_n(p) - E_i^n)^T \cdot (\sum_i^n)^{-1} \cdot (\nu_n(p) - E_i^n)\right], \quad (32)$$

где

H – число компонентов цвета,

\sum – матрица ковариации.

Для ускорения вычислений, чтобы не вычислять обратную матрицу в формуле (32), в случае цветного изображения предполагается, что для компонентов цвета соблюдаются их независимость, а также что они имеют одинаковое стандартное отклонение. Тогда матрица ковариации примет вид:

$$\sum_i^n = (\sigma_i^n) \cdot E, \quad (33)$$

где E – единичная матрица размерности H .

После того, как для всех пикселей получена смесь, для каждой из них производится сортировка распределений по убыванию величины $\frac{w_i^n}{\sigma_i^n}$. Это делается для того, чтобы

фоновые пиксели отвечали распределению с маленькой дисперсией и большим весом. Тогда число J первых гауссиан, соответствующих распределению цвета фона для пикселя p , будет определяться из условия:

$$J = \arg \min_a \left(\sum_{i=0}^a w_i^n > A \right), \quad (34)$$

где A – некоторый порог.

Решающее правило для текущего кадра выглядит следующим образом: для каждого пикселя определяют, какому из распределений смеси принадлежит его значение $\nu_{n+1}(p)$, используя расстояние Махаланобиса:

$$\sqrt{(\nu_{n+1}(p) - E_i^n)^T \cdot \left(\sum_i^n \right)^{-1} \cdot (\nu_{n+1}(p) - E_i^n)} < 2.5 \cdot \sigma_i^n. \quad (35)$$

После этого возможно два случая:

1. Распределение нашлось. Пиксель будет отнесён к фону, если эта гауссиана является одной из J первых. В противном случае он будет отнесён к движущемуся объекту.
2. Если они одной гауссианы не нашлось, то пиксель также относят к движущемуся объекту.

Теперь рассмотрим, как для пикселя p со значением $\nu_{n+1}(p)$ происходит обновление параметров его модели. Как и при принятии решения, относится ли пиксель к фону или нет, здесь также можно быть два случая. Не существует единого понимания того, каким образом следует вычислять новые параметры, поэтому рассмотрим здесь один из наиболее популярных:

1. Распределение нашлось. Происходит обновление его параметров и веса. Если нашлось более одного распределения, данный процесс выполняется для каждого из них:

$$\begin{aligned} w_i^{n+1} &= (1 - \alpha) \cdot w_i^n + \alpha, \\ E_i^{n+1} &= (1 - g) \cdot E_i^n + g \cdot \nu_{n+1}(p), \\ (\sigma_i^{n+1})^2 &= (1 - g) \cdot (\sigma_i^n)^2 + g \cdot (\nu_{n+1}(p) - E_i^{n+1}) \cdot (\nu_{n+1}(p) - E_i^{n+1})^T, \\ g &= \alpha \cdot N(\nu_n(p) | E_i^n, \sum_i^n). \end{aligned}$$

В этих формулах α – заданная константа.

Для всех остальных распределений в смеси пересчитываются только веса:

$$w_i^{n+1} = (1 - \alpha) \cdot w_i^n.$$

2. Распределение в смеси не нашлось. Тогда самую последнюю в уже отсортированной смеси гауссиану заменяют новой: $E_G^{n+1} = \nu_{n+1}(p)$, дисперсия – максимально возможная, а вес – минимально допустимый.

Для инициализации гауссиан для каждого пикселя чаще всего применяют либо EM-алгоритм (Expectation-maximization algorithm), либо k-means, что достаточно затратно в вычислительном плане. Число входящих в смесь распределений G обычно принимают равным от 3 до 5. Также существует подход, позволяющий автоматически подбирать необходимое количество гауссиан [9].

Рассмотрев наиболее популярные методы отделения кисти человека от фона изображения, можно сделать вывод, что некоторые из методов работают наилучшим образом при специфических условиях, поэтому необходимо их дополнительное исследование.

2 Методы построения контура объекта на изображении

Отслеживание границ – один из основных методов обработки оцифрованных двоичных изображений. Он производит последовательность координат или цепных кодов от границ между связным компонентом

2.1 Выделение границ с помощью оператора Кэнни

Оператор Кэнни (детектор границ Кэнни или алгоритм Кэнни) – оператор обнаружения границ изображения. Кэнни изучил математическую проблему получения фильтра, оптимального по критериям выделения, локализации и минимизации нескольких откликов одного края. Он показал, что искомый фильтр является суммой четырёх экспонент. Он также показал, что этот фильтр может быть хорошо приближен первой производной Гауссианы. Кэнни ввёл понятие *подавление немаксимумов* (Non-Maximum Suppression), которое означает, что пикселями границ объявляются пиксели, в которых достигается локальный максимум градиента в направлении вектора градиента.

Целью Кэнни было разработать оптимальный алгоритм обнаружения границ, удовлетворяющий трём критериям:

- хорошее обнаружение (повышение отношения сигнал/шум);
- хорошая локализация (правильное определение положения границы);
- единственный отклик на одну границу.

Из этих критериев затем строится целевая функция стоимости ошибок, минимизация которой находится "оптимальный" линейный оператор для свёртки с изображением.

Первым этапом алгоритма является *сглаживание*, то есть размытие изображения для удаления шума. Оператор Кэнни использует фильтр, который может быть хорошо приближен к первой производной гауссианы. Для $\sigma = 1.4$:

$$B = \frac{1}{159} \begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix} \cdot A. \quad (36)$$

Следующим этапом является *поиск градиентов*. Границы отмечаются там, где градиент изображения приобретает максимальное значение. Они могут иметь различное направление, поэтому алгоритм Кэнни использует четыре фильтра для обнаружения горизонтальных, вертикальных и диагональных ребер в размытом изображении.

$$G = \sqrt{G_x^2 + G_y^2}, \Theta = \arctan\left(\frac{G_y}{G_x}\right). \quad (37)$$

Угол направления вектора градиента при этом округляется и может принимать такие значения: 0° , 45° , 90° , 135° .

Затем происходит *подавление немаксимумов*, когда только локальные максимумы отмечаются как границы, *двойная пороговая фильтрация* – потенциальные границы определяются порогами и *трассировка области неоднозначности*, когда итоговые границы определяются путём подавления всех краёв, не связанных с определёнными (сильными) границами.

Перед применением детектора обычно преобразуют изображение в оттенки серого, чтобы уменьшить вычислительные затраты.

С помощью следующего кода построим примеры работы алгоритма Кэнни:

```
1 import cv2
2 from matplotlib import pyplot as plt
3
4 CV2_CVT = [cv2.COLOR_BGR2RGB, cv2.COLOR_BGR2GRAY,
5             cv2.COLOR_BGR2HSV, cv2.COLOR_BGR2YCR_CB]
6 TITLES = ['RGB', 'GRAY', 'HSV', 'YCbCr']
7 def plot_images(img_name):
8     fig, axes = plt.subplots(2, 2)
9     fig.set_figwidth(10)
10    fig.set_figheight(10)
11    axes = axes.flatten()
12    img = cv2.imread(img_name)
13    img.blur = cv2.GaussianBlur(img, (5, 5), 2)
14    for i, ax in enumerate(axes):
15        ax.imshow(canny_wrapper(img.blur, CV2_CVT[i]),
16                   cmap='gray')
17        ax.set_title(TITLES[i])
18    for ax in axes:
19        ax.set_xticks([])
20        ax.set_yticks([])
21    plt.show()
22    fig.savefig(f'canny_image1.png')
23 plot_image('image1.jpg')
```

Пример работы оператора в цветовых моделях RGB, GRAY, HSV и YCbCr показан на рис. 12.

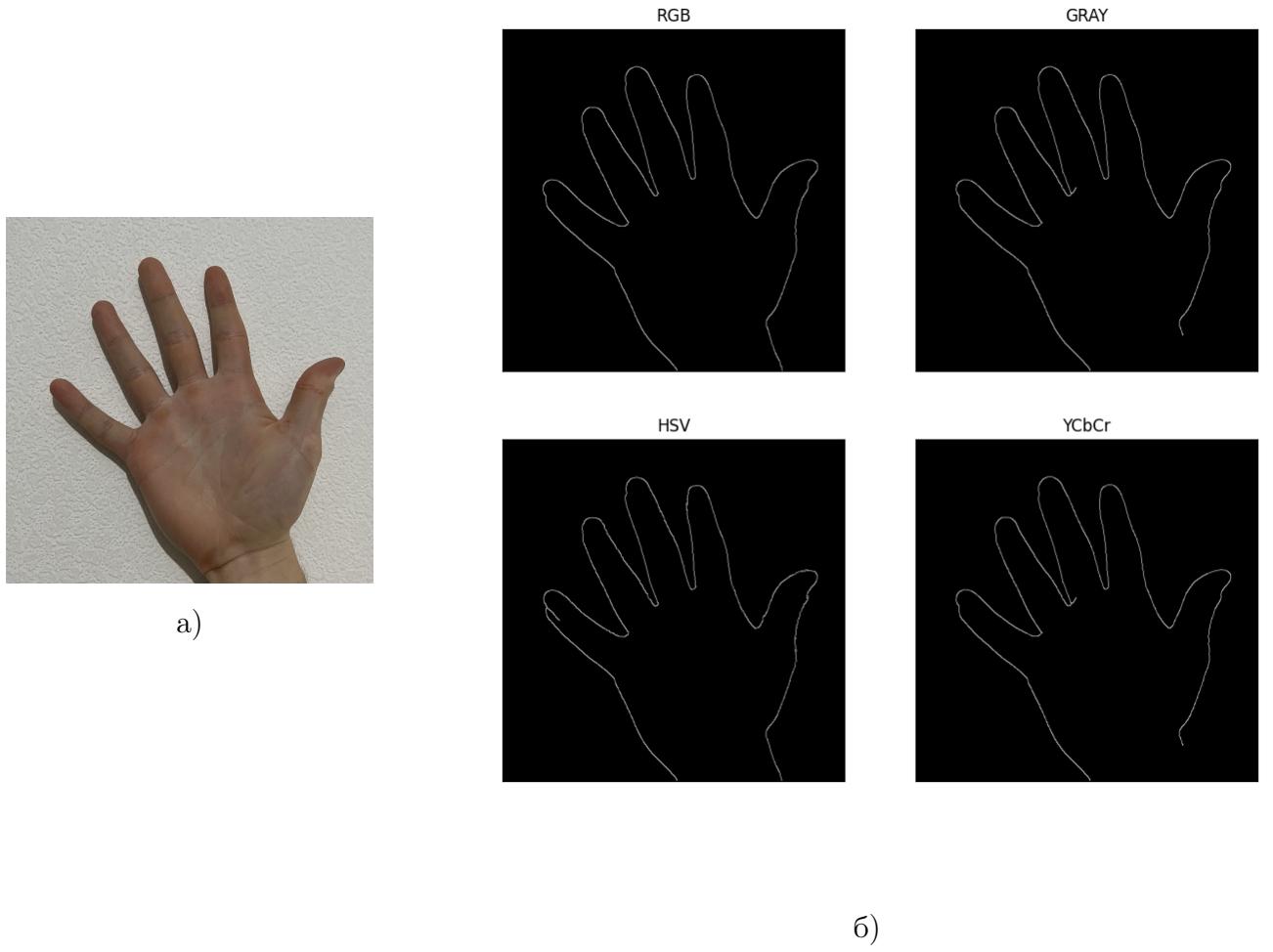


Рис. 12. Пример 1 работы оператора Кэнни.

Можно видеть, что алгоритм Кэнни довольно неплохо позволяет определить границы объекта. Но для задачи извлечения конфигурации кисти необходимо извлекать особые точки изображения, а этот метод лишь выделяет границы на изображении, не определяя сам контур. В связи с этим необходимо рассмотреть топологический структурный анализ цифрового бинарного изображения с помощью отслеживания границ.

2.2 Топологический структурный анализ цифрового бинарного изображения с помощью отслеживания границ.

Этот метод был разработан Сатоши Сузуки и Кейчи Эйбом в 1985 году [10]. Алгоритм предполагает нахождение контуров с учетом вложенности, то есть способен определить, когда в контур одного объекта вложен другой. Реализация данного алгоритма лежит в основе функции `findContours()` в библиотеке OpenCV, предназначеннной для

исследования и решения задач, связанных с компьютерным зрением.

2.2.1 Обзор функции `findContours()`.

Сигнатура функции `findContours()`, реализованной в библиотеке OpenCV для языка Python выглядит следующим образом:

```
1 def findContours(image, mode, method, contours=None,
2                   hierarchy=None, offset=None):
3     pass
```

Опишем каждый из параметров:

- **image** – Исходное 8-битное бинарное изображение. Для преобразование можно использовать функции `inRange()`, `threshold()`, `adaptiveThreshold()` и ранее использованная функция, реализующая алгоритм Кэнни – `Canny()`. Если `mode` является `RETR_CCOMP` или `RETR_FLOODFILL`, то изображение может быть 32-битным.
- **mode** – Режим поиска контуров. Может принимать следующие значения:
 - `RETR_EXTERNAL` – поиск только внешних контуров.
 - `RETR_LIST` – поиск всех контуров без установки их отношения.
 - `RETR_CCOMP` – поиск всех контуров и организация их в иерархию, состоящую из двух уровней: на верхнем уровне находится внешние границы компонент, а на втором – границ "дыр". Если внутри "дыры" есть другой контур, то он устанавливается на верхнем уровне.
 - `RETR_TREE` – поиск всех контуров и установка их иерархии вложенных контуров.
 - `RETR_FLOODFILL`
- **method** – Метод аппроксимации контуров. Может принимать следующие значения:
 - `CHAIN_APPROX_NONE` – без аппроксимации, хранятся абсолютно все точки контура. Это означает, что две последовательные точки (x_1, y_1) и (x_2, y_2) контура будут либо горизонтальными, либо вертикальными, либо диагональными соседями, то есть $\max(|x_1 - x_2|, |y_1 - y_2|) = 1$.
 - `CHAIN_APPROX_SIMPLE` – сжатие горизонтальных, вертикальных и диагональных сегментов, и хранение только их точек концов. Например, контур в виде прямоугольника будет описан только его четырьмя точками (вершинами).
 - `CHAIN_APPROX_TC89_L1` / `CHAIN_APPROX_TC89_KCOS` – применение двух подходов алгоритма Тена и Чина аппроксимации цепи [11].

- **contours** – Найденные контуры.
- **hierarchy** – (*опционально*) Вектор, хранящий информацию о топологии изображения. Количество элементов вектора равно количеству найденных контуров. Для каждого i -того контура **contours**[i] элемент:
 - **hierarchy**[i][0] = индексу следующего контура на текущем уровне,
 - **hierarchy**[i][1] = индексу предыдущего контура на текущем уровне,
 - **hierarchy**[i][2] = индексу первого контура на вложенном уровне,
 - **hierarchy**[i][3] = индексу родительского контура.

Если для i -того контура нет следующего, предыдущего, родительского и вложенного, то соответствующие элементы являются отрицательными.

- **offset** – (*опционально*) Величина сдвига каждой точки контура. Полезно, если контуры вычисляются из ROI (англ. Region Of Interest, область интереса) изображения и точки контура должны анализироваться в отношении к всему изображению.

Для отрисовки контуров, полученных с помощью данной функции, удобно пользоваться функцией **drawContours()**. Рассмотрим её поподробнее.

2.2.2 Обзор функции **drawContours()**.

Сигнатура функции **findContours()**, реализованной в библиотеке OpenCV для языка Python выглядит следующим образом:

```

1 def drawContours(image, contours, contourIdx, color, thickness=None,
2                  lineType=None, hierarchy=None, maxLevel=None, offset=None):
3     pass

```

Параметры:

- **image** – Изображение, на котором будут отрисовываться контуры.
- **contours** – Все контуры.
- **contourIdx** – Параметр, обозначающий индекс контура для отрисовки. Если отрицателен, то будут отрисованы все контуры.
- **color** – Цвет контуров.
- **thickness** – Толщина контуров. Если отрицателен, то контуры будут закрашены полностью вместе с внутренним пространством.
- **lineType** – Тип соединения линий (FILLED, LINE_4, LINE_8, LINE_AA).
- **hierarchy** – (*опционально*) Информация о иерархии контуров. Необходим, если нужно отрисовать только несколько контуров (см. **maxLevel**).
- **maxLevel** – Максимальный уровень контуров для отрисовки. Если **maxLevel** = – 0, то будут отрисованы только специфичные контуры,

- 1, то будут отрисованы все контур(ы) и все в них вложенные,
- 2, то функция отрисовывает все контуры, все в них вложенные, все вложенные во вложенные контуры, и так далее.

Этот параметр игнорируется, если не задан параметр `hierarchy`, то есть `hierarchy = None`.

- `offset` – (опционально) Сдвиг контуров для отрисовки: `offset = (dx, dy)`

Поскольку функция `findContours()` находит все контуры на изображении, то нужно среди них выбрать один единственный. Пусть кисть занимает наибольшее пространство на изображении и, соответственно, имеет наибольший контур. Среди всех контуров будем отбирать тот, *площадь* которого имеет наибольшее значение. Площадь контура будем находить с помощью *формулы площади Гаусса*.

2.2.3 Формула площади Гаусса.

Формула площади Гаусса (формула землемера или формула шнурования или алгоритм шнурования) – формула определения площади простого многоугольника, вершины которого заданы декартовыми координатами на плоскости. В формуле векторным произведением координат и сложением определяется площадь области, охватывающей многоугольник, а затем из нее вычитается площадь окружающего многоугольника, что дает площадь многоугольника внутри.

Формула может быть представлена следующим выражением:

$$\begin{aligned} S &= \frac{1}{2} \left| \sum_{i=1}^{n-1} x_i y_{i+1} + x_n y_1 - \sum_{i=1}^{n-1} x_{i+1} y_i - x_1 y_n \right| = \\ &= \frac{1}{2} |x_1 y_2 + x_2 y_3 + \cdots + x_{n-1} y_n + x_n y_1 - x_2 y_1 - x_3 y_2 - \cdots - x_n y_{n-1} - x_1 y_n|, \end{aligned} \quad (38)$$

где

S – площадь многоугольника,

n – количество сторон многоугольника,

$(x_i, y_i), i = \overline{1, n}$ – координаты вершин многоугольника.

Другие представления этой же формулы:

$$\begin{aligned} S &= \frac{1}{2} \left| \sum_{i=1}^n x_i (y_{i+1} - y_{i-1}) \right| = \frac{1}{2} \left| \sum_{i=1}^n y_i (x_{i+1} - x_{i-1}) \right| = \\ &= \frac{1}{2} \left| \sum_{i=1}^n x_i y_{i+1} - x_{i+1} y_i \right| = \frac{1}{2} \left| \sum_{i=1}^n \det \begin{pmatrix} x_i & y_i \\ x_{i+1} & y_{i+1} \end{pmatrix} \right|, \end{aligned} \quad (39)$$

где

$x_{n+1} = x_1, x_0 = x_n$,

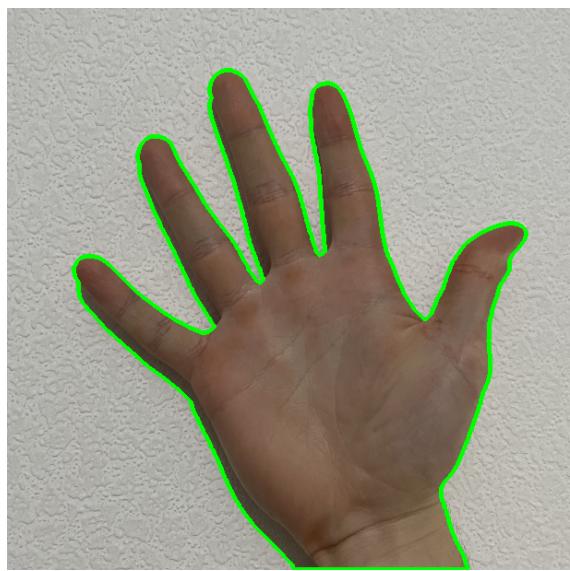
$y_{n+1} = y_1, y_0 = y_n$.

Рассмотрим пример использования данных функций. Сравним результат поиска контуров с предварительной обработкой изображения, заключающейся в отделении объекта от фона, и без обработки (только с переводом изображения в черно-белый формат). С помощью следующего кода получим результаты, представленные на рис. 13.

```
1 import cv2
2
3 def process_image(image):
4     blurred = cv2.GaussianBlur(image, (3, 3), 3)
5     _, thresh1 = cv2.threshold(blurred, 0, 255,
6                               cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
7     return thresh1
8
9 def find_contours_otsu(filename: str, with_processing: bool = False):
10    img = cv2.imread(filename)
11    img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
12    image_to_contours = process_image(img_gray) if with_processing else img_gray
13    contours, hierarchy = cv2.findContours(image_to_contours.copy(),
14                                           cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
15    print(f'Contours count = {len(contours)}')
16    # находим контур с максимальной площадью
17    cnt = max(contours, key=lambda x: cv2.contourArea(x))
18    print(f'Contour area = {cv2.contourArea(cnt)}')
19    print(f'Points of contour count = {len(cnt)}')
20    cv2.drawContours(img, [cnt], -1, (0, 255, 0), 3)
21    cv2.imwrite('contours_' +
22                ('processing_' if with_processing else '') +
23                'example.png', img)
24 find_contours_otsu('image1.jpg')
25 find_contours_otsu('image1.jpg', with_processing=True)
```



а)



б)

Рис. 13. Пример поиска контура на изображении без предварительной обработки (а) и с предварительной обработкой с помощью метода Оцу (б).

Можно видеть, что пороговая бинаризация изображения методом Оцу позволила с высокой точностью определить истинное расположение контура ладони. Это означает, что для дальнейшего исследования контуров необходима предобработка изображения.

3 Методы нахождения ключевых точек на кисти.

Особую сложность представляет задача определения положения ключевых точек на кисти человека. Для этой нетривиальной задачи было предложено несколько методов её решения. Рассмотрим каждый из них.

3.1 Нахождение точек путём определения дефектов выпуклости.

В предыдущем разделе был рассмотрен метод построения контура кисти с помощью топологического структурного анализа цифрового бинарного изображения с помощью отслеживания границ. Этот метод хорош тем, что позволяет определить точные координаты точек контура. С помощью этих точек существует возможность построить *выпуклую оболочку* или их *наименьшее множество*.

Выпуклой оболочкой множества X называется наименьшее выпуклое множество, содержащее X . «Наименьшее множество» здесь означает наименьший элемент по отношению к вложению множеств, то есть такое выпуклое множество, содержащее данную фигуру, что оно содержится в любом другом выпуклом множестве, содержащем данную фигуру.

Рассмотрим основные алгоритмы построения выпуклой оболочки.

3.1.1 Алгоритм Грэхема.

Алгоритм Грэхема [12] — алгоритм построения выпуклой оболочки в двумерном пространстве. В этом алгоритме задача о выпуклой оболочке решается с помощью стека, сформированного из точек-кандидатов. Все точки входного множества заносятся в стек, а потом точки, не являющиеся вершинами выпуклой оболочки, со временем удаляются из него. По завершении работы алгоритма в стеке остаются только вершины оболочки в порядке их обхода против часовой стрелки.

Временная сложность алгоритма = $O(n \log n)$.

Описание алгоритма:

Пусть точки $p = [p_0, \dots p_{n-1}]$ – входной массив точек.

1. Найти самую "нижнюю" точку в массиве (ту, в которой наименьшая среди всех координата y). Если таких точек несколько, то среди них выбрать точку с наименьшей координатой x . Найденная точка P_0 является первой точкой выпуклой оболочки;
2. Перебрать остальные $n - 1$ точек и отсортировать их по полярному углу относительно P_0 в направлении против часовой стрелки. Если полярный угол нескольких точек одинаков, то выбрать ближайшую к P_0 ;

3. После сортировки, проверить, есть ли точки с одинаковым полярным углом. Если да, то удалить все эти точки, кроме ближайшей к P_0 . Положить размер нового массива равным m ;
4. Если $m < 3$, то алгоритм прерывается. Выпуклую оболочку определить невозможно;
5. Создать пустой стек S и положить в него первые три точки нового массива p_0, p_1, p_2 ;
6. Для каждой из оставшихся $m - 3$ точек:
 - (a) Удаляем точки из стека пока ориентация трёх следующих точек не против часовой стрелки (они не совершают левый поворот): точка наверху стека, точка следующая после неё и текущая точка p_i ;
 - (b) Добавляем точку p_i в S ;
7. Стек S содержит точки выпуклой оболочки.

Реализация алгоритма Грэхема на языке Python находится в приложении. С помощью следующего кода протестируем алгоритм на случайном наборе точек. Результат работы показан на рисунке 14.

```

1 import numpy as np
2 import random
3 from matplotlib import pyplot as plt
4 import cv2
5
6 from my_convex_hulls.convexHullGraham import convex_hull \
7     as convex_hull_graham
8
9 def get_random_point(height, width) -> tuple:
10     return random.randint(width // 3, width), \
11             random.randint(height // 3, height)
12
13 def plot_images(img1, img2):
14     fig, axes = plt.subplots(1, 2)
15     fig.set_figwidth(10)
16     fig.set_figheight(5)
17     axes = axes.flatten()
18     axes[0].imshow(img1)
19     axes[1].imshow(img2)
20     for ax in axes:
21         ax.set_xticks([])
22         ax.set_yticks([])
```

```

23     plt.show()
24     fig.savefig(f'graham_test.png')
25
26 def draw_convex_hull(img_init, points):
27     img = img_init.copy()
28     for i in range(len(points) - 1):
29         cv2.line(img, points[i],
30                  points[i + 1], (0, 255, 0), 2)
31     cv2.line(img, points[0],
32             points[-1], (0, 255, 0), 2)
33     return img
34
35 if __name__ == '__main__':
36     width, height = 500, 500
37     num_points = 20
38     img = np.zeros((width, height, 3))
39     points = [get_random_point(width * 4 // 5, height * 4 // 5)
40               for _ in range(num_points)]
41     for point in points:
42         cv2.circle(img, point, 5, (255, 0, 0), -1)
43     convex_hull_points = convex_hull_graham(points)
44     img_copy = draw_convex_hull(img, convex_hull_points)
45     plot_images(img, img_copy)

```

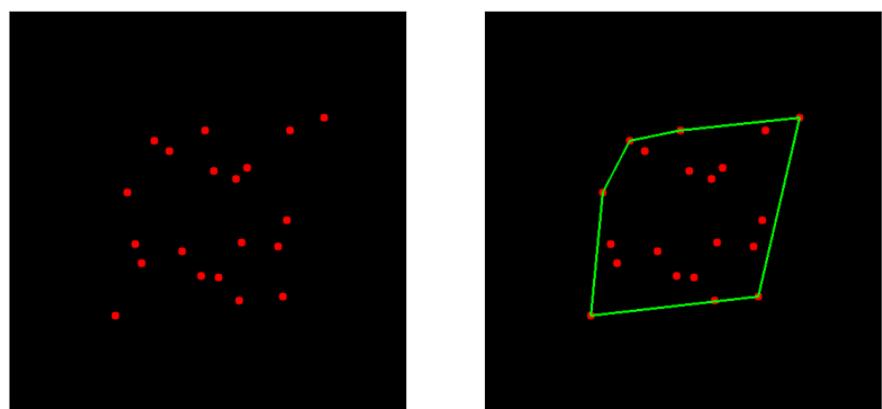


Рис. 14. Сгенерированные точки (слева) и их выпуклая оболочка, найденная с помощью алгоритма Грэхема (справа).

3.1.2 Алгоритм Джарвиса.

Алгоритм Джарвиса [13] (или алгоритм обхода Джарвиса, или алгоритм заворачивания подарка) определяет последовательность элементов множества, образующих выпуклую оболочку для этого множества. Метод можно представить как обтягивание верёвкой множества вбитых в доску гвоздей. Алгоритм работает за время $O(nh)$, где n – общее число точек на плоскости, а h – число точек в выпуклой оболочке. В худшем случае – $O(n^2)$, когда все точки попадают в выпуклую оболочку.

Описание алгоритма:

Пусть дано множество точек $P = \{p_1, p_2, \dots, p_n\}$. В качестве начальной берётся самая левая нижняя точка p_1 (её можно найти за $O(n)$ обычным проходом по всем точкам), она точно является вершиной выпуклой оболочки. Следующей точкой (p_2) берём такую точку, которая имеет наименьший положительный полярный угол относительно точки p_1 как начала координат. После этого для каждой точки p_i ($2 < i \leq |P|$) против часовой стрелки ищется такая точка p_{i+1} , путём нахождения за $O(n)$ среди оставшихся точек (+ самая левая нижняя), в которой будет образовываться наибольший угол между прямыми $p_{i-1}p_i$ и p_ip_{i+1} . Она и будет следующей вершиной выпуклой оболочки. Сам угол при этом не ищется, а ищется только его косинус через скалярное произведение между лучами $p_{i-1}p_i$ и $p_ip'_{i+1}$, где p_i – последний найденный минимум, p_{i-1} – предыдущий минимум, а p'_{i+1} – претендент на следующий минимум. Новым минимумом будет та точка, в которой косинус будет принимать наименьшее значение (чем меньше косинус, тем больше его угол). Нахождение вершин выпуклой оболочки продолжается до тех пор, пока $p_{i+1} \neq p_1$. В тот момент, когда следующая точка в выпуклой оболочке совпала с первой, алгоритм останавливается — выпуклая оболочка построена (рис. 15).

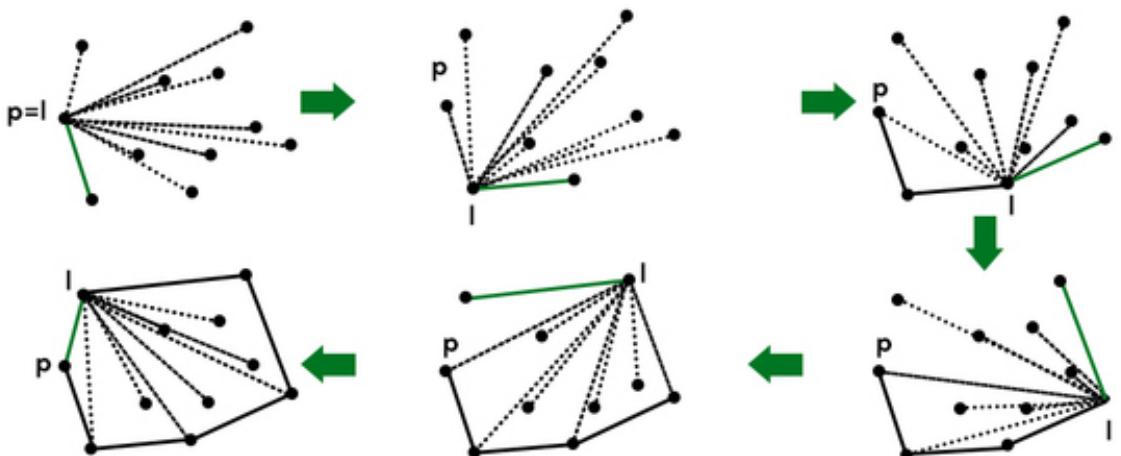


Рис. 15. Пример работы алгоритма Джарвиса.

Реализация алгоритма Джарвиса на языке Python находится в приложении. С помощью того же кода, но с заменой функции `convex_hull`, протестируем алгоритм на

случайном наборе точек. Результат работы показан на рисунке 16.

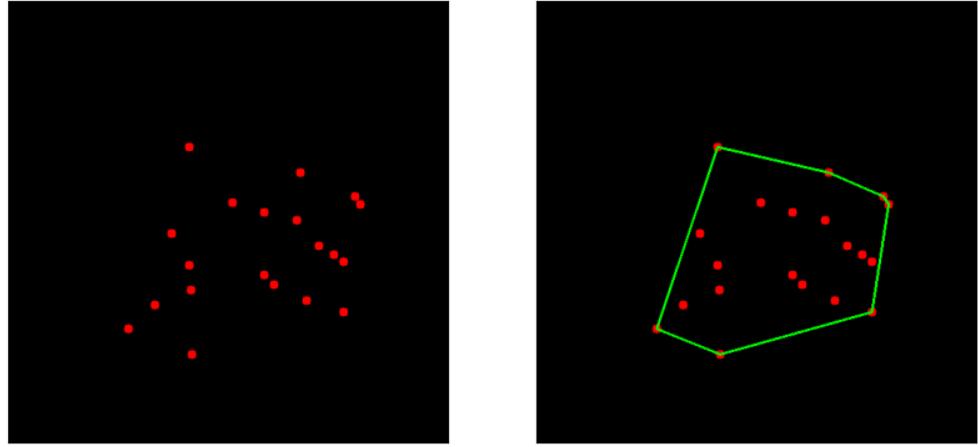


Рис. 16. Сгенерированные точки (слева) и их выпуклая оболочка, найденная с помощью алгоритма Джарвиса (справа).

Можно видеть, что алгоритм отработал верно, но у него есть один существенный недостаток – он намного медленнее по сравнению с алгоритмом Грэхема.

3.1.3 Алгоритм Киркпатрика.

Алгоритм Киркпатрика [14] заключается в построении выпуклой оболочки методом "разделяй и властвуй".

Описание алгоритма:

Дано множество S , состоящее из N точек.

1. Если $N \leq N_0$ (N_0 – некоторое небольшое целое число), то построить выпуклую оболочку одним из известных методов и остановиться, иначе перейти к шагу 2.
2. Разобьём исходное множество S произвольным образом на два примерно равных по мощности подмножества S_1 и S_2 (пусть S_1 содержит $N/2$ точек, а S_2 содержит $N - N/2$ точек).
3. Рекурсивно находим выпуклые оболочки каждого из подмножеств S_1 и S_2 .
4. Строим выпуклую оболочку исходного множества как выпуклую оболочку объединения двух выпуклых многоугольников $CH(S_1)$ и $CH(S_2)$.

Поскольку $CH(S) = CH(S_1 \cup S_2) = CH(CH(S_1) \cup CH(S_2))$, сложность этого алгоритма является решением рекурсивного соотношения $T(N) \leq 2T(N/2) + f(N)$, где $f(N)$ – время построения выпуклой оболочки объединения двух выпуклых многоугольников, каждый из которых имеет около $N/2$ вершин. Таким образом, временная сложность этого алгоритма равна $O(N \log N)$.

Реализация алгоритма Киркпатрика на языке Python находится в приложении. Результат работы показан на рисунке 17.

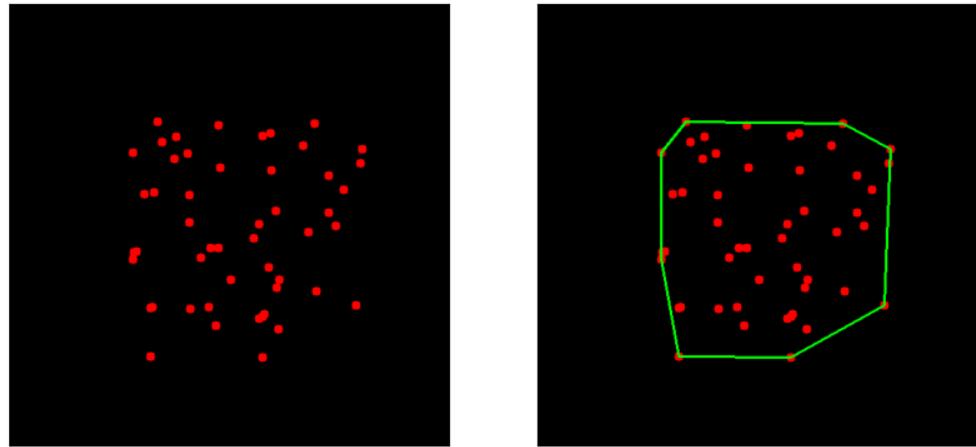


Рис. 17. Сгенерированные точки (слева) и их выпуклая оболочка, найденная с помощью алгоритма Киркпатрика (справа).

3.1.4 Сравнение алгоритмов.

Для того чтобы выбрать наиболее быстрый алгоритм, проведём их сравнение по времени работы на 1000 тестах с помощью следующего кода:

```
1 # imports
2 def generate_random_points(width, height, size=0, is_random=False):
3     if is_random or size == 0:
4         size = random.randint(100, 500)
5     return [get_random_point(width * 4 // 5, height * 4 // 5)
6             for _ in range(size)]
7
8 def get_random_point(height, width) -> tuple:
9     return random.randint(width // 3, width), \
10           random.randint(height // 3, height)
11
12 def measure_time(method: function, args) -> int:
13     time_start = time.time_ns()
14     method(args)
15     return time.time_ns() - time_start
```

```

16
17 GRAHAM, JARVIS, KPATRICK = 'GRAHAM', 'JARVIS', 'KIRKPATRICK'
18 NAME_TO_ALGO = {GRAHAM: convex_hull_graham,
19             JARVIS: convex_hull_jarvis,
20             KPATRICK: convex_hull_kirkpatrick}
21 width, height, num_tests = 5000, 5000, 1000
22 TIMES = {GRAHAM: [], JARVIS: [], KPATRICK: []}
23 for _ in range(num_tests):
24     points = generate_random_points(width, height, is_random=True)
25     for algo in TIMES.keys():
26         TIMES[algo].append(measure_time(NAME_TO_ALGO[algo], points))
27 for algo, times in TIMES.items():
28     print(f"-- {algo} --")
29     print(f'<Max>: {max(times) // 1000} ms')
30     print(f'<Min>: {min(times) // 1000} ms')
31     print(f'<Mean>: {np.array(times).mean() // 1000} ms')

```

Результаты работы сведены в таблицу 2.

Алгоритм	Максимальное время, мс	Минимальное время, мс	Среднее время, мс
Грэхема	5317	581	2193.0
Джарвиса	187204	5979	66401.0
Киркпатрика	23102	2008	6944.0

Таблица 2. Сравнение алгоритмов построения выпуклой оболочки.

Можно видеть, что самым эффективным в данном случае оказался алгоритм Грэхема, именно его и будем использовать.

Алгоритм Грэхема уже реализован в библиотеке OpenCV в виде функции `convexHull()`. Рассмотрим её подробнее.

3.1.5 Обзор функции `convexHull()`

Сигнатура функции `convexHull()`, реализованной в библиотеке OpenCV для языка Python выглядит следующим образом:

```

1 def convexHull(points, hull=None, clockwise=None, returnPoints=None):
2     pass

```

Параметры:

- `points` – Входные точки;
- `hull` – Выходной массив с точками выпуклой оболочки;
- `clockwise` – Флаг ориентации. Если равен `True`, то точки `hull` ориентированы по часовой стрелке;
- `returnPoints` – Если равен `True`, то возвращает точки выпуклой оболочки, иначе – индексы из `points`.

С помощью следующего кода построим контур и выпуклую оболочку кисти руки. Результат работы программы изображён на рисунке 18.

```

1 import cv2
2
3 img = cv2.imread('image1.jpg')
4 img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
5 blurred = cv2.GaussianBlur(img_gray, (3, 3), 3)
6 _, image_to_contours = cv2.threshold(blurred, 0, 255,
7                                     cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
8 contours, hierarchy = cv2.findContours(image_to_contours.copy(),
9                                         cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
10 cnt = max(contours, key=lambda x: cv2.contourArea(x))
11 cv2.drawContours(img, [cnt], -1, (0, 255, 0), 2)
12 hull = cv2.convexHull(cnt)
13 cv2.drawContours(img, [hull], -1, (255, 0, 0), 2)
14 cv2.imwrite('convex_hull_example.png', img)

```

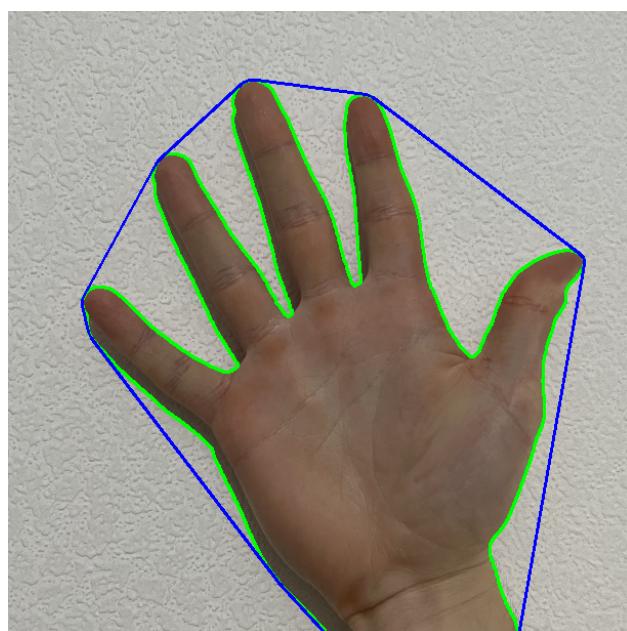


Рис. 18. Выделение контура и выпуклой его оболочки на кисти.

Для определения необходимых ключевых точек на кисти необходимо найти *дефекты выпуклости* контура и выпуклой оболочки.

Дефект выпуклости в данном случае – это максимальное расстояние между выпуклой оболочкой и контуром (рис. 19).

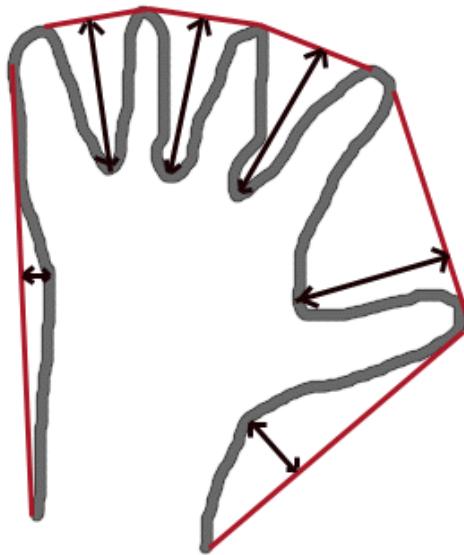


Рис. 19. Дефекты выпуклости.

В библиотеке OpenCV уже существует реализация расчётов дефектов выпуклости в виде функции `convexityDefects()`.

3.1.6 Обзор функции `convexityDefects()`.

Сигнатура функции в Python выглядит следующим образом:

```
1 def convexityDefects(contour, convexhull, convexityDefects=None):  
2     pass
```

Параметры довольно простые:

- `contour` – массив точек контура;
- `convexhull` – массив индексов выпуклой оболочки, где индексы берутся из `contour`;
- `convexityDefects` – массив дефектов выпуклости, в каждой строке которого находятся 4 элемента:
 - индекс начала дефекта;
 - индекс конца дефекта;
 - самая дальняя точка;
 - расстояние до контура;

Запустив следующий код Python, получим результат вычисления, представленный на рисунке 20.

```
1 import cv2
2
3 def process_image(image):
4     return cv2.threshold(cv2.GaussianBlur(image, (3, 3), 3),
5                         0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)[1]
6
7 img = cv2.imread('image1.jpg')
8 img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
9 image_to_contours = process_image(img_gray)
10 contours, hierarchy = cv2.findContours(image_to_contours.copy(),
11                                         cv2.RETR_TREE, cv2.CHAIN_APPROX_NONE)
12 cnt = max(contours, key=lambda x: cv2.contourArea(x))
13 cv2.drawContours(img, [cnt], -1, (0, 255, 0), 2)
14 hull = cv2.convexHull(cnt)
15 cv2.drawContours(img, [hull], -1, (255, 0, 0), 2)
16 hull_index = cv2.convexHull(cnt, returnPoints=False)
17 defects = cv2.convexityDefects(cnt, hull_index)
18 for defect in defects:
19     cv2.circle(img, cnt[defect[0][2]][0], 5, (0, 0, 255), -1)
20 cv2.imwrite('defects_example.png', img)
```

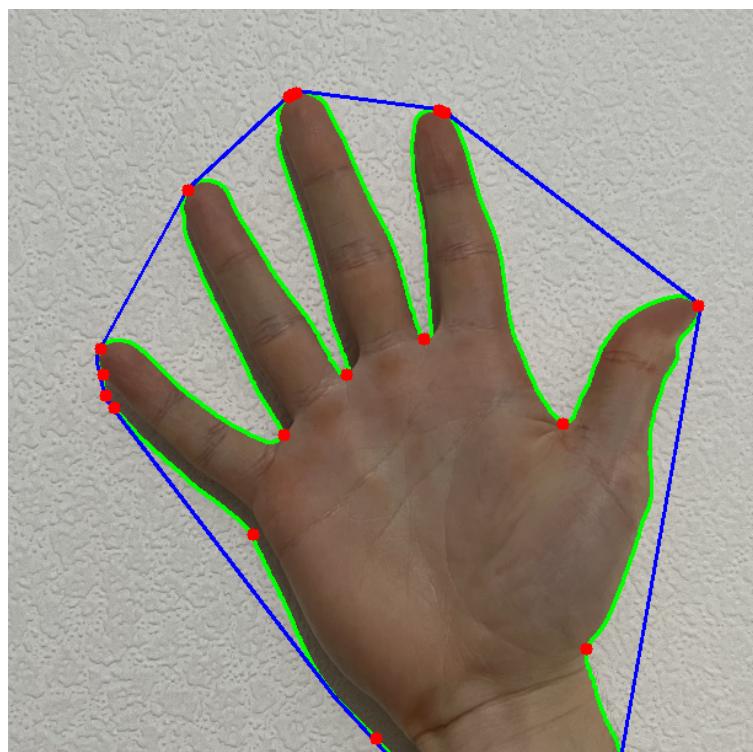


Рис. 20. Пример вычисления точек дефектов выпуклости (красные).

Можно видеть, что некоторые точки очень близко находятся друг к другу. Это связано с тем, что выпуклая оболочка касается контура несколько раз в почти одном и том же месте. Чтобы избавиться от этого напишем следующую функцию очистки этих точек:

```
1 import math
2 import numpy as np
3
4 def squeeze(array):
5     array = array[0] if len(array) == 1 else array
6     return [p[0] for p in array] if len(array[0]) == 1 else array
7
8 # "очистка" выпуклой оболочки до не больше 7 точек
9 def clear_convex_hull(hull_index, contour):
10    distance_point = lambda p1, p2: \
11        math.sqrt((p1[0] - p2[0]) ** 2 + (p1[1] - p2[1]) ** 2)
12    hull_index, contour = squeeze(hull_index), squeeze(contour)
13    ALPHA = 10
14    while True:
15        boundary = len(hull_index) - 1
16        clean_hull, i = [], 0
17        while i < boundary:
18            clean_hull.append(hull_index[i])
19            while i < boundary and \
20                distance_point(contour[hull_index[i]], \
21                               contour[hull_index[i + 1]]) < ALPHA:
22                i += 1
23            i += 1
24        if len(clean_hull) > 7:
25            ALPHA += 1
26        else:
27            break
28    return np.array(clean_hull[:-1]) if \
29        distance_point(contour[clean_hull[0]], \
30        contour[clean_hull[-1]]) < ALPHA else np.array(clean_hull)
```

Функция работает следующим образом. Берётся первая попавшаяся точка дефекта выпуклости, затем перебираются оставшиеся, пока они попадают в круг радиусом **ALPHA**.

с центром в первой точке. Эти точки не попадают в итоговый результат. Затем берётся следующая точка и т.д. Если после прохождения всех точек их количество больше 7 (2 точки снизу руки + максимум 5 точек-“пальцев”), то **ALPHA** инкрементируется.

Результаты работы программы поиска дефектов выпуклости с очисткой “ненужных” точек выпуклости можно видеть на рисунке 21.

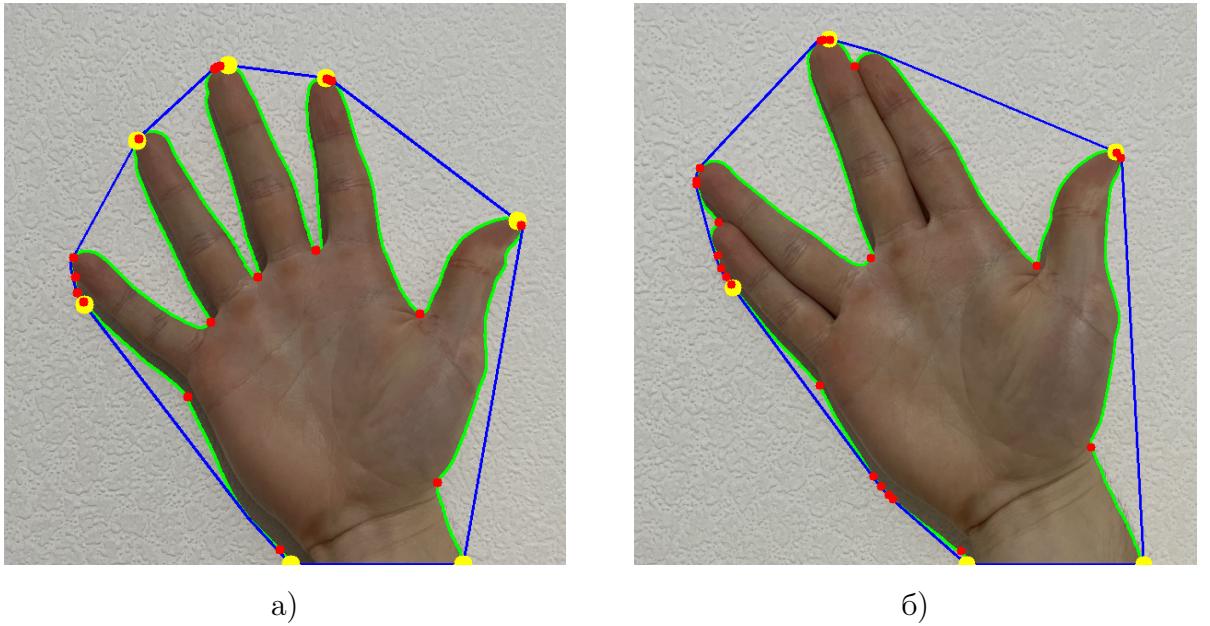


Рис. 21. Результат очистки дефектов выпуклости (оставшиеся точки отмечены на рисунках желтым цветом).

Для того чтобы окончательно найти все точки на руке необходимо найти центр масс контура.

Центром масс фигуры является арифметическое среднее всех точек фигуры. Пусть фигура состоит из n отдельных точек x_1, \dots, x_n , тогда центр масс определяется как

$$c = \frac{1}{n} \sum_{i=1}^n x_i.$$

В контексте обработки изображения и компьютерного зрения каждая фигура состоит из пикселей и центром масс является взвешенное среднее всех пикселей, составляющих фигуру.

Центр масс контура можно найти с помощью момента. Момент изображения – это конкретное средневзвешенное значение интенсивности пикселей изображения. Как и во всех остальных науках, моменты характеризуют распределение материи относительно точки или оси. Формула для момента изображения выглядит следующим образом:

$$M_{ij} = \sum_x \sum_y x^i y^j I(x, y), \quad (40)$$

где $I(x, y)$ – интенсивность пикселя в точке (x, y) , x, y относятся к строке и столбцу изображения.

Проблема формулы (40) состоит в том, что моменты чувствительны к позициям x и y . Если есть необходимость в определении моментов, независимых к месту расположения контура, то нужно использовать формулу *центральных моментов*:

$$M_{pq} = \sum_x \sum_y (x - \bar{x})^p (y - \bar{y})^q I(x, y),$$

где \bar{x} и \bar{y} – средние значения x и y соответственно.

Координата центра масс изображения таким образом будет определяться как:

$$\begin{aligned} C_x &= \frac{M_{10}}{M_{00}}, \\ C_y &= \frac{M_{01}}{M_{00}}, \end{aligned} \tag{41}$$

где (C_x, C_y) – координата центра масс изображения.

В библиотеке OpenCV есть функция `moments()`, определяющая моменты изображения. Добавив в предыдущий код определение координаты центра масс и построение лучей, получим результат, отображённый на рисунке 22.

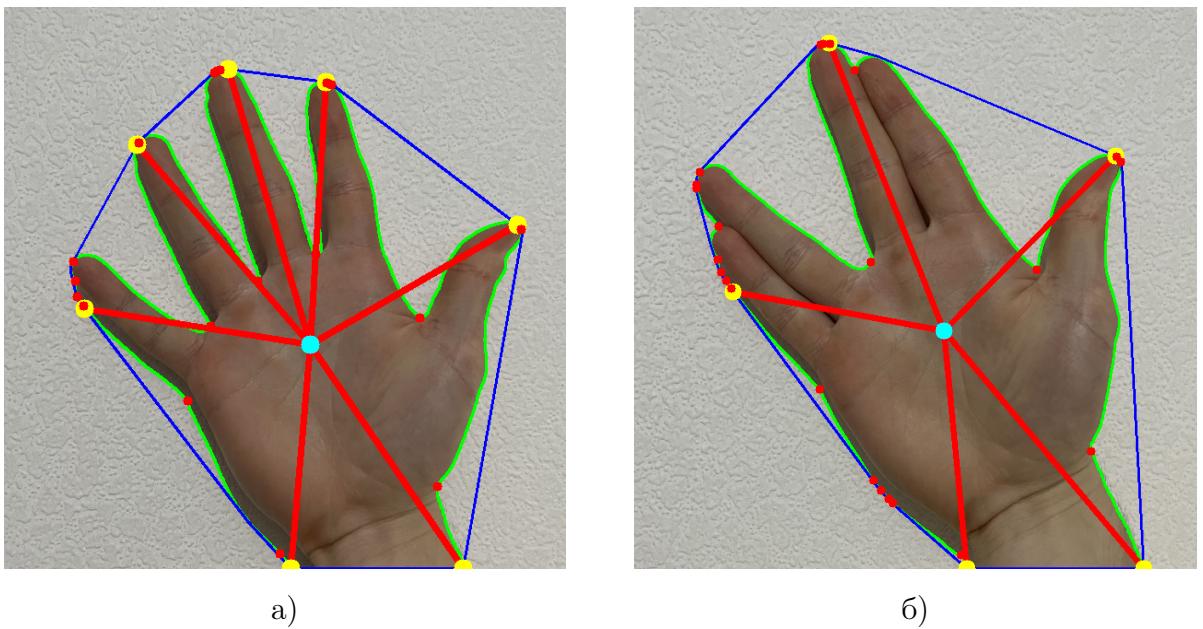


Рис. 22. Результат поиска ключевых точек руки.

3.2 Локализация ключевых точек кисти руки на изображении на основе непрерывного скелета.

Для локализации ключевых точек в статье [15] предлагается использовать непрерывный скелет. Предполагается, что уже успешно был выполнен этап сегментации и имеется отсегментированное изображение с силуэтом кисти руки. На основе контурного представления силуэта жеста строится его скелет. Для определения скелета используется понятие *максимального пустого круга*.

Определение 1. Для многоугольной фигуры F *максимальным пустым кругом* называется всякий круг B , полностью содержащийся внутри фигуры F , такой, что любой другой круг B' , содержащийся внутри фигуры F , не содержит в себе B .

Определение 2. *Скелетом* многоугольной фигуры F является множество центров её максимальных пустых кругов.

Непрерывный скелет многоугольной фигуры является подмножеством диаграммы Воронова [16]. Совокупность общих линий всех пар несмежных ячеек диаграммы Воронова образуют ветви скелета [16]. На скелете определена радиальная функция $R(x, y)$, ставящая в соответствие каждой точке скелета (x, y) значение радиуса максимального пустого круга с центром в этой точке.

В большинстве случаев скелет ладони имеет шумы в виде малозначимых ветвей, которые как правило, мешают дальнейшему анализу. Для удаления шумовых ветвей используется дополнительная обработка, называемая "стрижкой" [16]. Процесс "стрижки" заключается в удалении ветвей, граничащих с контурами силуэта руки.

Существующие эффективные алгоритмы позволяют выполнять построение скелета за время $O(N \log N)$, где N – число вершин в многоугольнике [17]. В связи с тем, что скорость построения скелета напрямую зависит от количества углов многоугольной фигуры, то для ускорения построения скелета можно применить аппроксимацию этой фигуры [18].

Демонстрация процесса построения скелета представлена на рисунке 23. На основе непрерывного скелета и радиальной функции $R(x, y)$ можно с большой точностью вычислить координаты кончиков пальцев и координаты центра ладони. Каждый палец может принимать два условных состояния: сжатый в кулак или разжатый. Все ветви скелета, соответствующие пальцу, оканчиваются вершиной степени 1. Ветвь пальца можно разделить на две части: палец и пясть.

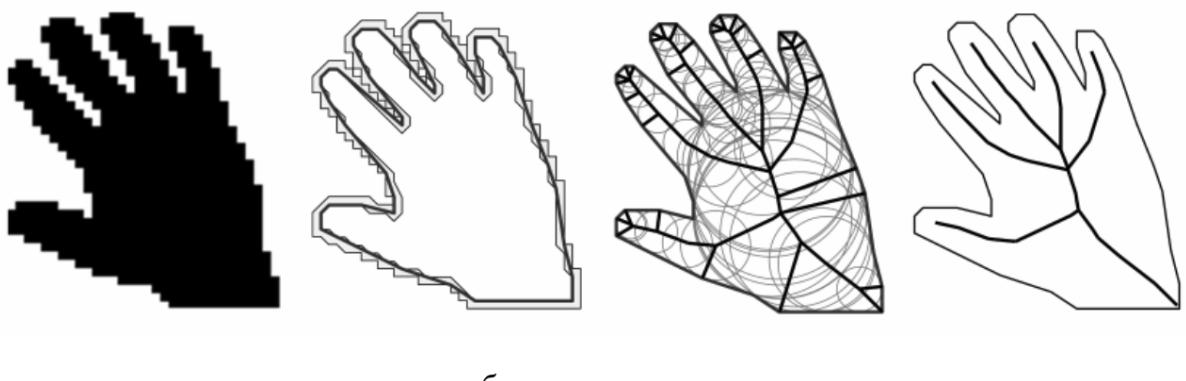


Рис. 23. Процесс построения скелета: *а* – исходное изображение; *б* – аппроксимированное изображение; *в* – скелет многоугольника; *г* – скелет после стрижки

Для классификации ветвей пальцев используется набор эвристических правил:

1. Ветвь пальца лежит на графе между вершинами со степенями 1 и 3.
2. Радиальная функция ветви на вершине степени 1 увеличивается более чем в 2,5 раза по сравнению с вершиной степени 3.
3. Радиальная функция начинает резко расти, то есть частные производные R' больше заданного порога.

Первая точка на ветви, где производная радиальной функции превышает заданный порог, является точкой конца пальца. Центром ладони будем считать точку, лежащую на скелете ладони, радиальная функция которой принимает максимальное значение. На рисунке 24 демонстрируется результат вычисления ключевых точек на изображении.

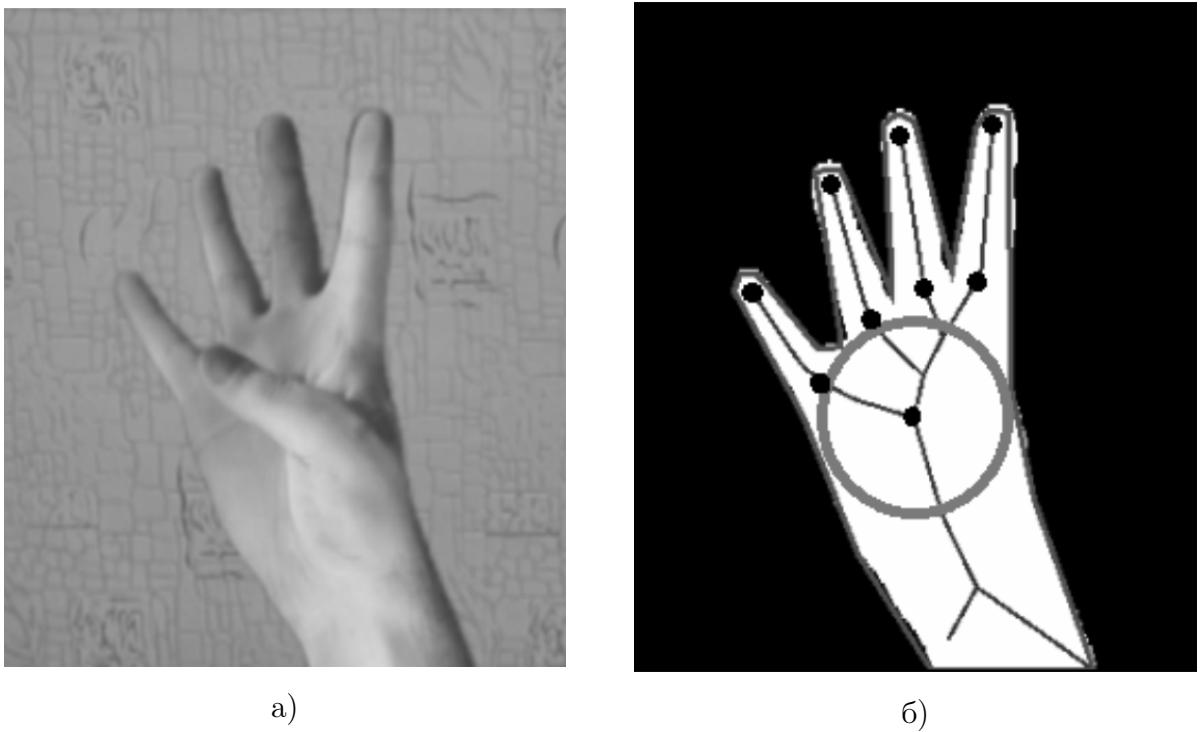


Рис. 24. Определение ключевых точек: *a* – исходное изображение; *б* – скелет и ключевые точки на нём.

Для распознавания простого, ограниченного набора жестов достаточно составить набор эвристических правил, основанных на следующих данных: количество пальцев, их длина, количество циклов в графе и их габариты. В более сложных случаях набора эвристических правил мало, и для распознавания жестов применяются дескрипторы формы кисти руки, состоящие из определённых инвариантных признаков.

Достоинством метода распознавания жестов на основе непрерывного скелета является его быстродействие, высокая точность локализации особых точек и, как следствие, высокий результат распознавания.

Список литературы

- [1] Dix A., Finlay J., Abowd G.D., Beale R. Human-Computer Interaction. - Third Edition, Pearson Education Limited: 2004. - p. 857
- [2] Jiangqin W., Wen G. A FAst Sign Word Recognition Method for Chinese Sign Language // In Proceedings of the Third International Conference on Advances in Multimodal Interfaces (ICMI '00). - Springer-Verlag, London, 2000. - p.599-606
- [3] Sanna A., Lamberti F., Paravati G., Henao R., Eduardo A., Manuri F. A Kinect-Based Natural Interface for Quadrotor Control // Intelligent Technologies for Interactive Entertainment, Volume 78. Springer Berlin Heidelberg, 2012. - p. 48-56
- [4] Basilio, Jorge & Torres, Gualberto & Sanchez-Perez, Gabriel & Toscano, Karina & Perez-Meana, Hector. Explicit image detection using YCbCr space color model as skin detection, 2011. - p. 123-128.
- [5] N. Otsu. A threshold selection method from gray-level histograms (англ.) // IEEE Trans. Sys., Man., Cyber. : journal. — 1979. — Vol. 9. — p. 62—66.
- [6] M. Van Droogenbroeck, O. Barnich. Vibe: A disruptive method for background subtraction. // In T. Bouwmans, F. Porikli, B. Hoferlin, A. Vacavant, editors, Background Modeling and Foreground Detection for Video Surveillance, chapter 7. Chapman and Hall/CRC, pages 7.1-7.23, July 2014.
- [7] P. Kaewtrakulpong, R. Bowden. An improved adaptive background mixture model for real-time tracking with shadow detection // Video-Based Surveillance Systems, pp. 135-144. Springer, 2002.
- [8] Z. Zivkovic, F. Heijden. Efficient adaptive density estimation per image pixel for the task of background subtraction // Pattern recognition letters, Vol. 27(7), pp. 773-780, 2006.
- [9] Z. Zivkovic, F. Heijden. Efficient adaptive density estimation per image pixel for the task of background subtraction // Pattern recognition letters, Vol. 27(7), pp. 773-780, 2006.
- [10] Satoshi S., Keiich A. 1985. Topological Structural Analysis of Digitized Binary Images by Border Following. Computer vision, graphics, and image processing, 30.
- [11] C-H Teh and Roland T. Chin. On the detection of dominant points on digital curves. Pattern Analysis and Machine Intelligence, IEEE Transactions on, 11(8):859–872, 1989.
- [12] Graham R. L. An efficient algorithm for determining the convex hull of a finite planar set //Info. Pro. Lett. – 1972. – Т. 1. – С. 132-133.

- [13] R.A. Jarvis. On the identification of the convex hull of a finite set of points in the plane // Information Processing Letters, Volume 2, Issue 1, 1973, p. 18-21.
- [14] Kirkpatrick, David G.; Seidel, Raimund (1986). "The ultimate planar convex hull algorithm?". SIAM Journal on Computing. 15 (1): 287–299.
- [15] А.В. Носов. Локализация ключевых точек кисти руки на изображении на основе непрерывного скелета. Математические методы моделирования, управления и анализа данных, с. 77-79, 2015.
- [16] Местецкий Л. М. Непрерывная морфология бинарных изображений: фигуры, скелеты, циркуляры. М. : Физматлит, 2009.
- [17] Местецкий Л. М., Рейер И. Непрерывное скелетное представление изображения с контролируемой точностью // International Conference Graphicon. M., 2003. С. 51–54.
- [18] Носов А. В. Алгоритм распознавания жестов рук на основе скелетной модели кисти руки // Вестник СибГАУ. 2014. Вып. 2(54). С. 62–67.

Код реализации алгоритма и построения изображений:

```
1 import numpy as np
2 import cv2
3 import matplotlib.pyplot as plt
4
5
6 def get_min_max(image_flatten: np.ndarray) -> tuple:
7     min_pix, max_pix = image_flatten[0], image_flatten[0]
8     for pixel in image_flatten:
9         if pixel < min_pix:
10             min_pix = pixel
11         if pixel > max_pix:
12             max_pix = pixel
13     return min_pix, max_pix
14
15
16 def create_histogram(image_flatten: np.ndarray,
17                      size: int, min_pix: int) -> np.array:
18     hist = np.zeros(size)
19     for pixel in image_flatten:
20         hist[pixel - min_pix] += 1
21     return hist
22
23
24 def otsu_threshold(image: np.ndarray):
25     img_gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
26     img_gray_flatten = img_gray.flatten()
27     min_pixel, max_pixel = get_min_max(img_gray_flatten)
28     size = max_pixel - min_pixel + 1
29     histogram = create_histogram(img_gray_flatten, size, min_pixel)
30     m = 0
31     n = 0
32     for i in range(size):
33         m += i * histogram[i]
34         n += histogram[i]
35     max_sigma = -1
36     threshold = 0
37     alpha1 = 0
```

```

38     beta1 = 0
39     for i in range(size - 1):
40         alpha1 += i * histogram[i]
41         beta1 += histogram[i]
42         w1 = float(beta1) / n # Вероятность класса 1
43         w2 = 1 - w1
44         a = float(alpha1) / beta1 - float(m - alpha1) / (n - beta1)
45         sigma = w1 * w2 * a * a
46         if sigma > max_sigma:
47             max_sigma = sigma
48             threshold = i
49     threshold += min_pixel
50     return threshold
51
52
53 def save_histogram(image, fname: str):
54     image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY).flatten()
55     plt.hist(image, 255)
56     plt.xlim([0, 255])
57     plt.savefig(fname)
58     plt.show()
59
60
61 INPUT_TO_OUTPUT = {'image3.jpg':
62                     {'histogram': 'histogram_for_3.png',
63                      'output': 'otsu_exmpl_for_3.png'},
64                     'image7.jpg':
65                     {'histogram': 'histogram_for_7.png',
66                      'output': 'otsu_exmpl_for_7.png'}
67                 }
68
69 if __name__ == '__main__':
70     for fname, outputs in INPUT_TO_OUTPUT.items():
71         image = cv2.imread(fname)
72         save_histogram(image, outputs['histogram'])
73         thresh = otsu_threshold(image)
74         print(f'Threshold = {thresh}')
75         ret, img_thresh = cv2.threshold(cv2.cvtColor(image, cv2.COLOR_BGR2GRAY), th
76         imask = img_thresh == 255

```

```
77     canvas = np.zeros_like(image, np.uint8)
78     canvas[imask] = image[imask]
79     cv2.imwrite(outputs['output'], canvas)
```

Код реализации алгоритма Грэхема:

```
1 from functools import cmp_to_key
2
3 class Point:
4     def __init__(self, x=None, y=None):
5         self.x = x
6         self.y = y
7
8     # Расстояние между двумя точками
9     def dist_sq(p1, p2):
10        return ((p1.x - p2.x) * (p1.x - p2.x) +
11                (p1.y - p2.y) * (p1.y - p2.y))
12
13    # Определение ориентации трёх точек
14    # (перекрестное произведение векторов pq и qr)
15    def orientation(p, q, r):
16        val = ((q.y - p.y) * (r.x - q.x) -
17               (q.x - p.x) * (r.y - q.y))
18
19        if val == 0:
20            return 0    # коллинеарны
21
22        elif val > 0:
23            return 1    # по часовой
24
25        else:
26            return 2    # против часовой
27
28    # Сравнение двух точек для сортировки
29    def compare(p1, p2):
30        global p0
31        o = orientation(p0, p1, p2)
32
33        if o == 0:
34            return -1 if dist_sq(p0, p2) >= dist_sq(p0, p1) else 1
35
36        else:
37            return -1 if o == 2 else 1
38
39    def convex_hull(input_points: list) -> list:
40        # Конвертируем в класс Point
41        points = [Point(point[0], point[1]) for point in input_points]
42        n = len(input_points)
```

```

38     # Находим минимальную точку
39     min_y = points[0].y
40     min_i = 0
41     for i in range(1, n):
42         y = points[i].y
43         if ((y < min_y) or
44             (min_y == y and points[i].x < points[min_i].x)):
45             min_y = points[i].y
46             min_i = i
47
48     points[0], points[min_i] = points[min_i], points[0]
49
50     # Сортируем массив
51     global p0
52     p0 = points[0]
53     points = sorted(points, key=cmp_to_key(compare))
54
55     m = 1    # Начальный размер нового массива
56     # Удаляем лишние точки
57     for i in range(1, n):
58         while ((i < n - 1) and
59                 (orientation(p0, points[i], points[i + 1]) == 0)):
60             i += 1
61             points[m] = points[i]
62             m += 1
63     if m < 3:
64         return None
65
66     S = [points[0], points[1], points[2]]
67     for i in range(3, m):
68         while ((len(S) > 1) and
69                 (orientation(S[-2], S[-1], points[i]) != 2)):
70             S.pop()
71             S.append(points[i])
72
73     return [(p.x, p.y) for p in S]

```
