РЕФЕРАТ

Отчет: 44 страницы, 10 рисунков, 13 источников. 1 приложение.

ОБРАБОТКА ИЗОБРАЖЕНИЙ, ДЕСКРИПТОРЫ ТОЧЕК, DAISY, SIFT, SURF, PYTHON.

В данной работе рассмотрены алгоритмы построения дескрипторов в задачах обработки изображений.

Цель работы – разработка устойчивого к пространственным преобразованиям алгоритма построения дескрипторов.

Рассмотрены принципы построения дескрипторов изображений, создана программная реализация с применением алгоритма DAISY, проведено экспериментальное исследование работы алгоритма.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В ряде задач цифровой обработки изображений возникает необходимость однозначного численного описания представленных на изображении точек. В частности, на численном описании отдельных точек и областей изображения основаны решения задач идентификации объектов, совмещения изображений, построения карт глубины и восстановления трехмерной сцены методами фотограмметрии, и многих других.

Математические объекты, описывающие точку изображения, называют дескрипторами. Существует значительное число алгоритмов построения дескрипторов, отличающихся деталями своей реализации, скоростью и точностью работы, а также корректностью результатов для различных исходных данных.

Основными требованиями к дескриптору являются однозначность результата для одинаковых точек изображений, устойчивость к пространственным и яркостным преобразованиям исходных данных, а также вычислительная сложность алгоритмов построения.

Целью работы является разработка дескриптора изображений, устойчивого к пространственным преобразованиям.

В данной работе были рассмотрены некоторые дескрипторы и алгоритмы их построения, методы программной реализации. Разработана реализация построения дескрипторов на основе алгоритма DAISY.

Используемые алгоритмы реализованы на языке программирования общего назначения Python 3.7 с использованием open-source библиотек для обработки изображений OpenCV версии 3.4.2 и scikit-image версии 0.6.12. В качестве тестовых данных использовались наборы изображений из ряда открытых источников.

1 Исследование алгоритмов построения дескрипторов изображений

1.1 Постановка задачи

Дескриптором называют математический объект, сопоставленный с определенной точкой изображения, и представляющий достаточно однозначное ее описание, позволяющее с высокой степенью уверенности идентифицировать аналогичную точку или область на другом изображении [1].

Как правило, дескриптор представляет собой вектор значений, вычисляемый определенной функцией для точки изображения.

Стоит заметить, что на практике для однозначного описания точки изображения недостаточно исключительно информации о яркости отдельного пикселя. В связи с этим, в качестве исходных данных для построения дескриптора используется набор из нескольких точек изображения, находящихся в окрестности заданной.

Рассмотрим следующую постановку задачи. Обозначим дескриптор точки p как d_p. Пусть дано изображение:

$$I : S \to \mathbb{R}, \quad S \subset \mathbb{R}^k, \tag{1}$$

где k – размерность, для плоских изображений равная 2.

Построение дескриптора будет выглядеть следующим образом:

$$d_p = f(I(\hat{p})), \quad \hat{p} \in S. \tag{2}$$

где f – функция построения дескриптора,

$\hat{p}$ – набор точек изображения I.

Данная схема сохраняется независимо от конкретного алгоритма построения дескриптора. Конкретный вид и свойства полученного результа-

та будут зависеть как от вида функции f, так и от конфигурации набора исходных точек $\hat{p}$.

Основным требованием к дескриптору является однозначность результата, т.е. удовлетворение значения (2) следующему выражению для произвольного числа изображений N:

$$f(I_i(\hat{p_i})) = f(I_j(\hat{p_j})), \quad i = 1 \dots N, \quad j = 1 \dots N, \tag{3}$$

где p_i, p_j – пара соответствующих точек изображений I_i, I_j.

Следует заметить, что построенные дескрипторы, как правило, подвергаются нормализации для уменьшения влияния на точность идентификации точек яркостных характеристик изображения и пространственных преобразований [6]. Конкретные методы нормализации будут рассмотрены в следующих разделах.

Таким образом, используя различные функции построения и варьируя набор исходных точек, можно получать дескрипторы, обладающие различными свойствами, достоинствами и недостатками для конкретных задач.

Основными факторами, принимаемыми в расчет при разработке алгоритма построения дескрипторов, являются:

1. Устойчивость к пространственным преобразованиям - сохранение значения дескриптора при повороте, сдвиге, масштабировании изображения;

2. Устойчивость к яркостным преобразованиями - сохранение значения дескриптора при изменении яркости и контрастности;

3. Вычислительная сложность построения;

4. Информативность - достаточность данных дескриптора для дальнейшей идентификации и сравнения точек.

Следует отметить разницу в требованиях к алгоритму построения дескрипторов в зависимости от предметной области применения. В случае использования дескрипторов для описания особых точек (features) необходима повышенная точность идентификации и инвариантность к преобразованиям, вычислительная сложность же не является самым важным критерием, т.к. число особых точек на изображении, как правило, на порядки меньше его размерности.

Напротив, в задачах плотного сопоставления изображений, возникающих в фотограмметрии и построении карт глубины, требуется вычисление дескриптора для каждого пикселя, что накладывает дополнительные ограничения на вычислительную сложность [13]. При этом точностью построения и идентификации возможно в некоторой степени пренебречь, т.к. в данных задачах дескрипторы применяются для оценки общих тенденций преобразований между изображениями, что позволяет за счет их большого количества добиться хороших результатов усреднением и дополнительной валидацией.

Существует значительное число реализаций алгоритмов построения дескрипторов. Далее будут рассмотрены некоторые популярные методы и их особенности.

1.2 Существующие методы построения дескрипторов

1.2.1 SIFT

Большинство существующих дескрипторов основаны на методах сбора информации о градиенте освещенности изображения в некоторой локальной окрестности точки. Распространенным решением для построения дескрипторов является алгоритм SIFT [2].

В методе SIFT дескриптором является вектор. Как и направление ключевой точки, дескриптор вычисляется на гауссиане, ближайшем по масштабу к ключевой точке, и исходя из градиентов в некотором окне ключевой точки. Перед вычислением дескриптора это окно поворачивают на угол направления ключевой точки, чем и достигается инвариантность относительно поворота.

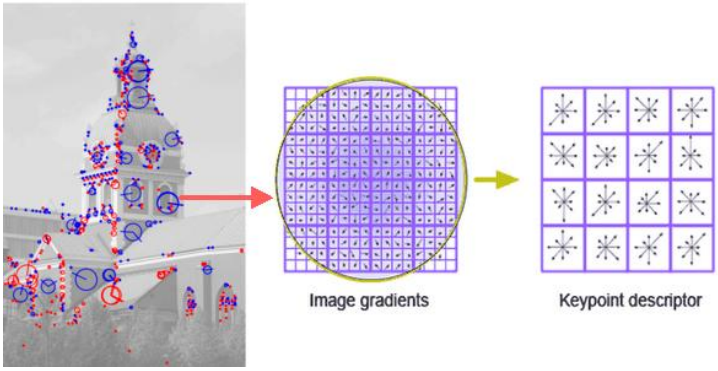

Рисунок 1 – Построение дескриптора SIFT

Дескриптор точки состоит из всех полученных гистограмм в ее локальной окрестности. На практике используются дескрипторы размерности 128 компонент (4x4x8).

Полученный дескриптор нормализуется, после чего все его компоненты, значение которых больше 0.2, урезаются до значения 0.2 и затем дескриптор нормализуется ещё раз. В таком виде дескрипторы готовы к использованию.

Существует большое число модификаций SIFT, направленных на улучшение устойчивости к определенным преобразованиям.

1.2.2 SURF

Дескриптор SURF (Speeded up Robust Features) относится к числу тех дескрипторов, которые одновременно выполняют поиск особых точек и строят их описание, инвариантное к изменению масштаба и вращения. Кроме того, сам поиск ключевых точек обладает инвариантностью в том смысле, что повернутый объект сцены имеет тот же набор особых точек, что и образец.

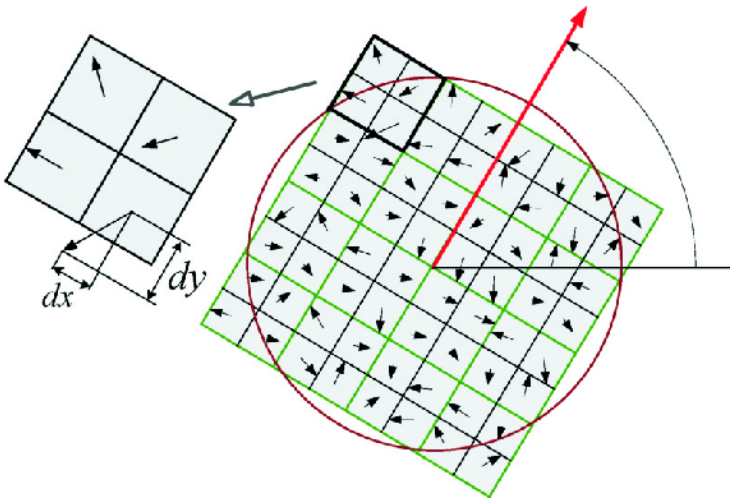

Рисунок 2 – Построение дескриптора SURF

Определение особых точек на изображении выполняется на основании матрицы Гессе. Использование Гессиана обеспечивает инвариантность относительно преобразования типа ”поворот но не инвариантность относительно изменения масштаба. Поэтому SURF применяет фильтры разного масштаба для вычисления Гессиана. Для каждой найденной особой точки вычисляется ориентация – преобладающее направление перепада яркости. Понятие ориентации близко к понятию направления градиента, но для определения ориентации особой точки применяется фильтр Хаара. Дескриптор формируется в результате склеивания взвешенных описаний

градиента для 16 квадрантов вокруг особой точки. Элементы дескриптора взвешиваются на коэффициенты Гауссова ядра. Веса необходимы для большей устойчивости к шумам в удаленных точках.

1.2.3 GLOH

Дескриптор GLOH (Gradient location-orientation histogram) является модификацией SIFT-дескриптора, который построен с целью повышения надежности. Вычисляется SIFT дескриптор, но используется полярная сетка разбиения окрестности на бины: 3 радиальных блока с радиусами 6, 11 и 15 пикселей и 8 секторов. В результате получается вектор, содержащий 272 компоненты, который проецируется в пространство размерности 128 посредством использования анализа главных компонент (PCA).

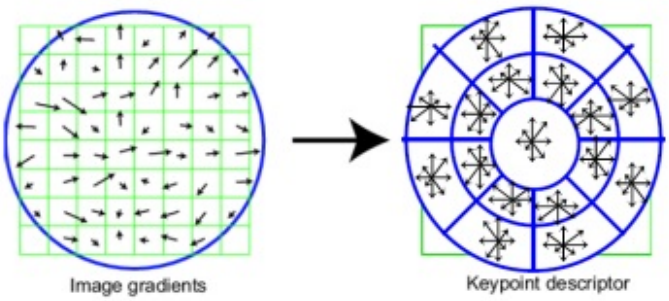

Рисунок 3 – Построение дескриптора GLOH

Общим для всех описанных алгоритмов является основанность на механизмах расчета локального градиента. Данный процесс, несмотря на высокую точность, может быть достаточно вычислительно сложным, что в особенности проявляется в задачах плотного сопоставления, где вычисления производятся для каждой точки изображения [6]. Альтернативное решение используется в методе DAISY [3], выбранном в качестве основы алгоритма в данной работе.

2 Разработка алгоритма построения дескрипторов изображений

2.1 Алгоритм построения дескрипторов

В качестве основы для разрабатываемого метода используется алгоритм DAISY [3]. В основе алгоритма лежит метод построения гистограмм направленных градиентов.

Гистограммой направленных градиентов называют набор направлений градиентов изображения, построенный в определенной области. Данная техника позволяет описать точку распределением интенсивности в ее окрестности.

Для построения градиента используются стандартные фильтры, например, оператор Собеля.

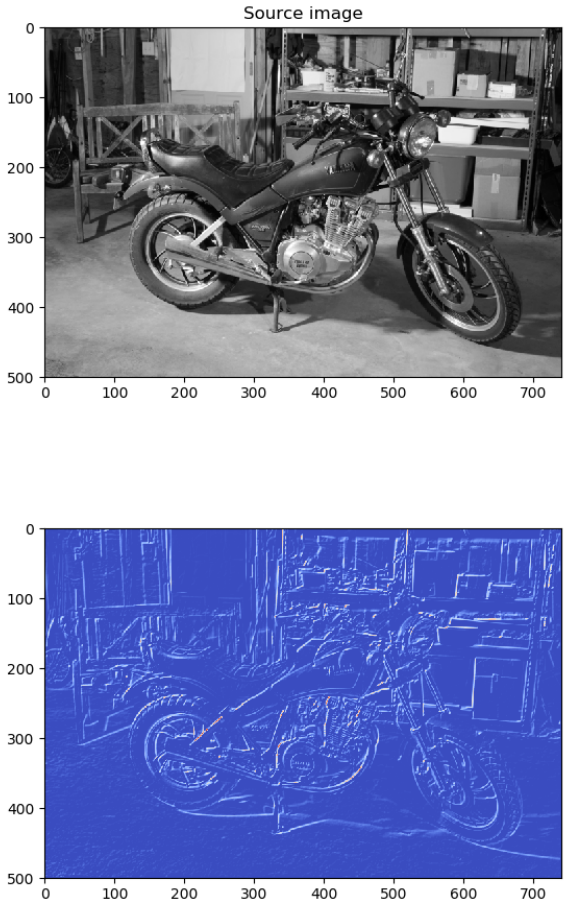

Рисунок 4 – Пример построения градиента изображения

Построенные карты градиентов подвергаются последовательному размытию с постепенным увеличением ядра гауссовского фильтра. Полученные карты называются свернутыми картами направлений (Convolved Orientation Maps, COM).

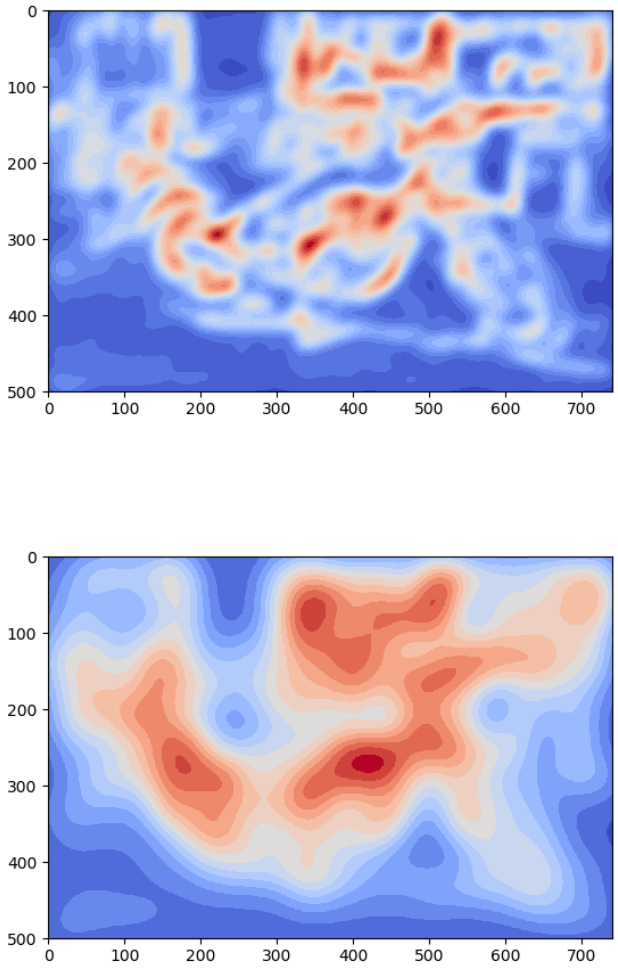

Рисунок 5 – Пример COM с различными степенями размытия

Вычисление COM для изображения I производится следующим образом:

$$G_o^\sigma = G^\sigma * \left(\frac{dI}{do}\right)^+, \left(\frac{dI}{do}\right)^+ = max\left(\frac{dI}{do}, 0\right), \tag{4}$$

где G^σ – ядро Гауссовского фильтра с стандартным отклонением σ,

o – направление градиента.

Таким образом, получается набор карт градиента, каждая из которых с увеличением уровня размытия представляет собой все более обобщенную информацию о направлениях интенсивности исходного изображения.

Описанные операции выполняются один раз при начале работы алгоритма, полученные T карт с различными уронями размытия далее используются для выборки результатов.

Выборка результатов для каждой точки (u, v) изображения I производится по следующей схеме:

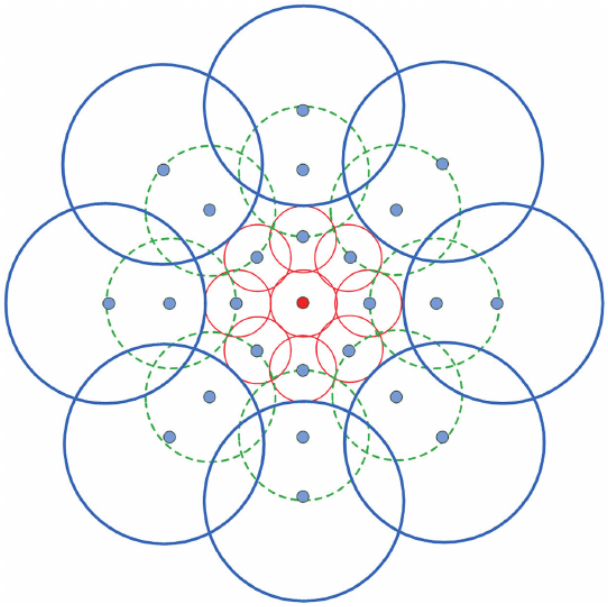

Рисунок 6 – Схема выборки значений из набора COM

Для каждой указанной на схеме точки выбираются значения COM, составляя очередной элемент дескриптора $h_\sigma(u, v)$.

Каждый круг на схеме соответствует одной области выборки значений COM в его центральной точке. COM строятся для H отдельных направлений, получая таким образом H значений в точке.

Точки выборки расположены Q концентрическими кольцами вокруг центральной, для которой производится вычисление дескриптора, каждое кольцо задает $T \cdot H$ значений результата.

Следовательно, общее число элементов дескриптора будет равно:

$$D_s = (Q \cdot T + 1) \cdot H. \tag{5}$$

Вектор значений COM $h_\sigma(u, v)$ для точки (u, v) задается следующим образом:

$$h_\sigma(u, v) = [G_1^\sigma(u, v), \ldots, G_H^\sigma(u, v)]^T. \tag{6}$$

Полученный вектор значений нормализуется.

Полный дескриптор для точки (u_0, v_0) получается путем конкатенации всех нормализованных векторов значений COM $h_{\sigma i}(u, v)$, начиная с центральной точки:

$$D(u_0, v_0) = [h_{\sigma 1}(u_0, v_0), h_{\sigma 1}(I_1(u_0, v_0, R_1)), \ldots, h_{\sigma 1}(I_T(u_0, v_0, R_1),$$

$$h_{\sigma 2}(I_1(u_0, v_0, R_2)), \ldots, h_{\sigma 2}(I_T(u_0, v_0, R_2),$$

$$\ldots\ldots$$

$$h_{\sigma Q}(I_1(u_0, v_0, R_Q)), \ldots, h_{\sigma Q}(I_T(u_0, v_0, R_Q)],$$

где $I_j(u, v, R)$ – точка на расстоянии R от (u, v) в направлении j при числе направлений T.

При необходимости может производиться нормализация вектора значений дескриптора одним из следующих способов:

1. Нормализация отдельных векторов $h_\sigma(u, v)$;

2. Полная нормализация вектора $D(u_0, v_0)$;

3. Метод нормализации, применяемый в SIFT. После полной нормализации $D(u_0, v_0)$ вектор приводится к единичной сумме.

Как можно видеть, описанный алгоритм отличается достаточно низкой вычислительной сложностью и простотой программной реализации.

Наиболее ресурсоемким этапом является построение диаграммы направленных градиентов и последовательное размытие COM [13].

Данные операции хорошо поддаются распараллеливанию или оптимизации путем замены свертки с крупным ядром последовательными свертками с ядрами меньшего размера, и выполняются один раз для всего изображения. Конфигурация паттерна выборки и методы нормализации влияют на производительность в незначительной степени, и могут варьироваться в широких пределах без серьезного повышения вычислительной сложности.

2.2 Программная реализация

Вышеописанный алгоритм построения дескрипторов для исследования был реализован на языке Python 3.7 с использованием библиотеки OpenCV 3.4.2 в программной среде jupyter.

Программная реализация поддерживает задание произвольных значений параметров R, Q, T, H алгоритма DAISY, задание последовательностей стандартных отклонений σ для каждого кольца, а также четыре варианта нормализации построенного дескриптора - без нормализации, нормализация отдельных векторов $h_\sigma(u, v)$, нормализация полного вектора $D(u_0, v_0)$, SIFT-style нормализация $D(u_0, v_0)$ с приведением к единичной сумме.

Паттерн расположения точек выборки значений алгоритмом DAISY может быть графически визуализирован, что позволяет наглядно наблюдать влияние изменения отдельных параметров на работу алгоритма.

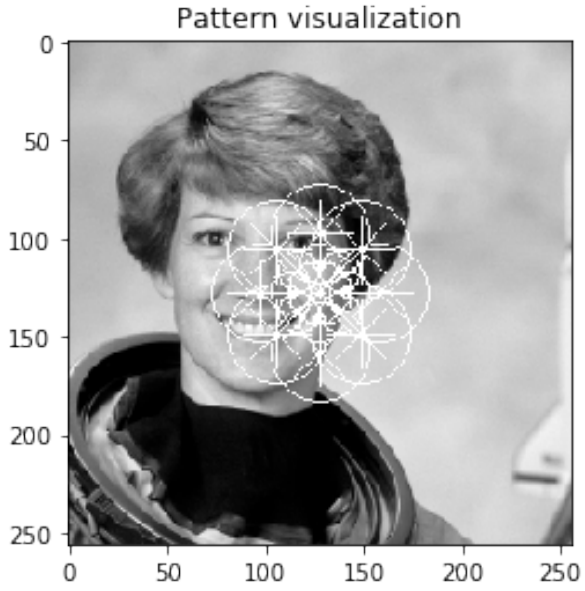

Рисунок 7 – Пример визуализации паттерна точек выборки

Поддерживается обработка изображений любых размеров, приведенных к градациям серого.

Все построенные в процессе работы алгоритма карты градиента и COM сохраняются и могут быть выведены в виде отдельных изображений

для более подробного анализа.

Реализованы два варианта построения дескрипторов:

1. Тестовая функция построения дескриптора для одной точки в центре изображения (файл приложения daisy-custom.py);

2. Функция построения множества дескрипторов по сетке на исходном изображении с варьируемым шагом (файл приложения daisy-dense.py);

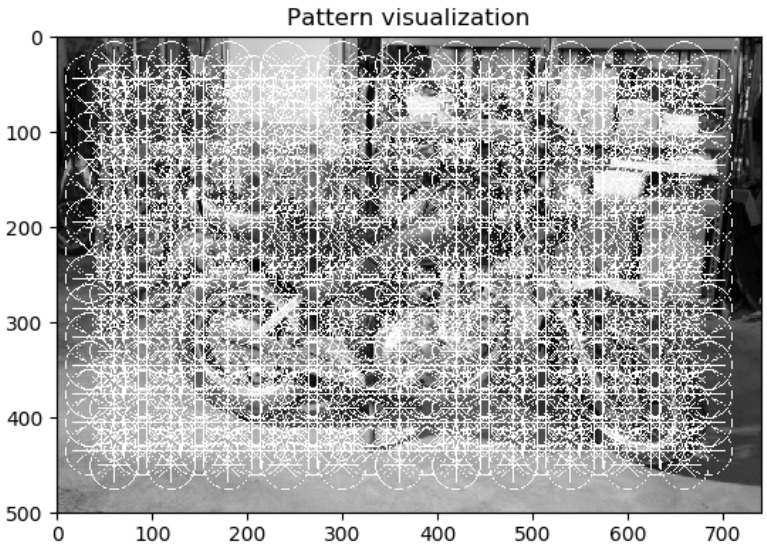

Рисунок 8 – Пример визуализации работы плотного построения дескрипторов по сетке

Вычисление градиента изображения производится посредством комбинации горизонтального и вертикального оператора Собеля:

```
def gradient_orientation(image, tetha):
    # Steering filter - combination of X and Y derivatives
    # to det a derivative on a specific angle
    d_hor = sobel_h(image)
    d_ver = sobel_v(image)
    derivative = d_hor * np.cos(tetha) + d_ver * np.sin(tetha)
    # Operator max(d, 0) leaves only positive derivative
    derivative = derivative.clip(min=0)
    return derivative
```

На COM, представленных на рисунке 9, представлен пример разницы результатов при построении градиента по 8 различным направлениям с шагом в $45°$.

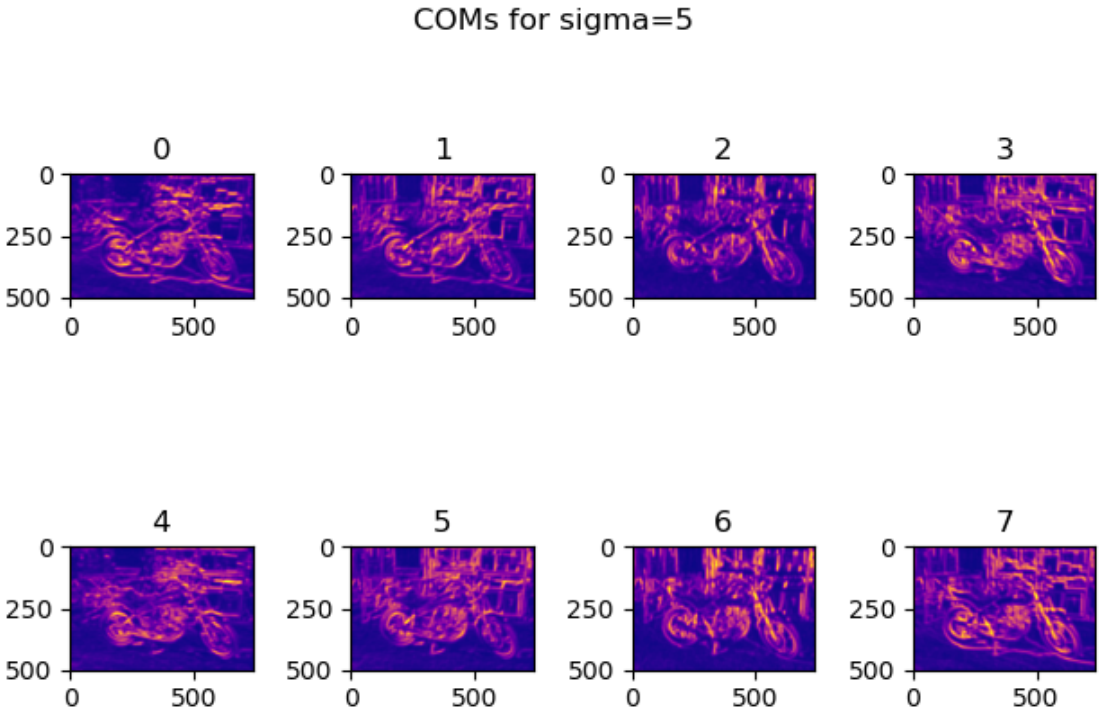

Рисунок 9 – Пример построения COM посредством комбинации горизонтального и вертикального оператора Собеля

После построения COM производится выборка и формирование векторов $h_\sigma(u, v)$:

```
            for i in range(0, Q):
                    r = (i+1) * radius_step
                    maps = conv_orient_maps[i]
                    for j in range(0, T):
                            tetha = j * tetha_step
                            x = int(center[0] + r * np.cos(tetha))
                            y = int(center[1] + r * np.sin(tetha))

                            h = sample_h_vector(x, y, maps, visualize, sigmas[i+1])
```

Формирование векторов $h_\sigma(u, v)$ из выборки значений COM:

```
# Samples the single h vector at point
def sample_h_vector(x, y, maps, visualize, sigma):
        # Sampling the vector
        h = np.zeros(H)
        for k in range(0, H):
        h[k] = maps[k][x, y]
        return h
```

После формирования результирующего вектора дескриптора $D(u_0, v_0)$ и произведения нормализации выбранным способом дескриптор может быть выведен в следующем виде (представлен пример с нормированием значений SIFT-style):

```
Single descriptor:
[[0.00273817 0.01315129 0.01715979 0.01531712 0.00407314 0.01419661
0.01715979 0.01447451 0.00375679 0.01569022 0.01715979 0.01715979
0.00495935 0.01266778 0.01715979 0.01325313 0.00724256 0.00450238
0.00218893 0.00315323 0.00457862 0.00901201 0.01123045 0.01143024
0.00482866 0.00384051 0.00382658 0.00454053 0.00506419 0.00986872
0.01211622 0.01023564 0.0048071 0.00724932 0.00974763 0.00951843
0.00596861 0.01580851 0.01715979 0.01643501 0.00444591 0.00996207
0.01451799 0.01715979 0.01393738 0.00965086 0.0045864 0.00522394
0.00699046 0.00985555 0.01017609 0.01033389 0.00711645 0.00592839
0.00449626 0.00622854 0.00471659 0.01351514 0.01715979 0.01462342
0.00416372 0.0118479 0.01674489 0.01373806 0.00661411 0.00674226
0.00667678 0.00716366 0.00677847 0.00632861 0.00592743 0.00651756
0.00501556 0.0086749 0.01124444 0.01052421 0.00628421 0.00961532
0.01130574 0.00967048 0.01139017 0.01517235 0.01581414 0.01381832
0.00966419 0.01571628 0.01715979 0.01680317 0.00426777 0.01104361
0.01525348 0.013842 0.00621065 0.00545466 0.00540664 0.00550541
0.00533998 0.00908383 0.01204839 0.01122037 0.0060103 0.00510225
0.00574727 0.00629083]]

Descriptor norm: 0.10821192596770395
Sum:0.9999999999999999
```

Как можно видеть, результат работы программы предоставляет достаточно обширное количество информации для анализа - построенные дескрипторы, полные наборы GO и COM, использовавшихся в вычислении,

графическую визуализацию паттерна выборки.

Этот факт, вкупе с возможностью варьировать любые параметры алгоритма и реализацией в jupyter notebook - гибкой программной среде с широким функционалом для исследования и анализа - позволяет заявить, что данная реализация является подходящей базой для дальнейшего исследования, изучения работы алгоритма в различных условиях, модификации и оптимизации.

Тестирование алгоритма показывает достаточную для исследовательских задач скорость работы:

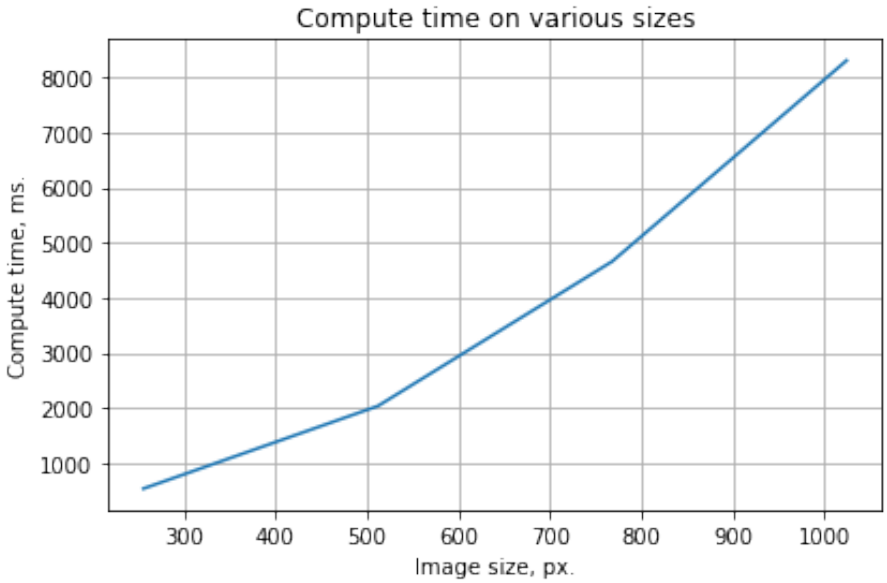

Рисунок 10 – Скорость выполнения алгоритма в зависимости от размера исходного изображения

Тестирование производилось на системе с процессором Intel Xeon X3460 в среде jupyter, операционная система Windows.

Как можно видеть, время выполнения неоптимизированного алгоритма с необязательными дополнительными шагами в виде визуализации паттерна выборки и COM не превышает 9 секунд на размере изображения 1024x1024, что является достаточным результатом для исследовательских задач.

Полный код программы представлен в приложении А.

ЗАКЛЮЧЕНИЕ

В ходе данной работы была собрана и проанализирована информация о существующих методах построения дескрипторов изображений. На основе существующего алгоритма DAISY был разработан алгоритм, потенциально обладающий высокой степенью устойчивости к пространственным преобразованиям исходного изображения. Разработанный алгоритм обладает набором параметров, позволяющих достаточно свободно варьировать точность и вычислительную сложность построения без изменения исходного кода, что позволяет эффективно адаптировать алгоритм к конкретным задачам. Построенные дескрипторы можно использовать как в задачах распознавания, так и в задачах плотного сопоставления изображений.

Создана программная реализация алгоритма с окружением, позволяющим гибко варьировать параметры и анализировать результаты, данная реализация планируется к использовано при продолжении научной работы с целью разработки инвариантной к пространственным преобразованиям модификации дескриптора.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Bres, S. Detection of interest points for image indexation [Текст] / S. Bres, J. M. Jolion // International Conference on Advances in Visual Information Systems. – Springer, Berlin, Heidelberg, 1999. – P. 427-435.

2 Lowe, D. G. Distinctive image features from scale-invariant keypoints [Текст] / D. G. Lowe // International journal of computer vision. – 2004. – Vol. 60. – I. 2. – P. 91-110.

3 Tola, E. DAISY: A Fast Local Descriptor for Dense Matching [Текст] / E. Tola, V. Lepetit, P. Fua // IEEE Transactions on Pattern Analysis and Machine Intelligence. 2010. Vol. 32, num. 5 – P. 815-830.

4 Rublee, E. ORB: an efficient alternative to SIFT or SURF [Текст] / E. Rublee, D. Brando, J. Joestar // ICCV '11 Proceedings of the 2011 International Conference on Computer Vision. – IEEE Computer Society Washington, DC, USA, 2011. – P. 2564-2571.

5 Kong, H. A generalized Laplacian of Gaussian filter for blob detection and its applications [Текст] / H. Kong, H. C. Akakin, S. E. Sarma // IEEE transactions on cybernetics. – IEEE Computer Society Washington, DC, USA, 2013. – V. 43. – I. 6. – P. 1719-1733.

6 Rekhil, M. A Survey on Image Feature Descriptors [Текст] / M. Rekhil, K. Sreekumar // (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 5 (6) , 2014, – P. 7668-7673.

7 Funayama, R. Robust interest points detector and descriptor [Текст] / R. Funayama, H. Yanagihara // Computer Vision and Image Understanding (CVIU), Vol. 110, No. 3 – P. 346–359.

8 Panchal, P. A Comparison of SIFT and SURF [Текст] / R. Funayama, H. Yanagihara // International Journal of Innovative Research in Computer and Communication Engineering Vol. 1, Issue 2, April 2013 – P. 287–302.

9 OpenCV Documentation [Электронный ресурс] : официальная документация библиотеки OpenCV. / Intel Corporation, Willow Garage Inc., Itseez Ltd. – Электрон. дан. – 2019. – URL: https://docs.opencv.org (дата обращения: 20.11.2019).

10 Scikit-image Documentation [Электронный ресурс] : официальная документация библиотеки scikit-image. / The scikit-image team. – Электрон. дан. – 2019. – URL: https://scikit-image.org/docs/stable/ (дата обращения: 25.11.2019).

11 Karami, E. Image Matching Using SIFT, SURF, BRIEF and ORB: Performance Comparison for Distorted Images [Электронный ресурс] / E. Karami, S. Prasad, M. Shehata // Journal of Visual Communication and Image Representation – 2015. – Электрон. дан. – URL: https://www.sciencedirect.com/science/article/pii/S1047320315002230 (дата обращения: 03.12.2019).

12 Xue, B. A DAISY descriptor based multi-view stereo method for large-scale scenes [Электронный ресурс] / B. Xue, L. Cao // Advances in Intelligent Systems and Computing – 2015. – Электрон. дан. – URL: https://doi.org/10.1016/j.jvcir.2015.11.007 (дата обращения: 06.12.2019).

13 Tola, E. DAISY: An Efficient Dense Descriptor Applied to Wide-Baseline Stereo [Электронный ресурс] / E. Tola //IEEE Xplore Digital Library – 2014. – Электрон. дан. – URL: https://ieeexplore.ieee.org/document/4815264 (дата обращения: 05.12.2019).

ПРИЛОЖЕНИЕ А

Код программы

Файл daisy-dense.py:

```
#!/usr/bin/env python
# coding: utf-8

# In[10]:



import numpy as np

from skimage import data
from skimage.color import rgb2gray
from skimage.filters import sobel_h, sobel_v, gaussian
from skimage.feature import match_descriptors
from skimage.exposure import rescale_intensity
from skimage.util import img_as_float, img_as_ubyte
from skimage import transform
from sklearn import preprocessing

import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
import seaborn as sb



from skimage.draw import (line, circle_perimeter)
from enum import Enum

get_ipython().run_line_magic('matplotlib', 'notebook')



# ### DAISY implementation
#
# implementation of DAISY descriptors calculation for the whole image

# In[11]:
```

```
class NormType(Enum):
    NRM_NONE = 1
    NRM_PARTIAL = 2
    NRM_FULL = 3
    NRM_SIFT_STYLE = 4

# Computes a Gradient Orientation Map on derivative axis
# with angle tetha
def gradient_orientation(image, tetha):
    # Steering filter - combination of X and Y derivatives
    # to det a derivative on a specific angle
    d_hor = sobel_h(image)
    d_ver = sobel_v(image)
    derivative = d_hor * np.cos(tetha) + d_ver * np.sin(tetha)
    # Operator max(d, 0) levaes only positive derivative
    derivative = derivative.clip(min=0)
    return derivative

# Compute GOs for a number of directions
def compute_global_orientation_maps(image, H):
    tetha_step = 2 * np.pi / H
    result = []
    for k in range(0, H):
        tetha = tetha_step * k
        result.append(gradient_orientation(image, tetha))
    return result

# Convolves orientation maps with Gaussian
# kernels with specified standard deviations
def convolve_orientation_maps(gos, sigmas):
    result = []
    for sigma in sigmas:
        conv = []
        for go in gos:
            conv.append(gaussian(go, sigma))
        result.append(conv)
    return result

def daisy_full(image, R, Q, T, H, sigmas, norm=NormType.NRM_NONE):
```

```
 # Setting steps in angle and radius
tetha_step = 2 * np.pi / T
radius_step = float(R) / Q
# Calculating total number of entries in a descriptor
Ds = (Q*T+1)*H

# Copy of image for visualization
visualize = np.copy(image)

# Compute GOs
gradient_orientations = compute_global_orientation_maps(image, H)
# Convolve GOs with Gaussian kernels to get COMs
conv_orient_maps = convolve_orientation_maps(gradient_orientations, sigmas)

img_shape = image.shape
rows = img_shape[0] // (2 * R)
columns = img_shape[1] // (2 * R)

descriptors = []

for desc_i in range(1, rows):
    for desc_j in range(1, columns):
        descriptor = np.zeros((1, Ds))
        # Sampling vectors from COMs at DAISY points
        # Center point
        center = [R * desc_i * 2, R * desc_j * 2]
        h = sample_h_vector(center[0], center[1], conv_orient_maps[0], visualize,
            ↪ sigmas[0])
        descriptor[0, 0:H] = h
        # Sample the rest of the points in DAISY pattern
        for i in range(0, Q):
            r = (i+1) * radius_step
            maps = conv_orient_maps[i]
            for j in range(0, T):
                tetha = j * tetha_step
                x = int(center[0] + r * np.cos(tetha))
                y = int(center[1] + r * np.sin(tetha))

                h = sample_h_vector(x, y, maps, visualize, sigmas[i+1])
                # Normalization for partial norm type
```

```
                if norm == NormType.NRM_PARTIAL:
                    h = preprocessing.normalize(h.reshape((1, -1)), norm='l2')

                idx_start = ((i)*T+j)*H+H
                idx_end = ((i)*T+j)*H+2*H
                descriptor[0, idx_start:idx_end] = h

            # Normalization for full norm type
            if norm == NormType.NRM_FULL:
                descriptor = preprocessing.normalize(descriptor.reshape((1, -1)), norm='
                    ↪ l2')
            # Normalization for SIFT-style norm
            if norm == NormType.NRM_SIFT_STYLE:
                descriptor = preprocessing.normalize(descriptor.reshape((1, -1)), norm='
                    ↪ l2')
                descriptor = descriptor.clip(max=0.154)
                descriptor /= descriptor.sum()

            descriptors.append(descriptor)

    result = np.array(descriptors)

    return result, visualize, gradient_orientations, conv_orient_maps


# Samples the single h vector at point
def sample_h_vector(x, y, maps, visualize, sigma):
    # Sampling the vector
    h = np.zeros(H)
    for k in range(0, H):
        h[k] = maps[k][x, y]
    # Visualize the point
    draw_circle(visualize, x, y, sigma)
    draw_lines(visualize, x, y, sigma, H)
    return h


# #### Utility functions
# Functions for data visualization
```

```
# In[12]:


# Draws a circle
def draw_circle(image, x, y, radius):
    rr, cc = circle_perimeter(x, y, radius)
    if x+radius >= image.shape[0] or y+radius >= image.shape[1]:
        return
    image[rr, cc] = 1

# Draws a set of evenly arranged lines from center
def draw_lines(image, x, y, radius, n):
    tetha_step = 2 * np.pi / n
    for i in range(0, n):
        tetha = i * tetha_step
        line_len = int(radius * 0.7)
        x1 = int(x + line_len * np.sin(tetha))
        j1 = int(y + line_len * np.cos(tetha))
        rr, cc = line(x, y, x1, j1)
        if x1 >= image.shape[0] or j1 >= image.shape[1]:
            continue
        image[rr, cc] = 1

# Shows an image
def imgshow(image, title='', cmap=plt.cm.gray):
    fig, ax = plt.subplots()
    ax.imshow(img_as_ubyte(image), cmap=cmap)
    ax.set_title(title)
    plt.show()

# Shows a set of images in a grid
def show_maps(maps, title=''):
    grid_i = 2
    grid_j = int(np.ceil(len(maps) / 2))
    fig, ax = plt.subplots(grid_i, grid_j)
    fig.suptitle(title)
    for i in range(0, grid_i):
        for j in range(0, grid_j):
            idx = i*grid_j+j
            if(idx >= len(maps)):
```

```
                break
            ax[i, j].set_title(idx)
            ax[i, j].imshow(maps[idx], cmap = plt.cm.plasma)
    plt.tight_layout()

# Simple XY 2d plot
def plot_xy(x, y, title='', xlabel='', ylabel=''):
    fig, ax = plt.subplots()
    ax.plot(x, y)
    ax.grid()
    ax.set_title(title)
    ax.set_ylabel(ylabel)
    ax.set_xlabel(xlabel)
    plt.tight_layout()
    plt.show()


# ## Test data preparation
# We use a 128 * 128 sample of a contrast grayscale image from skimage.data

# In[13]:


img_left, img_right, disp = data.stereo_motorcycle()

img_src = rgb2gray(img_as_float(img_left))

print(img_src.shape)

image = img_src
imgshow(image, 'Source image')
print(f'Image size:{image.shape}')


# ## Testing section
# Here is the list of modifiable parameters:
# * Maximum radius $R$
# * Number of rings $Q$
# * Number of points per ring $T$
# * Number of histograms per point $H$
```

```
# * Gaussian filter standard deviations σ for each ring
# * Normalization type
#
# **Notice that big sigma values or radius may result in index out of bounds exception
    ↪ !**

# In[14]:


R = 30
Q = 2
T = 6
H = 8
SIGMAS = [5, 10, 25]
NORM = NormType.NRM_SIFT_STYLE


# In[15]:


# Computing one descriptor at the center of the image
x, y = image.shape
point = [int(x/2), int(y/2)]

import time
time1 = time.time()

descriptors, visualize, gradient_orientations, coms = daisy_full(image, R, Q, T, H,
    ↪ SIGMAS, NORM)

time2 = time.time()
print('Took {:.3f} ms'.format((time2-time1)*1000.0))


imgshow(visualize, title='Pattern visualization')

part = visualize[32:164, 32:164]
imgshow(part, title='Close-up:')
```

```
# In[16]:


print(f'Results shape: {descriptors.shape}')
print(f'Single descriptor shape:\n {descriptors[0].shape}\n')
print(f'Single descriptor:\n {descriptors[0]}\n')
print(f'Descriptor norm: {np.linalg.norm(descriptors[0])}')
print(f'Sum:{np.sum(descriptors[0])}')


# In[17]:


for i in range(len(SIGMAS)):
    show_maps(coms[i], title=f'COMs for sigma={SIGMAS[i]}')


# In[18]:


idx = 1
blur_step = 2

imgshow(gradient_orientations[idx], cmap=plt.cm.plasma)
imgshow(coms[blur_step][idx], cmap=plt.cm.plasma)


# In[ ]:
```

Файл daisy-custom.py:

```
#!/usr/bin/env python
# coding: utf-8

# # Basic implementation of DAISY descriptor

# ## DAISY Algorithm
# DAISY samples the local gradient information as shown in a figure below. Each circle
    ↪  represents a region where the radius is proportional to the standard
    ↪ deviations of the Gaussian kernels. Dots represents locations where the
    ↪ Convolved Orientation Maps (COMs) are sampled.
#
# ![image.png](attachment:image.png)
#
# Each circle represents one histogram region, which is part of the descriptor vector.
    ↪  Each histogram represents the Gradient Orientations (GOs) within this region.
    ↪ The gradient is split into $H$ discrete orientations, so each single histogram
    ↪ has $H$ entries.
#
# Algortihm uses $Q$ rings around the central point, on which the COMs are sampled, each
    ↪  ring has $T$ histograms.
#
# Therefore, each descriptor has $D_s = (Q \cdot T + 1) \cdot H$ entries.
#
# To compute GOs at a specified point $(u, v)$, several oriented derivatives of image $I$
    ↪ are computed as following:
#
# $G_o^\sigma = G^\sigma * \left(\dfrac{dI}{do}\right)^+ , \left(\dfrac{dI}{do}\right)
    ↪ ^+ = max\left(\dfrac{dI}{do}, 0\right),$
# where $G^\sigma$ is a Gaussian kernel with standard deviation $\sigma$ and $o$ is a derivative
    ↪ orientation.
#
# The results $G_o^\sigma$ are referred as Convolved Orientation Maps (COMs).
#
# In the next step the vector $h_\sigma(u, v)$ is being built as following:
#
# $h_\sigma(u,v) = \left[G_1^\sigma(u,v), \dots, G_H^\sigma(u,v)\right]^T$
#
# Vector $h(u, v)$ is then normalized to unit norm. It represents the values of all the
    ↪ GOs at a point $(u, v)$ after convolution with a Gaussian kernel with a standard
```

```
    ↪ deviation σ.
#
# The full DAISY descriptor for location (u_0, v_0) is defined as a simple concatenation
    ↪ of all the vectors h_{σi}(u, v) for all points beginning with the center:
#
# D(u_0,v_0) = [h_{\sigma 1}(u_0, v_0), h_{\sigma 1}(I_1(u_0, v_0, R_1)), \dots, h_{\
    ↪ sigma 1}(I_T(u_0, v_0, R_1),
# h_{\sigma 2}(I_1(u_0, v_0, R_2)), \dots, h_{\sigma 2}(I_T(u_0, v_0, R_2),
# \dots \dots
# h_{\sigma Q}(I_1(u_0, v_0, R_Q)), \dots, h_{\sigma Q}(I_T(u_0, v_0, R_Q)],
#
# where I_j(u, v, R) is the location with distance R from (u, v) in the direction given by
    ↪ j
# when the directions are quantized into the T values.

# In[31]:


import numpy as np

from skimage import data
from skimage.color import rgb2gray
from skimage.filters import sobel_h, sobel_v, gaussian
from skimage.feature import match_descriptors
from skimage.exposure import rescale_intensity
from skimage.util import img_as_float, img_as_ubyte
from skimage import transform
from sklearn import preprocessing

import matplotlib.pyplot as plt
import matplotlib.ticker as mtick
import seaborn as sb


from skimage.draw import (line, circle_perimeter)
from enum import Enum

get_ipython().run_line_magic('matplotlib', 'inline')
```

```
# ## DAISY implementation
# Next function computes a single Gradient Orientation Map. As image derivative we use
    ↪ a Sobel filter.
# Standard edge-detection filter can work only in x or y axis.
# However, method, known as steerable filter, allows us to compute derivative at any
    ↪ virtual axis z at some angle θ:
# d_z = \left(d_x \cos\theta + d_y \sin\theta\right)^+

# In[2]:


# Computes a Gradient Orientation Map on derivative axis
# with angle tetha
def gradient_orientation(image, tetha):
    # Steering filter - combination of X and Y derivatives
    # to det a derivative on a specific angle
    d_hor = sobel_h(image)
    d_ver = sobel_v(image)
    derivative = d_hor * np.cos(tetha) + d_ver * np.sin(tetha)
    # Operator max(d, 0) levaes only positive derivative
    derivative = derivative.clip(min=0)
    return derivative


# Next function computes a set of GO's for a given number of directions. Directions
    ↪ are arranged evenly in a circle.

# In[3]:


# Compute GOs for a number of directions
def compute_global_orientation_maps(image, H):
    tetha_step = 2 * np.pi / H
    result = []
    for k in range(0, H):
        tetha = tetha_step * k
        result.append(gradient_orientation(image, tetha))
    return result
```

```
# Next function convolves a set of GO's with several Gaussian kernels to produce
    ↪ Convolved Orientation Maps for each level of Gaussian standard deviation σ.

# In[4]:


# Convolves orientation maps with Gaussian
# kernels with specified standard deviations
def convolve_orientation_maps(gos, sigmas):
    result = []
    for sigma in sigmas:
        conv = []
        for go in gos:
            conv.append(gaussian(go, sigma))
        result.append(conv)
    return result


# DAISY descriptor can be normalized in several different manners:
# * No normalization at all
# * Normalizing each h vector to unit norm separately
# * Normalazing full descriptor to unit norm
# * SIFT-style full descriptor normalization with no individual element bigger than
    ↪ 0.154 and sum equals to 1

# In[5]:


class NormType(Enum):
    NRM_NONE = 1
    NRM_PARTIAL = 2
    NRM_FULL = 3
    NRM_SIFT_STYLE = 4


# Here is the main DAISY function. For illustrative purposes, we compute only one
    ↪ descriptor, at the specified point of the image. Function returns descriptor
    ↪ vector Ds, visualization of a DAISY pattern, and a set of computed COM's to
    ↪ illustrate it's work.
```

```
# In[6]:


# Calculation of a single DAISY descriptor for a given point of an image
def daisy(image, center, R, Q, T, H, sigmas, norm=NormType.NRM_NONE):

    # Setting steps in angle and radius
    tetha_step = 2 * np.pi / T
    radius_step = float(R) / Q
    # Calculating total number of entries in a descriptor
    Ds = (Q*T+1)*H
    descriptor = np.zeros((1, Ds))

    # Copy of image for visualization
    visualize = np.copy(image)

    # Compute GOs
    gradient_orientations = compute_global_orientation_maps(image, H)
    # Convolve GOs with Gaussian kernels to get COMs
    conv_orient_maps = convolve_orientation_maps(gradient_orientations, sigmas)

    # Sampling vectors from COMs at DAISY points
    # Center point
    h = sample_h_vector(center[0], center[1], conv_orient_maps[0], visualize, sigmas
        ↪ [0])
    descriptor[0, 0:H] = h
    # Sample the rest of the points in DAISY pattern
    for i in range(0, Q):
        r = (i+1) * radius_step
        maps = conv_orient_maps[i]
        for j in range(0, T):
            tetha = j * tetha_step
            x = int(center[0] + r * np.cos(tetha))
            y = int(center[1] + r * np.sin(tetha))

            h = sample_h_vector(x, y, maps, visualize, sigmas[i+1])
            # Normalization for partial norm type
            if norm == NormType.NRM_PARTIAL:
                h = preprocessing.normalize(h.reshape((1, -1)), norm='l2')
```

```
            idx_start = ((i)*T+j)*H+H
            idx_end = ((i)*T+j)*H+2*H
            descriptor[0, idx_start:idx_end] = h

    # Normalization for full norm type
    if norm == NormType.NRM_FULL:
        descriptor = preprocessing.normalize(descriptor.reshape((1, -1)), norm='l2')
    # Normalization for SIFT-style norm
    if norm == NormType.NRM_SIFT_STYLE:
        descriptor = preprocessing.normalize(descriptor.reshape((1, -1)), norm='l2')
        descriptor = descriptor.clip(max=0.154)
        descriptor /= descriptor.sum()

    return descriptor, visualize, gradient_orientations, conv_orient_maps

# Samples the single h vector at point
def sample_h_vector(x, y, maps, visualize, sigma):
    # Sampling the vector
    h = np.zeros(H)
    for k in range(0, H):
        h[k] = maps[k][x, y]
    # Visualize the point
    draw_circle(visualize, x, y, sigma)
    draw_lines(visualize, x, y, sigma, H)
    return h


# #### Utility functions
# Functions for data visualization

# In[95]:


# Draws a circle
def draw_circle(image, x, y, radius):
    rr, cc = circle_perimeter(x, y, radius)
    image[rr, cc] = 1

# Draws a set of evenly arranged lines from center
def draw_lines(image, x, y, radius, n):
```

```
    tetha_step = 2 * np.pi / n
    for i in range(0, n):
        tetha = i * tetha_step
        line_len = int(radius * 0.7)
        x1 = int(x + line_len * np.sin(tetha))
        j1 = int(y + line_len * np.cos(tetha))
        rr, cc = line(x, y, x1, j1)
        image[rr, cc] = 1

# Shows an image
def imgshow(image, title='', cmap=plt.cm.gray):
    fig, ax = plt.subplots()
    ax.imshow(img_as_ubyte(image), cmap=cmap)
    ax.set_title(title)
    plt.show()

# Shows a set of images in a grid
def show_maps(maps, title=''):
    grid_i = 2
    grid_j = int(np.ceil(len(maps) / 2))
    fig, ax = plt.subplots(grid_i, grid_j)
    fig.suptitle(title)
    for i in range(0, grid_i):
        for j in range(0, grid_j):
            idx = i*grid_j+j
            if(idx >= len(maps)):
                break
            ax[i, j].set_title(idx)
            ax[i, j].imshow(maps[idx], cmap = plt.cm.plasma)
    plt.tight_layout()

# Simple XY 2d plot
def plot_xy(x, y, title='', xlabel='', ylabel=''):
    fig, ax = plt.subplots()
    ax.plot(x, y)
    ax.grid()
    ax.set_title(title)
    ax.set_ylabel(ylabel)
    ax.set_xlabel(xlabel)
    plt.tight_layout()
```

```
    plt.show()


# ## Test data preparation
# We use a 128 * 128 sample of a contrast grayscale image from skimage.data

# In[131]:


img_src = rgb2gray(img_as_float(data.astronaut()))

print(img_src.shape)

#center = [img_src.shape[1] / 2, img_src.shape[0] / 2]
center = [128, 256]
size = [256, 256]

image = img_src[int(center[0]-size[0] / 2):int(center[0]+size[0] / 2),
          int(center[1]-size[1] / 2):int(center[1]+size[1] / 2)]
imgshow(image, 'Source image')
print(f'Image size:{image.shape}')


# ## Testing section
# Here is the list of modifiable parameters:
# * Maximum radius R
# * Number of rings Q
# * Number of points per ring T
# * Number of histograms per point H
# * Gaussian filter standard deviations σ for each ring
# * Normalization type
#
# **Notice that big sigma values or radius may result in index out of bounds exception
    ↪ !**

# In[155]:


R = 30
Q = 2
```

```
T = 8
H = 8
SIGMAS = [5, 10, 25]
NORM = NormType.NRM_SIFT_STYLE


# In[156]:


# Computing one descriptor at the center of the image
x, y = image.shape
point = [int(x/2), int(y/2)]

descriptor, visualize, gradient_orientations, coms = daisy(image, point, R, Q, T, H,
    ↪ SIGMAS, NORM)
imgshow(visualize, title='Pattern visualization')


# Here we can explore the algorithm data. Let's check gradient orientation maps:

# In[151]:


show_maps(gradient_orientations, title='Gradient orientations')


# The convolved orientation maps for each ring:

# In[152]:


for i in range(len(SIGMAS)):
    show_maps(coms[i], title=f'COMs for sigma={SIGMAS[i]}')


# ### Resulting descriptor
# Resulting descriptor vector is consists of D_s elements:

# In[153]:
```

```
print(descriptor, '\n')
print(f'Descriptor magnitude: {np.linalg.norm(descriptor)}')
print(f'Descriptor sum: {descriptor.sum()}')


# ## Testing descriptor invariancy to image transformations
# From here we will test descriptors invariancy to image transformations, such as
    ↪ rotation, scale, and contrast.
# ### Testing effects of rotation
# Let's start with rotation. We will rotate image by certain angle up to 360°, and on
    ↪ each step compare descriptor for it's center point with initial by calculating
    ↪ mean squared error.

# In[161]:


n = 32
tethas = np.zeros(n)
mean_stds = np.zeros(n)
tetha_step = 360 / n

point2 = [point[0], point[1]]

for i in range(0, n):
    tetha = i * tetha_step
    image2 = transform.rotate(image, tetha)
    descriptor2, visualize2, gradient_orientations2, coms2 = daisy(image2, point2, R, Q
        ↪ , T, H, SIGMAS, NORM)
    desc_norm = max(np.max(descriptor), np.max(descriptor2))
    mean_stds[i] = (((descriptor - descriptor2) / desc_norm) ** 2).mean()
    tethas[i] = tetha
    #imgshow(visualize2)

plot_xy(tethas, mean_stds,
        title='Invariancy to rotation',
        xlabel='Rotation, deg.',
        ylabel='Descriptors mean squared error')
```

```
# ### Testing effects of scaling
# To test invariancy to scale, we will rescale image up to a certain factor and on
    ↪ each step compare descriptor for it's center point with initial by calculating
    ↪ mean squared error.

# In[159]:


max_scale = 6
n = 6
scale_levels = np.zeros(n)
mean_stds = np.zeros(n)
scale_step = max_scale / n
for i in range(0, n):
    scale = 1 + i * scale_step
    image2 = transform.rescale(image, scale, multichannel=False)
    descriptor2, visualize2, gradient_orientations2, coms2 = daisy(image2, point, R, Q,
        ↪ T, H, SIGMAS, NORM)
    desc_norm = max(np.max(descriptor), np.max(descriptor2))
    mean_stds[i] = (((descriptor - descriptor2) / desc_norm) ** 2).mean()
    scale_levels[i] = scale

plot_xy(scale_levels, mean_stds,
       title='Invariancy to scale',
       xlabel='Scale, times.',
       ylabel='Descriptors mean squared error')


# ### Testing effects of intensity changes
#
# *This part needs additional work*
#
# To test invariancy changes in intensity, we will shrink image dynamic diapazone and
    ↪ on each step compare descriptor for it's center point with initial by
    ↪ calculating mean squared error.
#
# For example, image with shrinked down dynamic diapazone looks like this:

# In[147]:
```

```
image2 = rescale_intensity(image, (0.45, 0.55))
imgshow(image2)


# In[157]:


initial_intensity = [0, 1]
n = 5
int_levels = np.zeros(n)
mean_stds = np.zeros(n)
int_step = initial_intensity[1] / 2 / n
for i in range(0, n):
    intenity = i * int_step
    low = initial_intensity[0]+intenity
    high = initial_intensity[1]-intenity
    image2 = rescale_intensity(image, (low, high))
    descriptor2, visualize2, gradient_orientations2, coms2 = daisy(image2, point, R, Q,
        ↪ T, H, SIGMAS, NORM)
    desc_norm = max(np.max(descriptor), np.max(descriptor2))
    mean_stds[i] = (((descriptor - descriptor2) / desc_norm) ** 2).mean()
    perc = float(high - low)
    int_levels[i] = perc

plot_xy(int_levels, mean_stds,
       title='Invariancy to intensity changes',
       xlabel='Intensity diapazone, perc.',
       ylabel='Descriptors mean squared difference')


# ## Next steps
# We will add more examples and tests for descriptors, and test on a greater scale
    ↪ with large numbers of points.
```