РЕФЕРАТ

Отчет: 61 страницы, 22 рисунка, 2 таблицы, 10 источников, 1 приложение.

ОБРАБОТКА ИЗОБРАЖЕНИЙ, ДЕСКРИПТОРЫ ИЗОБРАЖЕНИЙ, DENSE MATCHING, DAISY, 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 корректностю 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 алгоритма0 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 Обзор задачи

1.1 Общие сведения о дескрипторах изображений

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.

Как правило, дескриптор представляет собой векторное значение, вычисляемое определенной функцией f от точки изображения x.

Рассмотрим следующую постановку задачи. Дано изображение:

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

где d – размерность, для обычных плоских изображений равная 2.

Построение дескриптора задается следующим образом:

$$\vec{v} = f(I(x)), \quad x \in S. \tag{2}$$

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

$$f(I_i(x_i)) = f(I_j(x_j)), \quad i = 1 \dots N, \quad j = 1 \dots N, \tag{3}$$

где x_i, x_j – пара соответствующих точек изображений I_i, I_j.

Таким образом, задача совмещения изображений сводится к нахождению пространственного преобразования g и преобразования яркости f, которые дают минимум некой целевой функции, задающей критерий качества. Различные оценки качества определяют специфические для конкретной задачи стратегии поиска.

Существуют различные способы поиска пространственного преобразования, в том числе попиксельное сравнение полным перебором, подбор значений с использованием оптимизационных стратегий и так далее. Наиболее распространенным на практике и эффективным является метод построения пространственного преобразования по опорным точкам, выбранный для изучения и реализации в рамках данной работы.

Кратко общий алгоритм работы данного метода можно описать следующим образом:

1. Производится поиск пар совпадающих точек x_i, x'_i на двух изображениях.

2. По координатам x_i, x'_i вычисляется матрица, задающая преобразование $g : x'_i = g(x_i)$.

3. Преобразование g применяется к пикселям изображения I_1, трансформируя его до совпадения соответствующих точек с I_2.

4. Изображения I_1 и I_2 подвергаются преобразованию яркости f и складываются, формируя результат совмещения.

В данной работе основное внимание уделяется искажениям пространственного преобразования, вызванными цифровыми шумами, поэтому преобразования яркости подробно рассмотрены не будут. Разберем основные этапы алгоритма и способы их реализации.

1.2 Методы поиска опорных точек

Как было описано, опорными точками для вычисления параметров пространственного преобразования в задаче сшивки должны выступать некие однозначно идентифицируемые области исходных изображений [1]. В качестве таковых используются наборы особых точек, найденные для

используемых изображений одним из существующих алгоритмов. Особыми точками называют уникальный набор точек изображения, позволяющий однозначно его характеризовать. Как правило, это области углов крупных контрастных объектов, границы изменения уровня яркости, цвета и так далее. Критерии для определения особых точек подбираются так, чтобы с наибольшей вероятностью определить аналогичные точки на различных изображениях. Важнейшим требованием к алгоритмам поиска особых точек является инвариантность определяемых наборов относительно преобразований изображения, в первую очередь, сдвига, масштабирования и поворота.

Рисунок 1 – Пример наборов особых точек на фотографиях

Для однозначного определения особой точки, инвариантного относительно ее координат в изображении, используются особые идентификаторы, называемые дескрипторами особых точек. Итак, алгоритм поиска должен обеспечивать вычисление достаточного набора дескрипторов особых точек, инвариантных к преобразованиям изображения, в первую очередь, пространственным. Далее будет дан общий обзор нескольких часто используемых на практике алгоритмов вычисления дескрипторов, реализации которых присутствуют в библиотеке компьютерного зрения OpenCV.

Алгоритм SIFT (Scale Invariant Feature Transform) был предложен Дэвидом Лоу в 2004 году [2]. Дескрипторы SIFT устойчивы к изменению масштаба, аффинным преобразованиям и смене проекции исходных изображений. Алгоритм показал высокую эффективность, однако имеет высокую вычислительную сложность, что может являться проблемой при использовании его в режиме реального времени. SIFT включает как сам поиск точек, так и построение их дескрипторов.

Алгоритм состоит из четырех основных шагов. На первом шаге производится выделение экстремумов изображения при помощи фильтра разности гауссианов (Difference of Gaussian). На втором шаге локализуются ключевые точки путем выявления наиболее контрастных областей. Далее, оценивается ориентация ключевых точек на основе локального градиента. На последнем этапе составляются дескрипторы, основанные на магнитуде и направлении градиента точек.

Рисунок 2 – Пример работы SIFT

Алгоритм SURF (Speeded up Robust Features) строит дескрипторы, инвариантные к изменению масштаба и вращения. SURF работает быстрее SIFT при незначительном снижении качества обнаружения.

Для поиска экстремумов алгоритм аппроксимирует разность гауссианов матричными фильтрами, что позволяет увеличить производительность за счет высокой скорости самой операции свертки и легкости реализации параллельных вычислений. SURF использует детектор BLOB (Binary

Large Objects) - крупных контрастных объектов [8], основанный на матрице Гессе, для поиска ключевых точек. Вычисление ориентации точек и построение дескрипторов основывается на вейвлет-образах (wavelet responses) окрестности точки.

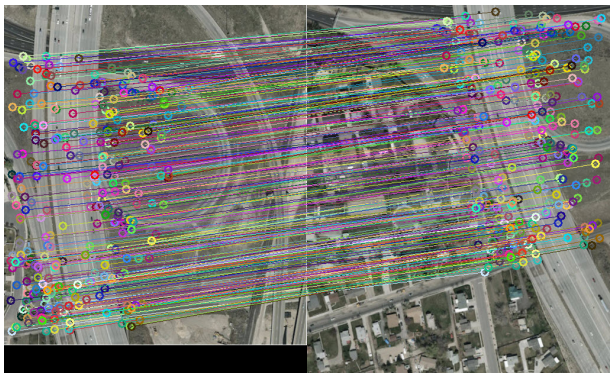

Рисунок 3 – Пример работы SURF

Алгоритм ORB (Oriented FAST and Rotated BRIEF) Эффективной альтернативной SIFT и SURF является ORB, комбинация детектора особых точек FAST и дескриптора BRIEF с некоторыми модификациями [3].

Первоначальный набор точек ищется с помощью FAST, после чего применяется детектор углов Харриса для нахождения наиболее интересных из них. FAST не вычисляет ориентацию точки, и, следовательно, не инвариентен к поворотам изображения. Для устранения этого фактора, для каждого найденного угла вычисляется точка наибольшей интенсивности в окрестности, и вектор из цетра угла до этой точки используется для описания ориентации. Некоторые дополнительные модификации позволяют получить достаточно инвариантные к преобразованиям дескрипторы точек.

Рисунок 4 – Пример работы ORB

1.3 Пространственные преобразования изображений

Для работы с изображениями из существующих видов пространственных преобразований, как правило, используют три их группы: преобразования подобия, аффинные и проективные преобразования.

Для рассмотрения и практического применения в данной работе наиболее интересными являются проективные преобразования, являющиеся общим случаем пространственных преобразований [4]. Практическую ценность представляет то, что проективные преобразования позволяют бороться с перспективными искажениями - изменением масштаба объектов на фотографиях, возникающим, если снимаемые объекты находятся не плоскости, перпендикулярной направлению камеры, либо у края кадра при использовании широкого поля зрения.

Проективные преобразования отображают проекцию двумерного изображения на плоскость в трехмерном пространстве. Отличительной особенностью является сохранение коллинеарности точек, т.е. точки, лежавшие на одной прямой, после преобразования остаются на ней. Отличительной особенностью является несохранение параллельности прямых линий, а также изменение размера объектов в зависимости от расстояния от центра проекции.

Преобразование над изображением I задается как операция над каждым его пикселем с координатами $\begin{bmatrix} x & y \end{bmatrix}$, преобразующая данные коорди-

наты к $\begin{bmatrix} x' & y' \end{bmatrix}$. Изменение координат задается матрицей размерности 3x3 следующего вида:

$$H = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix}.$$

Операция над координатами пикселей выполняется в виде умножения координатного вектора на матрицу преобразования. Очевидным образом, для работы с матрицей размерности 3x3, к координатному вектору положения точки $\begin{bmatrix} x & y \end{bmatrix}$ добавляется третья координата, равная 1 и не влияющая на результат.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{00} & h_{01} & h_{02} \\ h_{10} & h_{11} & h_{12} \\ h_{20} & h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

В соответствии с правилами матричного умножения, координаты пикселей вычисляются по следующим формулам:

$$x' = h_{00}x + h_{01}y + h_{02}, \qquad y' = h_{10}x + h_{11}y + h_{12}. \tag{4}$$

Отметим, что использование в качестве матрицы преобразования единичной матрицы размерности 3x3 оставляет исходное изображение без изменений. На рисунке 5 приведен пример проективных преобразований изображения.

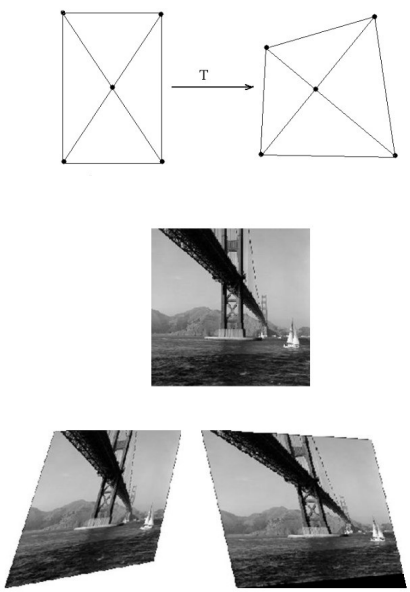

Рисунок 5 – Пример проективных преобразований

Подмножеством проективных преобразований являются аффинные преобразования. Их отличительной чертой выступает сохранение параллельности линий на изображении [5]. При помощи аффинных преобразований выполняется смещение, поворот и масштабирование.

Операция смещения на e по оси x и на f по оси y задается следующей матрицей преобразования:

$$H_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ e & f & 1 \end{bmatrix}.$$

Операция поворота на угол α задается матрицей поворота:

$$H_r = \begin{bmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Операция масштабирования с коэффициентом a по оси x и d по оси y за-

дается следующей матрицей:

$$H_s = \begin{bmatrix} a & 0 & 0 \\ 0 & d & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Рассмотрим нахождение пространственного преобразования по опорным точкам. Напомним, что система уравнений, описывающих его, записана в выражении (4). Приведем пример вычисления коэффициентов матрицы $h_{00} \dots h_{12}$ для простейшего случая с использованием минимально необходимых трех пар соответствующих ключевых точек. Система соответствующих уравнений в матричной форме запишется как:

$$\begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix} = A \begin{bmatrix} h_0 0 \\ h_0 1 \\ h_0 2 \end{bmatrix},$$

$$\begin{bmatrix} y'_1 \\ y'_2 \\ y'_3 \end{bmatrix} = A \begin{bmatrix} h_1 0 \\ h_1 1 \\ h_1 2 \end{bmatrix}.$$

где A - матрица следующего вида:

$$A = \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}.$$

Для получения значений коэффициентов домножим каждую часть обоих систем на обратную матрицу A^{-1}. Таким образом, получим следующие выражения для вычисления коэффициентов в матричной форме:

$$\begin{bmatrix} h_0 0 \\ h_0 1 \\ h_0 2 \end{bmatrix} = A^{-1} \begin{bmatrix} x'_1 \\ x'_2 \\ x'_3 \end{bmatrix},$$

$$\begin{bmatrix} h_1 0 \\ h_1 1 \\ h_1 2 \end{bmatrix} = A^{-1} \begin{bmatrix} y'_1 \\ y'_2 \\ y'_3 \end{bmatrix}.$$

Вычисленный набор коэффициентов подставляется в выражение преобразования и используется для произведения операции с каждым пикселем исходного изображения. При использовании большего числа пар опорных точек, чем три, значения усредняются, что позволяет уменьшить неизбежно возникающие на практике погрешности.

2 Модели цифровых шумов в обработке изображений

Цифровым шумом изображения называют дефекты, вносимые в изображение спецификой конструкции и несовершенством цифровых фотосенсоров, электроники фото- и видеотехники, их использующей, а также фотонной природой света. Цифровой шум визуально проявляется отдельными элементами изображения, имеющими размеры в один или несколько пикселей, отличающимися от основного изображения своей яркостью (яркостной шум, luminance noise) и (или) цветом (chrominance noise, хроматический шум).

По способу влияния на изображение цифровые шумы разделяются на три группы:

1. Аддитивные – значение шума складывается со значением сигнала. Производят пиксели изображения, отличающиеся по яркости и (или) цвету от исходного на некоторое значение. Величина шума не зависит от величины сигнала.

2. Мультипликативные – значение шума умножается на значение сигнала. Производят пиксели изображения, отличающиеся по яркости и (или) цвету от исходного на определенную долю. Величина шума зависит от величины сигнала.

3. Импульсные – замещают случайные пиксели исходного изображения точками со значительными скачками яркости и (или) цвета. Величина шума от величины сигнала не зависит.

Рассмотрим несколько математических моделей цифрового шума, стандартно используемых для оценки качества алгоритмов обработки изображений.

Гауссовский шум – случайный аддитивный шум с независимыми значениями для каждого пикселя, не зависящий от их исходной яркости. Плотность распределения вероятности гауссовского шума задается нормальным распределением:

$$p_G(z) = \frac{1}{\sigma\sqrt{2\pi}} \mathrm{e}^{-\frac{(z-\mu)^2}{2\sigma^2}},$$

где z – яркость пикселя;

μ – среднее значение распределения;

σ – среднеквадратичное отклонение.

Гауссовский шум используется в качестве стандарта белого шума при тестировании цифрового оборудования и алгоритмов. Применительно к фотографиям данная модель шума имитирует помехи, возникающие при недостаточной освещенности или экспозиции. Пример представлен на рисунках 6 - 7.

Рисунок 6 – Пример гауссовского шума с $\sigma = 30$

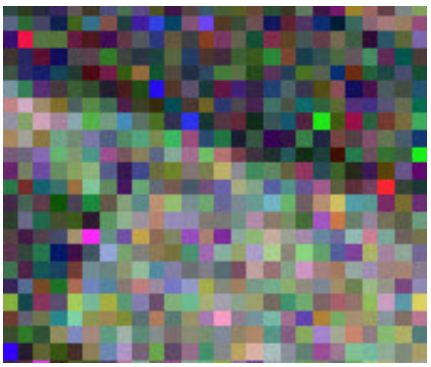

Рисунок 7 – Участок изображения при увеличении

Шум соль-перец – случайный импульсный шум с независимыми значениями для каждого пикселя. Характеризуется случайно распределенными по изображению точками черного и белого цвета. В данной работе использовалась реализация с одним параметром вероятности возникновения шума p и определением черного или белого цвета с вероятностями p и $1-p$ соответственно. Пример представлен на рисунках 8 - 9.

Рисунок 8 – Пример шума соль-перец с $p = 0{,}05$

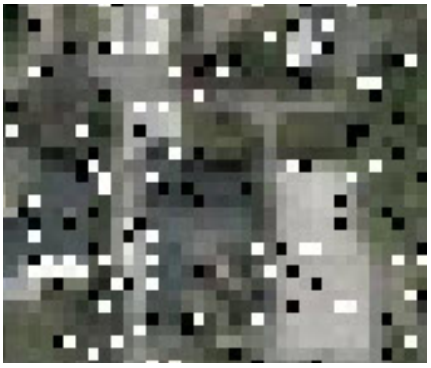

Рисунок 9 – Участок изображения при увеличении

Размытие – строго говоря, не являясь цифровым шумом, представляет собой часто встречающийся дефект изображения, возникающий из-за неправильной фокусировки. В данной работе размытие интересно тем, что обладает свойством сглаживать высокочастотные компоненты изображения (мелкие детали), серьезно влияя на определение особых точек. Также, размытие является статичным процессом без использования случайных значений, что позволяет легко воспроизводить результаты. На практике, как правило, используется реализация размытия в виде фильтра Гаусса – матричного фильтра, усредняющего значений соседних пикселей в соответствии с нормальным распределением. Формула размытия является двумерной функцией Гаусса:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \mathrm{e}^{-\frac{x^2 + y^2}{2\sigma^2}},$$

где x, y – расстояния по осям от центра;

σ – среднеквадратичное отклонение.

Пример представлен на рисунках 10 - 11.

Рисунок 10 – Пример размытия изображения с $m = 21$

Рисунок 11 – Участок изображения при увеличении

Как было сказано ранее, дескрипторы при поиске особых точек определяются как функции от яркости определенной окрестности пикселя. Зная это, можно понять, что зашумленность изображения будет вызывать сбои при вычислении дескрипторов, приводя к уменьшению количества найденных совпадений, или к искажению координат точек. В предельном случае, когда необходимое для построения матрицы проективного преобразования количество особых точек не может быть найдено, высокая зашумленность изображения полностью блокирует работу алгоритма. В остальных случаях ошибки в вычисленных дескрипторах будут приводить к смещению изображения относительно требуемого после трансформации положения, и, соответственно, к визуально заметным неточностям.

3 Экспериментальное исследование

3.1 Реализация алгоритма сшивки

Алгоритм сшивки изображений для исследования был реализован на языке Python 3.7 с использованием библиотеки OpenCV 3.4.2.

Функция, осуществляющая процедуру совмещения, принимает на вход два изображения и тип используемого дескриптора. Используются дескрипторы SIFT, SURF, ORB, BRIEF. Заметим, что алгоритмы SIFT и SURF являются патентованными, и не присутствуют в стандартной библиотеке OpenCV из-за лицензионных ограничений. Данные алгоритмы доступны в пакете xfeatures2d при использовании версии библиотеки, включающей неофициальные алгоритмы, opencv-contrib. Использование данных алгоритмов разрешается только в некоммерческих целях [11].

Для вычисления дескрипторов производится преобразование исходных изображений в градации серого.

```
# convert the image to grayscale
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

Производится поиск наборов особых точек при помощи выбранного дескриптора.

```
# detect keypoints in the image
detector = cv2.FeatureDetector_create("SIFT")
kps = detector.detect(gray)

# extract features from the image
extractor = cv2.DescriptorExtractor_create("SIFT")
(kps, features) = extractor.compute(gray, kps)
```

Найденные наборы проверяются на совпадение дескрипторов методом knn. Полученные совпадения дополнпительно тестируются на предмет нахождения расстояния между точками в заданных пределах. Пары, прошедшие тест, используются для вычисления преобразования.

```
# compute the raw matches and initialize the list of actual
# matches
matcher = cv2.DescriptorMatcher_create("BruteForce")
rawMatches = matcher.knnMatch(featuresA, featuresB, 2)
matches = []

# loop over the raw matches
for m in rawMatches:
# ensure the distance is within a certain ratio of each
# other (i.e. Lowe's ratio test)
if len(m) == 2 and m[0].distance < m[1].distance * ratio:
matches.append((m[0].trainIdx, m[0].queryIdx))
```

После нахождения совпавших точек, по их наборам для каждого изображения вычисляется матрица проективного преобразования. Если совпавших опорных точек найдено меньше, чем четыре, проективное преобразование не может быть вычислено, и функция возвращает статус ошибки.

```
 # compute the homography between the two sets of points
(H, status) = cv2.findHomography(ptsA, ptsB, cv2.RANSAC,
reprojThresh)
```

Построенная матрица проективного преобразования применяется ко второму входному изображению, трансформируя его до совпадения положений опорных точек.

Исходное первое изображение и преобразованное второе записываются в одно результирующее. В зависимости от требуемых параметров, сложение может производиться с различными значениями альфа-канала, в том числе с выделением яркостью пересекающейся области либо одного из изображений в целях повышения наглядности.

Функция возвращает результирующее изображение и отчет о работе алгоритма, включающий время выполнения и построенную матрицу проективного преобразования.

Полный код функции представлен в приложении А.

3.2 Экспериментальная оценка влияния цифровых шумов на качество работы алгоритма

Выбранные для исследования цифровые шумы реализованы в виде отдельных функций, принимающих на вход изображение, подлежащее зашумлению, и параметры шума. Исходный код функций зашумления представлен в приложении А.

В качестве оценочных параметров точности работы алгоритма были выбраны общее количество найденных особых точек, процент совпадений от общего числа и смещение преобразованного изображения относительно истинного положения в результате неточности построенной матрицы проективного преобразования. Количество найденных точек возвращается функцией сшивки после успешного завершения работы. Смещение относительно истинного положения вычисляется по координатам углов исходного и преобразованного изображений, оценочными параметрами смещения выступают медиана и максимум расстояний между соответствующими углами в пикселях.

Результаты экспериментов сохраняются в виде текстовых отчетов и (или) изображений, доступных для дальнейшего анализа и визуализации.

Для приведенных экспериментов вторым изображением, поступающим на вход алгоритма сшивки, выступала программно вырезанная прямоугольная часть центральной области первого изображения, равная 25 процентам его площади. Такой подход, в отличии от аналогичного практическому применению алгоритма использования частично пересекающихся фотографий, позволяет гарантировать нахождение максимального принципиально возможного количества совпадающих точек, а также упростить расчет смещения и визуализацию.

3.1.1 Сравнение работы алгоритмов при применении размытия

Размытие изображения, строго говоря, не является цифровым шумом, хотя и может быть достаточно распространенным дефектом фотографии. Тем не менее, размытие хорошо подходит для общего сравнения алгоритмов, так как является стационарным и легко воспроизводимым процессом понижения качества картинки.

В данном эксперименте входное изображение размывалось с итеративным увеличением размера ядра гауссовского фильтра m. Эксперимент прекращался, когда число найденных опорных точек становилось меньше четырех. Входные изображения приводились к размеру в 512 пикселей по горизонтали. На рисунках 12-13 представлены примеры результатов работы программы.

Рисунок 12 – Пример отображения программой совпадений особых точек на размытом изображении

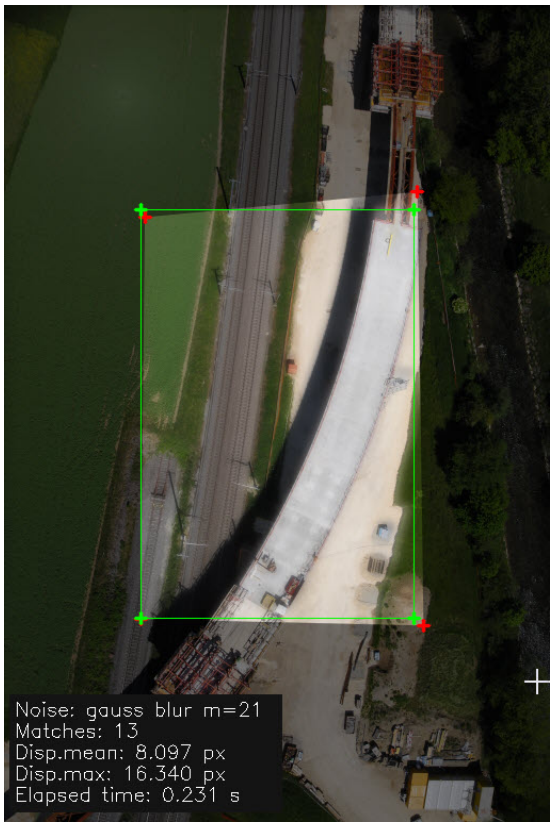

Рисунок 13 – Пример отображения программой результата совмещения

Таблица 1 – Сравнение числа найденных точек при увеличении уровня размытия

m	Алгоритмы		
	SIFT	SURF	ORB
-	496	996	1286
3	276	481	685
7	93	187	237
11	51	87	106
17	25	48	23
19	21	42	17
21	16	30	9
27	9	22	-
35	6	16	-
41	7	8	-
49	5	6	-
53	5	-	-

Прочерком обозначены значения, при которых алгоритм находит менее четырех совпадающих опорных точек.

Таблица 2 – Сравнение медианы смещения при увеличении уровня размытия

m	Алгоритмы		
	SIFT	SURF	ORB
-	0,032	0,000	0,130
3	0,082	0,085	0,163
7	0,146	0,170	2,670
11	0,270	0,303	1,953
17	0,917	1,100	2,852
19	0,405	1,476	15,543
21	2,243	2,893	336,972
27	5,564	1,012	-
35	13,711	5,951	-
41	44,210	10,701	-
49	84,046	22,116	-
53	120,100	-	-

В таблице 2 цветом выделены значения медианы смещения в более чем 10 пикселей, как выбранный критерий хорошо различимого визуально показателя недостаточной точности. Прочерком обозначены значения, при которых алгоритм находит менее четырех совпадающих опорных точек.

Результаты эксперимента отражены на рисунке 14.

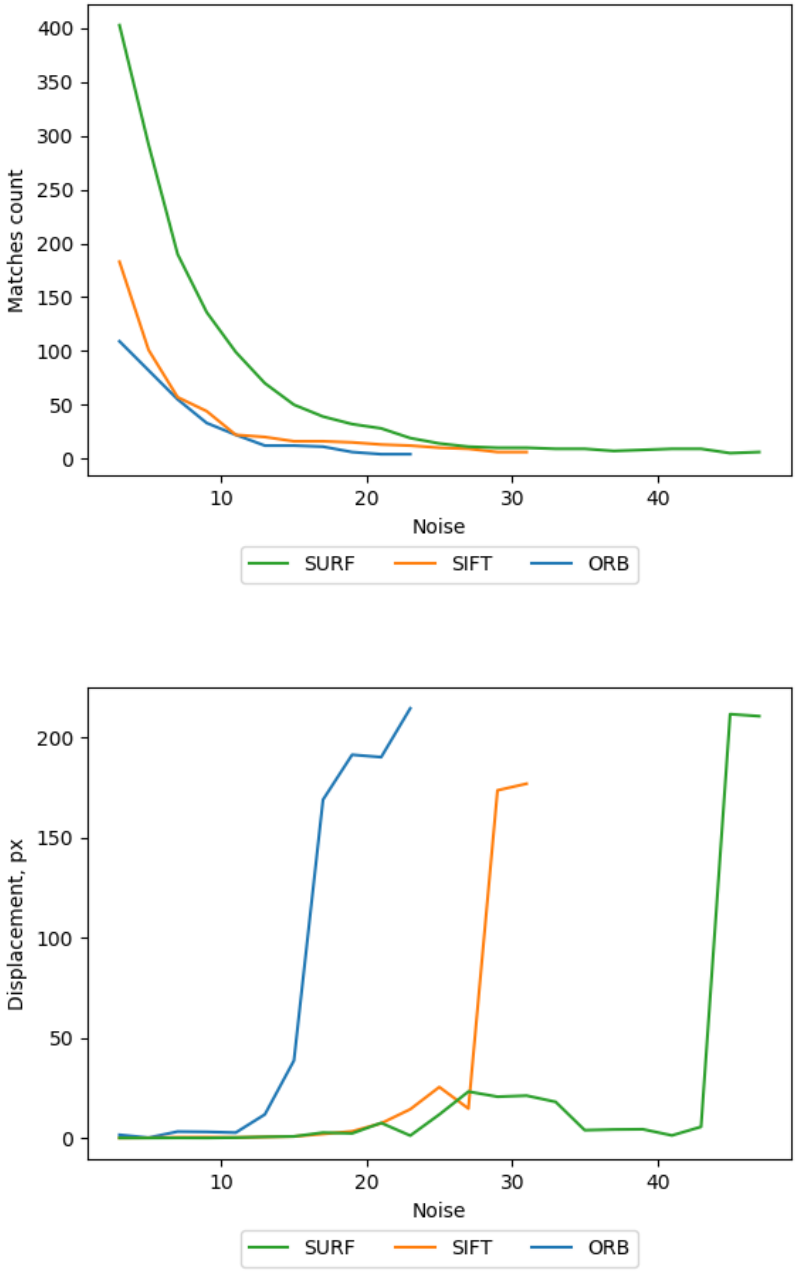

Рисунок 14 – Поведение числа совпадений точек и смещения при размытии изображения

3.1.2 Сравнение работы алгоритмов при применении случайных шумов

При использовании случайных шумов эксперимент производился несколько раз для каждого уровня зашумленности с усреднением результатов для устранения флуктуаций и получения более общей картины.

Для каждого уровня зашумленности значения усреднялись по 30 экспериментам. Варьируемыми параметрами шумов выступали стандартное отклонение σ для гауссовского шума и вероятность p для шума соль-и-перец. Эксперимент прекращался, когда число найденных опорных точек становилось меньше четырех. Входные изображения приводились к размеру в 512 пикселей по горизонтали.

Результаты экспериментов отражены на рисунках 15-16.

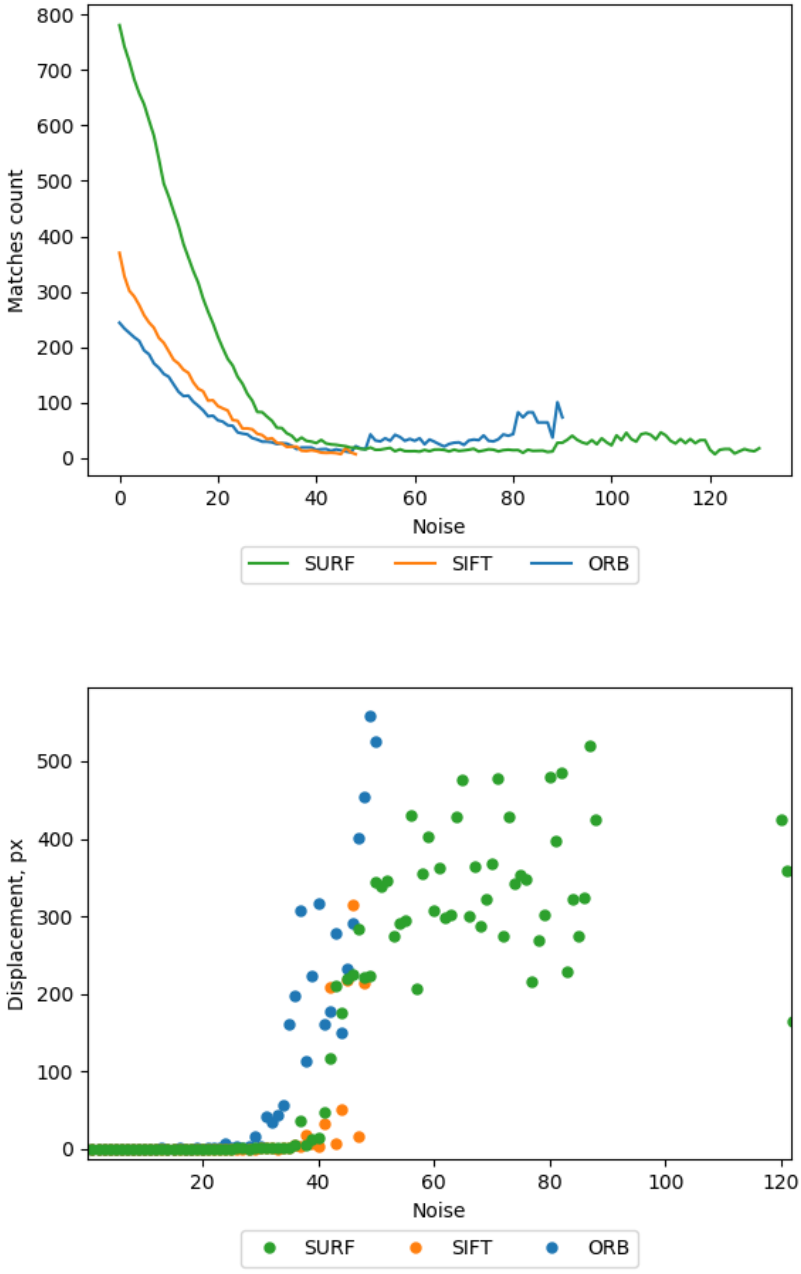

Рисунок 15 – Поведение числа совпадений точек и смещения при использовании гауссовского шума

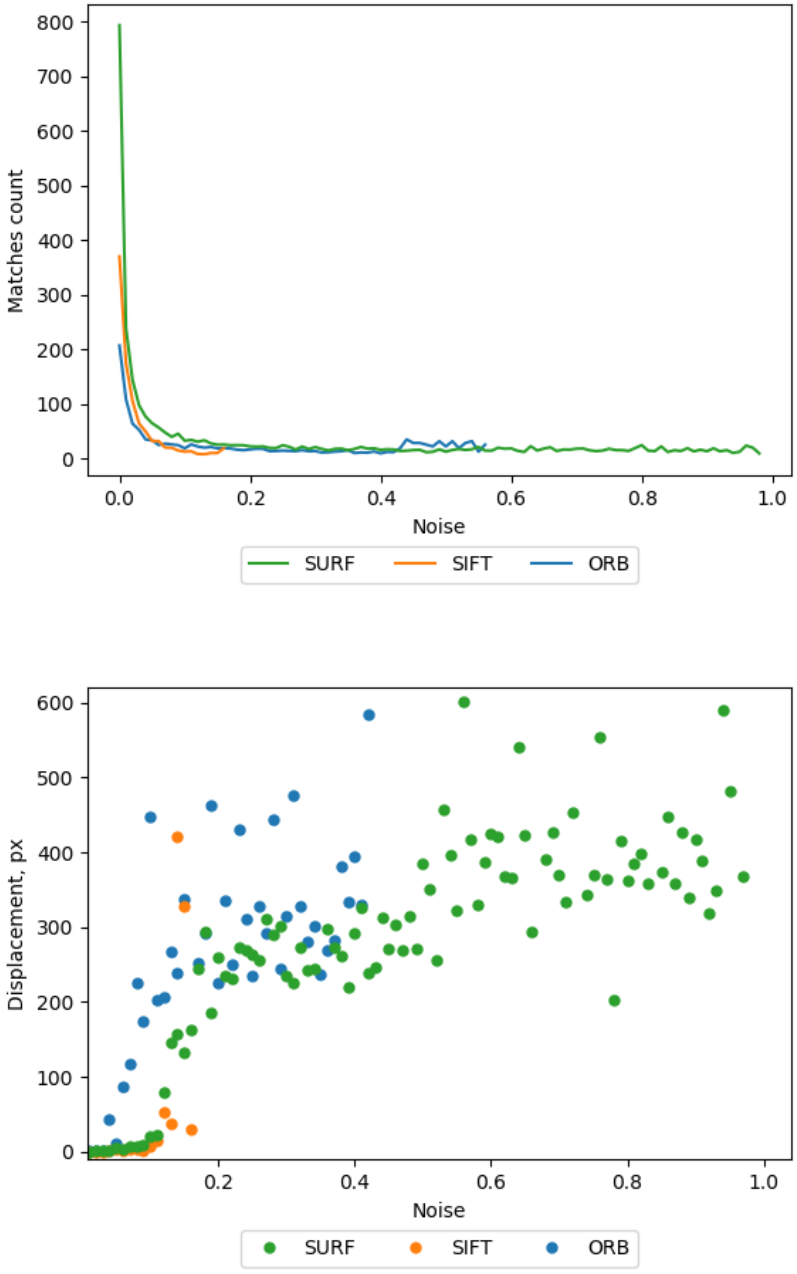

Рисунок 16 – Поведение числа совпадений точек и смещения при использовании шума соль-и-перец

4 Анализ полученных результатов

Проанализируем полученные результаты. Эксперименты показывают несколько интересных факторов в работе алгоритмов.

При размытии изображения дескриптор ORB раньше других рассмотренных начинает терять точность, несмотря на значительно большее общее количество найденных совпадений. Серьезное искажение результата достигается при размерах ядра Гауссова фильтра более 17 для тестового изображения размером 512 на 512 пикселей, с дальнейшим увеличением размера ядра ORB первым перестает находить достаточное число точек. Наилучшим образом при работе с размытым изображением показывает себя дескриптор SURF.

У всех рассмотренных алгоритмов наблюдается резкое увеличение искажения результата при переходе определенного порогового значения размытия.

На графике 17 можно наблюдать резкое падение качества при числе точек, меньшем чем 20. Это объясняется тем, что размытие может сильно ухудшать точность определения координат особых точек, даже при сохранении общей яркостной картины достаточном для построения дескриптора. Соответственно, при малых количествах точек, ошибки в отдельных координатах сильно влияют на построенную матрицу преобразования.

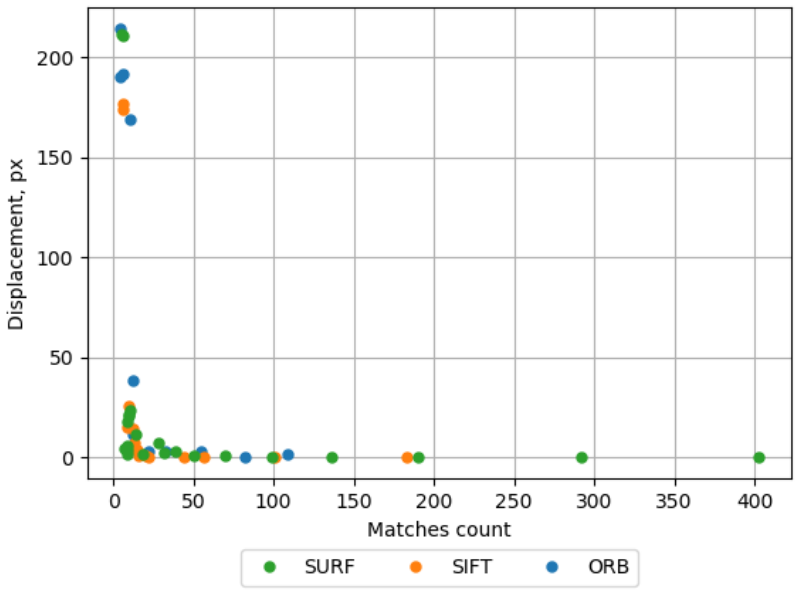

Рисунок 17 – Изменение смещения с ростом числа точек при использовании размытия

При применении случайных шумов можно наблюдать более интересную картину. При использовании дескриптора SIFT точность падает равномерно, алгоритм перестает работать на тестовых изображениях при значении σ гауссовского шума более 50 и при вероятности шума соль-перец более 0.15. При этом, остальные дескрипторы продолжают работать при значительно большем зашумлении, но стремительно теряют в точности, приводя к абсолютно непригодным для практического применения результатам. На рисунке 18 приведен пример результата, выданного алгоритмом SURF на значениях, при которых SIFT перестает работать.

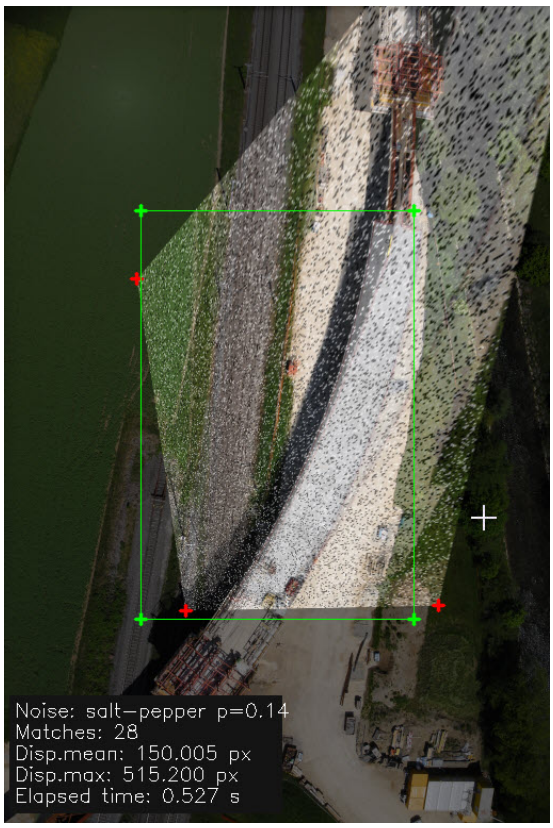

Рисунок 18 – Пример искажения при использовании алгоритма SURF

Принимая во внимание практическую непригодность изображений с подобными уровнями шумов и искажений, можно отбросить диапазоны значений за пределами стабильной работы алгоритма SIFT. На графиках 19-20 можно наблюдать этот диапазон.

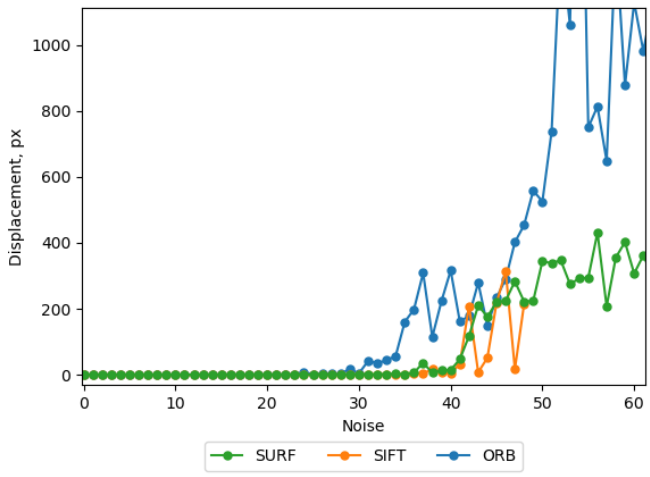

Рисунок 19 – Увеличенный фрагмент графика смещения при использовании гауссовского шума

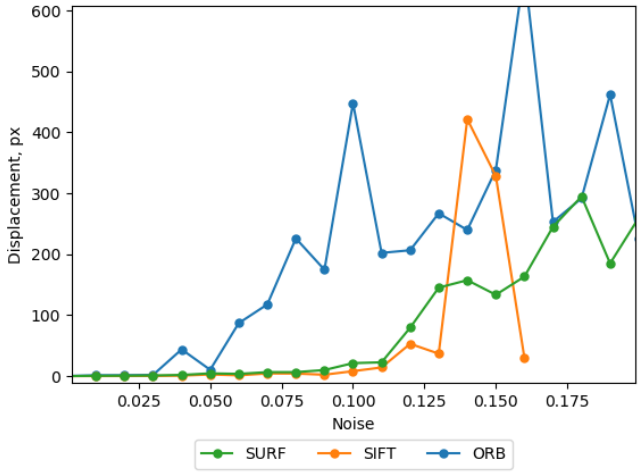

Рисунок 20 – Увеличенный фрагмент графика смещения при использовании шума соль-перец

Как можно видеть, все три алгоритма сохраняют достаточную точность при практически достижимых уровнях гауссовского шума. Однако, ORB наименее устойчив к шуму соль-и-перец, что можно видеть по резкому пику на графике при вероятности 0.03, сохраняющемуся даже при усреднении по набору экспериментов.

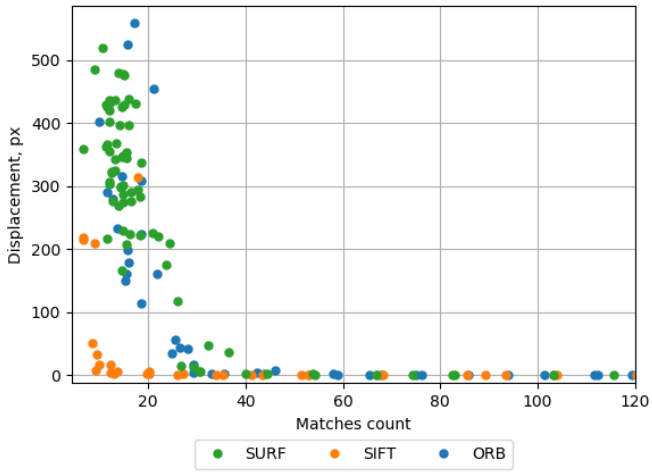

Рисунок 21 – Изменение смещения с ростом числа точек при использовании гауссовского шума

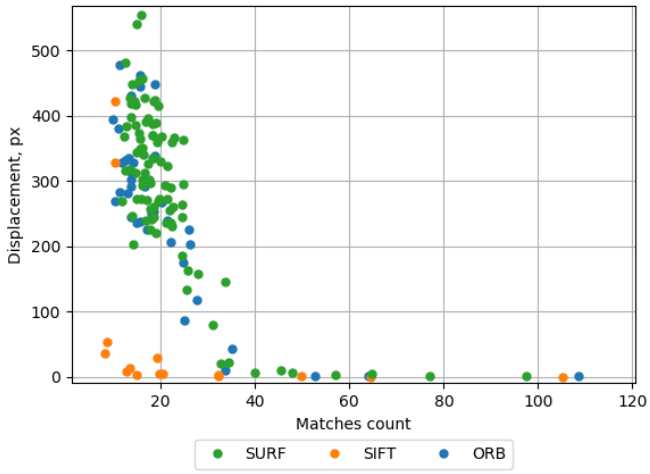

Рисунок 22 – Изменение смещения с ростом числа точек при использовании шума соль-и-перец

Зависимость смещения от количества найденных точек, как видно на графиках 21-22, выражена не так ярко, как при размытии, и можно наблюдать более плавный переход.

Итак, наивысшая точность при применении случайных шумов достигается при использовании алгоритма SIFT. Наихудший результат показывает ORB. Стоит заметить, что и при использовании импульсного шума соль-перец, и при использовании аддитивного гауссовского шума наибольшие количества особых точек находит алгоритм SURF.

Обобщая полученные результаты, можно сказать, что все рассмотренные алгоритмы проявили себя достаточно хорошо при тех уровнях зашумления, которые потенциально могут встретиться на практике. Значения, при которых наблюдается серьезное смещение относительно требуемого, в любом случае сделали бы на практике такую фотографию непригодной к использованию.

При размытии изображения наилучшим образом ведет себя алгоритм SURF, что позволяет рекомендовать его к использованию, если исходные фотографии сделаны с заметным нарушением фокусировки. При случайных шумах, как аддитивном, так и импульсном, наилучшим образом рабо-

тает SIFT, ценой большего времени выполнения и вычислительной сложности. Хуже всего в проведенных экспериментах отработал алгоритм ORB, в особенности, при использовании шума соль-перец.

ЗАКЛЮЧЕНИЕ

В ходе данной работы была рассмотрена задача сшивки изображения и методы ее решения. Проведен обзор математической постановки задачи. Рассмотрен алгоритм нахождения пространственного преобразования по опорным точкам. Был дан обзор нескольких методов поиска опорных точек, реализованных в открытой библиотеке OpenCV.

Рассмотрены цифровые шумы в задачах обработки изображений и их особенности, разобраны популярные математические модели искусственно генерируемых шумов, используемых для тестирования алгоритмов и оборудования.

Реализована компьютерная программа, осуществляющая совмещение изображений по особым точкам с возможностью выбора типа дескриптора. Также, реализована программная среда для экспериментов, позволяющая использовать изображения с генерацией различных типов и уровней зашумления, формировать отчеты о работе алгоритмов и анализировать графическое представление результатов.

В ходе серии экспериментов было выяснено, что представленные алгоритмы являются достаточно устойчивыми к зашумлению, и работают с достаточной точностью при уровнях шума, допускающих сохранение практической ценности таких изображений. Однако, можно выделить алгоритмы, лучше или хуже ведущие себя при определенных типах шумов. Результаты экспериментов показывают, что наилучшим образом все представленные алгоритмы справились с размытием, наихудшим - с импульсным шумом. Были сделаны выводы о применимости отдельных алгоритмов при конкретных практически возможных дефектах фотографий.

Дальнейшим развитием работы видится оценка при использовании на зашумленных изображениях восстанавливающих фильтров, либо восстановления шума при помощи моделей глубокого обучения.

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 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.

4 Проективное преобразование [Электронный ресурс] // Википедия : свободная энцикл. – Электрон. дан. – 2019. – URL: https://ru.wikipedia.org/?oldid=99107559 (дата обращения: 15.04.2019).

5 Аффинное преобразование [Электронный ресурс] // Википедия : свободная энцикл. – Электрон. дан. – 2019. – URL: https://ru.wikipedia.org/?oldid=93707864 (дата обращения: 15.04.2019).

6 Karami, E. Image Matching Using SIFT, SURF, BRIEF and ORB: Performance Comparison for Distorted Images [Электронный ресурс] / E. Karami, S. Prasad, M. Shehata // arXiv preprint. – 2015. – Электрон. дан. – URL: https://arxiv.org/ftp/arxiv/papers/1710/1710.02726.pdf (дата обращения: 25.05.2019).

7 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.

8 Moeslund, T. B. BLOB Analysis: An introduction to video and image processing [Текст] / T. B. Moeslund. – Springer, London, 2012. – 227 p. – p. 5-20.

9 Цифровой шум изображения [Электронный ресурс] // Википедия : свободная энцикл. – Электрон. дан. – 2019. – URL: https://ru.wikipedia.org/?oldid=95235149 (дата обращения: 15.04.2019).

10 Deng, G. An adaptive Gaussian filter for noise reduction and edge detection [Текст] / G. Deng, L. W. Cahill // IEEE Conference Record. – IEEE Computer Society Washington, DC, USA, 1993. – P. 1615-1619. b

11 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: 25.04.2019).

ПРИЛОЖЕНИЕ А

Код программы

```
import numpy as np
import imutils
import cv2
import argparse
import random
import glob
from timeit import default_timer as timer
import os


class OutputData():
    """ Contains output data from a single run of stitching algorythm
    """

    def __init__(self):
        self.matches_count = 0
        self.work_time = 0
        self.displacement = None
        self.src_points = None
        self.warp_points = None
        self.disp_mean = 0
        self.disp_median = 0
        self.imgA = None
        self.imgB = None
        self.matches_visualized = None
        self.result = None
        self.noise_info = ""
        self.matches_percentage = 0

    def compute_displacement(self):
        (pts, vrs) = self.src_points.shape
        self.displacement = np.zeros((pts), dtype="float32")
        for p in range(pts):
            for v in range(vrs):
                dsp = self.src_points[p][v] - self.warp_points[p][v]
                dsp = np.sqrt(dsp**2)
```

```
                self.displacement[p] = dsp
        self.disp_mean = np.mean(self.displacement)
        self.disp_median = np.median(self.displacement)

    def print_all(self):
        print("\n=============OUTPUT=DATA===============")
        print(f"Elapsed time: {self.work_time}")
        print(f"Noise: {self.noise_info}")
        print(f"Keypoint matches: {self.matches_count}")
        print(f"Matches %: {self.matches_percentage}")
        print(f"Displacement mean: {self.disp_mean}")
        print(f"Displacement median: {self.disp_median}")
        print(f"Displacement max:{max(abs(self.displacement))}")
        print(f"\nHomography:\n{self.homography}\n")
        print(f"Source pts:\n{self.src_points}\n")
        print(f"Warped pts:\n{self.warp_points}\n")
        print(f"Displacement:\n{self.displacement}\n")
        print("=======================================\n")


class Stitcher():
    """ This class is used for running stitching algorythm on two images
    """

    def __init__(self):
        self.isv3 = imutils.is_cv3(or_better=True)

    def stitch(self, images, ratio=0.75, reprojThresh=4.0, showMatches=False, desc='
        ↪ sift'):

        start = timer()

        # unpack the images, then detect keypoints and extract
        # local invariant descriptors from them
        (imgA, imgB) = images
        (kps_A, features_A, kps_raw_A) = self.detectAndDescribe(imgA, desc)
        (kps_B, features_B, kps_raw_B) = self.detectAndDescribe(imgB, desc)

        # match features between the two images
        matched_keypoints = self.matchKeypoints(kps_B, kps_A, features_B,
```

```
                                    features_A, ratio,
                                    reprojThresh)

    # if the match is None, then there aren't enough matched
    # keypoints to create a panorama
    if matched_keypoints is None:
        return (None, None)

    # otherwise, apply a perspective warp to stitch the images
    # together
    (matches, H, status) = matched_keypoints

    result = imgA
    try:
        warped = cv2.warpPerspective(imgB, H, (imgA.shape[1], imgA.shape[0]))
    except Exception:
        print("Finished by exception")
        return (None, None)

    result = cv2.addWeighted(warped, 0.5, result, 0.5, 0.0)

    end = timer()

    output_data = OutputData()

    if showMatches:
        vis = self.drawMatches(imgB, imgA, kps_B, kps_A, matches, status)
        output_data.matches_visualized = vis

    mchs_count = len(matches)
    output_data.matches_count = mchs_count
    output_data.matches_percentage = (mchs_count / len(kps_A) +
                                      mchs_count / len(kps_B)) / 2
    output_data.homography = H
    output_data.work_time = end - start
    output_data.result = result
    output_data.imgB_warped = warped
    output_data.imgA = imgA
    output_data.imgB = imgB
```

```
        # return the stitched image
        return (result, output_data)

    def detectAndDescribe(self, image, desc):
        # convert the image to grayscale
        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        # check to see if we are using OpenCV 3.X
        if self.isv3:
            # detect and extract features from the image
            if desc == 'sift':
                descriptor = cv2.xfeatures2d.SIFT_create()
            elif desc == 'surf':
                descriptor = cv2.xfeatures2d.SURF_create()
            elif desc == 'orb':
                descriptor = cv2.ORB_create(4096)
            (kps, features) = descriptor.detectAndCompute(image, None)

        # otherwise, we are using OpenCV 2.4.X
        else:
            # detect keypoints in the image
            detector = cv2.FeatureDetector_create("SIFT")
            kps = detector.detect(gray)

            # extract features from the image
            extractor = cv2.DescriptorExtractor_create("SIFT")
            (kps, features) = extractor.compute(gray, kps)

        # convert the keypoints from KeyPoint objects to NumPy
        # arrays
        kps_32 = np.float32([kp.pt for kp in kps])

        # return a tuple of keypoints and features
        return (kps_32, features, kps)

    def matchKeypoints(self, kpsA, kpsB, featuresA, featuresB, ratio,
                       reprojThresh):
        # compute the raw matches and initialize the list of actual
        # matches
        matcher = cv2.DescriptorMatcher_create("BruteForce")
```

```
        rawMatches = matcher.knnMatch(featuresA, featuresB, 2)
        matches = []

        # loop over the raw matches
        for m in rawMatches:
            # ensure the distance is within a certain ratio of each
            # other (i.e. Lowe's ratio test)
            if len(m) == 2 and m[0].distance < m[1].distance * ratio:
                matches.append((m[0].trainIdx, m[0].queryIdx))

        # computing a homography requires at least 4 matches
        if len(matches) >= 4:
            # construct the two sets of points
            ptsA = np.float32([kpsA[i] for (_, i) in matches])
            ptsB = np.float32([kpsB[i] for (i, _) in matches])

            # compute the homography between the two sets of points
            (H, status) = cv2.findHomography(ptsA, ptsB, cv2.RANSAC,
                                             reprojThresh)

            # return the matches along with the homograpy matrix
            # and status of each matched point
            return (matches, H, status)

        # otherwise, no homograpy could be computed
        return None

    def drawMatches(self, imageA, imageB, kpsA, kpsB, matches, status):
        # initialize the output visualization image
        (hA, wA) = imageA.shape[:2]
        (hB, wB) = imageB.shape[:2]
        vis = np.zeros((max(hA, hB), wA + wB, 3), dtype="uint8")
        vis[0:hA, 0:wA] = imageA
        vis[0:hB, wA:] = imageB

        # vis = cv2.drawMatches(imageA,
        # kpsA,
        # imageB,
        # kpsB,
        # matches,
```

```
# outImg=vis,
# flags=2)

#kek = imageA
# loop over the matches
for ((trainIdx, queryIdx), s) in zip(matches, status):
    # only process the match if the keypoint was successfully
    # matched
    if s == 1:
        # draw the match
        ptA = (int(kpsA[queryIdx][0]), int(kpsA[queryIdx][1]))
        ptB = (int(kpsB[trainIdx][0]) + wA, int(kpsB[trainIdx][1]))

        color = (int(random.random() * 255),
                int(random.random() * 255),
                int(random.random() * 255))

        # cv2.drawMarker(vis,
        # ptA,
        # (255, 255, 0),
        # markerType=cv2.MARKER_TILTED_CROSS,
        # markerSize=10,
        # thickness=1,
        # line_type=cv2.LINE_AA)
        # cv2.drawMarker(vis,
        # ptB,
        # (255, 255, 0),
        # markerType=cv2.MARKER_TILTED_CROSS,
        # markerSize=10,
        # thickness=1,
        # line_type=cv2.LINE_AA)

        # cv2.line(vis, ptA, ptB, (255, 255, 0), 1)
        # cv2.circle(vis, ptA, 6,
        # (0, 0, 255), 2)
        # cv2.circle(vis, ptB, 6,
        # (0, 0, 255), 2)

        cv2.line(vis, ptA, ptB, color, 1)
        cv2.circle(vis, ptA, 6, color, 2)
```

```
            cv2.circle(vis, ptB, 6, color, 2)

            #cv2.circle(kek, ptA, 4, color, 1)


        #cv2.imshow("kek", kek)
        # return the visualization
        return vis


class Noiser():
    """ Contains several noise functions
    """

    def sp_noise(self, image, prob):
        output = np.zeros(image.shape, np.uint8)
        thres = 1 - prob
        for i in range(image.shape[0]):
            for j in range(image.shape[1]):
                rdn = random.random()
                if rdn < prob:
                    output[i][j] = 0
                elif rdn > thres:
                    output[i][j] = 255
                else:
                    output[i][j] = image[i][j]
        return output

    def gauss_noise(self, image, mean, sigma):
        row, col, sp = image.shape
        output = np.zeros(image.shape, np.uint8)
        for i in range(image.shape[0]):
            for j in range(image.shape[1]):
                for k in range(image.shape[2]):
                    output[i][j][k] = image[i][j][k] + np.random.normal(
                        mean, sigma)
                    # if output[i][j][k] > 255:
                    # output[i][j][k] = 255
        cv2.normalize(output, output, 0, 255, cv2.NORM_MINMAX, dtype=-1)
        return output
```

```
        # gaussian = np.random.normal(mean, sigma, image.shape)

        # output = np.zeros(image.shape, np.float32)

        # if len(image.shape) == 2:
        # output = image + gaussian
        # else:
        # print(image.shape)
        # print(gaussian.shape)
        # print(output.shape)
        # print(output[:, :, 0])
        # print(image[:, :, 0])
        # output[:, :, 0] = image[:, :, 0] + gaussian
        # output[:, :, 1] = image[:, :, 1] + gaussian
        # output[:, :, 2] = image[:, :, 2] + gaussian

        # cv2.normalize(output, output, 0, 255, cv2.NORM_MINMAX, dtype=-1)
        # output = output.astype(np.uint8)
        # return output

    def gauss_blur(self, image, kernel_size):
        output = cv2.GaussianBlur(image, (kernel_size, kernel_size), 0)
        return output


class Drawer():
    """ Draws information from OutputData object on an image
    """

    def draw_corners(self, data: OutputData, img):
        def marker(point, color):
            cv2.drawMarker(img,
                           point,
                           color,
                           markerType=cv2.MARKER_CROSS,
                           markerSize=10,
                           thickness=2,
                           line_type=cv2.LINE_AA)
```

```
    def circle(point, color):
        cv2.circle(img, point, 3, color, -1)

    def line(ptA, ptB, color):
        cv2.line(img, ptA, ptB, color, 1)

    color_src = (0, 255, 0)
    color_warp = (0, 0, 255)

    marker((data.warp_points[0][0], data.warp_points[0][1]), color_warp)
    marker((data.warp_points[1][0], data.warp_points[1][1]), color_warp)
    marker((data.warp_points[2][0], data.warp_points[2][1]), color_warp)
    marker((data.warp_points[3][0], data.warp_points[3][1]), color_warp)

    marker((data.src_points[0][0], data.src_points[0][1]), color_src)
    marker((data.src_points[1][0], data.src_points[1][1]), color_src)
    marker((data.src_points[2][0], data.src_points[2][1]), color_src)
    marker((data.src_points[3][0], data.src_points[3][1]), color_src)

    line((data.src_points[0][0], data.src_points[0][1]),
         (data.src_points[1][0], data.src_points[1][1]), color_src)
    line((data.src_points[1][0], data.src_points[1][1]),
         (data.src_points[3][0], data.src_points[3][1]), color_src)
    line((data.src_points[3][0], data.src_points[3][1]),
         (data.src_points[2][0], data.src_points[2][1]), color_src)
    line((data.src_points[2][0], data.src_points[2][1]),
         (data.src_points[0][0], data.src_points[0][1]), color_src)

def draw_texts(self, data: OutputData, result):
    (heigth, width) = result.shape[:2]

    cv2.rectangle(result, (5, heigth - 120), (260, heigth - 10),
                  (20, 20, 20), -1)

    font = cv2.FONT_HERSHEY_SIMPLEX
    cv2.putText(result, f"Noise: {data.noise_info}", (10, heigth - 100),
                font, 0.6, (255, 255, 255), 1)
    cv2.putText(result, f"Matches: {data.matches_count}",
                (10, heigth - 80), font, 0.6, (255, 255, 255), 1)
    cv2.putText(result, f"Disp.mean: {round(data.disp_mean, 5):.3f} px",
```

```
                (10, heigth - 60), font, 0.6, (255, 255, 255), 1)
        cv2.putText(result,
                f"Disp.max: {round(max(data.displacement), 5):.3f} px",
                (10, heigth - 40), font, 0.6, (255, 255, 255), 1)
        cv2.putText(result, f"Elapsed time: {round(data.work_time, 5):.3f} s",
                (10, heigth - 20), font, 0.6, (255, 255, 255), 1)


class Writer():
    """This class writes OutputData objects to file of specified format"""

    def write_results_plain_text(self, dir: str, filename: str, results):
        def build_str(data: OutputData, idx):
            return f"{idx} {data.noise_info} {data.matches_count} {data.disp_median:.5f}
                ↪ {data.displacement}\n"

        # dirpath = filename.split("__")[0]
        # if not os.path.exists(f"results/{dirpath}"):
        # os.makedirs(f"results/{dirpath}")
        dirpath = dir
        if not os.path.exists(f"results/{dirpath}"):
            os.makedirs(f"results/{dirpath}")

        i = 0
        while os.path.isfile(f"results/{dirpath}/{filename}_n{i}.txt"):
            i += 1
        filename = f"{filename}_n{i}"

        file = open(f"results/{dirpath}/{filename}.txt", "w+")

        header = "Idx Noise Matches D.median D\n"
        file.write(header)
        i = 1
        for d in results:
            file.write(build_str(d, i))
            i += 1
        file.close()
        print(f"Writed to {filename}")
```

```
def test_cutted_part(directory_path: str):
    """ This test cuts a part from an original image, applies noise function with
        given value, then tries to put cutted part back to it's original position
    """

    def blur_test(noiser: Noiser, img, value):
        noised = noiser.gauss_blur(img, value)
        return noised, f"gauss blur m={value}"

    def gauss_noise_test(noiser: Noiser, img, value):
        (mean, var) = value
        noised = noiser.gauss_noise(img, mean, var)
        return noised, f"gauss noise s={var}"

    def sp_test(noiser: Noiser, img, value):
        noised = noiser.sp_noise(img, value)
        return noised, f"salt-pepper p={value}"

    def identity_test(noiser: Noiser, img, value):
        noised = img
        return noised, f"none"

    filenames = [img for img in glob.glob(f"{directory_path}*.jpg")]
    image = [cv2.imread(img) for img in filenames][0]

    width = 512
    image = imutils.resize(image, width=width)
    (heigth, width) = image.shape[:2]

    # Cutting the part of an original image
    corners_src = [
        int(heigth * 0.25),
        int(heigth * 0.75),
        int(width * 0.25),
        int(width * 0.75)
    ]
    square = image[corners_src[0]:corners_src[1], corners_src[2]:
                corners_src[3]]
    (sqr_heigth, sqr_width) = square.shape[:2]
```

```
corners_local = np.array(
    [[[0, 0], [0, sqr_heigth]], [[sqr_width, 0], [sqr_width, sqr_heigth]]],
    dtype="float32")

# width_shift = int(sqr_width * 0.25)
# # heigth_shift = int(sqr_width * 0.1)
# persp_shift = np.array(
# [[0, 0], [0-width_shift, sqr_heigth],
# [sqr_width, 0], [sqr_width+width_shift, sqr_heigth]],
# dtype="float32")

# M = cv2.getPerspectiveTransform(np.reshape(corners_local, (4, 2)), persp_shift)
# square = cv2.warpPerspective(square, M, (sqr_width, sqr_heigth))

# Adding selected type of noise
noiser = Noiser()
(square_noised, noise_param) = sp_test(noiser, square, 0.14)
#(square_noised, noise_param) = gauss_noise_test(noiser, square, (0, 30))
#(square_noised, noise_param) = blur_test(noiser, square, 21)
#(square_noised, noise_param) = identity_test(noiser, square, None)

# Stitching images together
stitcher = Stitcher()
(result, data) = stitcher.stitch([image, square_noised], showMatches=True, desc='
    ↪ surf')

# Performing perspective transformation on image corner points
# to compute displacement and check precision
corners_global = np.array(
    [[[corners_src[2], corners_src[0]], [corners_src[2], corners_src[1]]],
     [[corners_src[3], corners_src[0]], [corners_src[3], corners_src[1]]]],
    dtype="float32")
corners_warp = cv2.perspectiveTransform(corners_local, data.homography)

data.src_points = np.reshape(corners_global, (4, 2))
data.warp_points = np.reshape(corners_warp, (4, 2))
data.compute_displacement()
data.noise_info = noise_param
data.print_all()
```

```
    drawer = Drawer()
    drawer.draw_corners(data, result)
    drawer.draw_texts(data, result)

    cv2.imshow("Source", data.imgA)
    cv2.imshow("Cutted", data.imgB)
    cv2.imshow("Warped", data.imgB_warped)
    cv2.imshow("Result", result)
    cv2.imshow("Matches", data.matches_visualized)

    cv2.waitKey(0)
    cv2.destroyAllWindows()


def test_multiple_pass_cutted_part(image, directory_path: str,
                                   report_path: str):
    """ This test is running as test_cutted_part, but with several iterations,
        increasing the noise amount until stitching algorythm stops working
    """

    def single_pass(image, square, corners, stitcher: Stitcher, noiser: Noiser,
                    fun_args, desc):

        (n_type, values) = fun_args
        (corners_local, corners_global) = corners

        square_noised = square
        # Adding selected type of noise
        if n_type == "sp":
            (p1, ) = values
            square_noised = noiser.sp_noise(square, p1)
        elif n_type == "gauss_blur":
            (p1, ) = values
            square_noised = noiser.gauss_blur(square, p1)
        elif n_type == "gauss_noise":
            (p1, p2) = values
            square_noised = noiser.gauss_noise(square, p1, p2)

        # Stitching images together
        (result, data) = stitcher.stitch([image, square_noised],
```

```
                                showMatches=False, desc=desc)

    # TODO
    if result is None:
        print("Work done")
        return None

    corners_warp = cv2.perspectiveTransform(corners_local, data.homography)

    data.src_points = np.reshape(corners_global, (4, 2))
    data.warp_points = np.reshape(corners_warp, (4, 2))
    data.compute_displacement()

    if n_type == "sp":
        data.noise_info = f"salt pepper p={p1:.5f}"
    elif n_type == "gauss_blur":
        data.noise_info = f"gauss blur k={p1:.5f}"
    elif n_type == "gauss_noise":
        data.noise_info = f"gauss noise m={p1:.5f} v={p2:.5f}"

    print("Single pass done")

    return data

def blur_test(desc):
    results = []
    init_val = 3
    noise_val = init_val
    noise_step = 2
    while True:
        data = single_pass(image, square, (corners_local, corners_global),
                           stitcher, noiser, ("gauss_blur", (noise_val, )), desc)
        if data is None:
            break
        else:
            results.append(data)
            noise_val += noise_step
    return results, f"blur_{init_val}_{noise_step}"

def gauss_noise_test(desc):
```

```
    results = []
    mean = 0
    init_variance = 0
    noise_val = init_variance
    noise_step = 1
    while True:
        data = single_pass(image, square, (corners_local, corners_global),
                           stitcher, noiser,
                           ("gauss_noise", (mean, noise_val)), desc)
        if data is None:
            break
        else:
            results.append(data)
            noise_val += noise_step
    return results, f"gauss_{mean}_{init_variance}_{noise_step}"

def sp_test(desc):
    results = []
    init_val = 0
    noise_val = init_val
    noise_step = 0.01
    while True:
        data = single_pass(image, square, (corners_local, corners_global),
                           stitcher, noiser, ("sp", (noise_val, )), desc)
        if data is None:
            break
        else:
            results.append(data)
            noise_val += noise_step
    return results, f"salt_pepper_{init_val}_{noise_step}"

width = 512
image = imutils.resize(image, width=width)
(heigth, width) = image.shape[:2]

# Cutting the part of an original image
corners_src = [
    int(heigth * 0.25),
    int(heigth * 0.75),
    int(width * 0.25),
```

```
        int(width * 0.75)
    ]
    square = image[corners_src[0]:corners_src[1], corners_src[2]:
                corners_src[3]]
    (sqr_heigth, sqr_width) = square.shape[:2]

    corners_local = np.array(
        [[[0, 0], [0, sqr_heigth]], [[sqr_width, 0], [sqr_width, sqr_heigth]]],
        dtype="float32")

    # Performing perspective transformation on image corner points
    # to compute displacement and check precision
    corners_global = np.array(
        [[[corners_src[2], corners_src[0]], [corners_src[2], corners_src[1]]],
         [[corners_src[3], corners_src[0]], [corners_src[3], corners_src[1]]]],
        dtype="float32")

    stitcher = Stitcher()
    noiser = Noiser()

    desc = 'orb'
    (results, name) = blur_test(desc)

    filename = f"{directory_path.replace('/', '_')}_{name}_{desc}"
    writer = Writer()
    writer.write_results_plain_text(report_path, filename, results)


def test_iterative(directory_path: str,
                   i: int,
                   report_path: str,
                   single_img=True):

    filenames = [img for img in glob.glob(f"{directory_path}*.jpg")]

    if single_img:
        image = [cv2.imread(img) for img in filenames][0]

        for j in range(i):
            test_multiple_pass_cutted_part(image, directory_path, report_path)
```

```
            print(f"Done {j+1} from {i} tests")
    else:
        images = [cv2.imread(img) for img in filenames]
        for j in range(i):
            k = 0
            for img in images:
                test_multiple_pass_cutted_part(img,
                                               f"{directory_path}_file{[k]}",
                                               report_path)
                print(
                    f"Done {j+1} from {i} tests for img {k} from {len(filenames)}"
                )
                k += 1


def stitch_sequence(directory_path: str):
    """ This method tries to combine several images into one sequence. Files must be
        named in alphabetical (or increasing numeric) order.
    """

    filenames = [img for img in glob.glob(f"{directory_path}*.jpg")]

    filenames.sort()

    images = [cv2.imread(img) for img in filenames]

    width = 256
    images = [imutils.resize(img, width=width) for img in images]

    noiser = Noiser()
    stitcher = Stitcher()

    result = images[0]
    for i in range(1, len(images)):
        (result, data) = stitcher.stitch([result, images[i]],
                                         showMatches=False)

    cv2.imshow("Result", result)

    cv2.waitKey(0)
```

```
    cv2.destroyAllWindows()


def stitch_two_images(directory_path: str):
    """ This is a basic stitching of two images
    """

    # load the two images and resize them to have a width of 400 pixels
    # (for faster processing)
    width = 512
    imageA = cv2.imread(f"{directory_path}1.jpg")
    imageB = cv2.imread(f"{directory_path}2.jpg")
    imageA = imutils.resize(imageA, width=width)
    imageB = imutils.resize(imageB, width=width)

    noiser = Noiser()
    # imageB = imageA[50:400, 50:200]

    # imageB = noiser.gauss_noise(imageB, 10, 10)
    # imageB = noiser.sp_noise(imageB, 0.08)
    # imageB = noiser.gauss_blur(imageB, 27)

    # stitch the images together
    stitcher = Stitcher()
    (result, data) = stitcher.stitch([imageA, imageB], showMatches=True, desc='orb')

    # imageC = cv2.imread("images/road_dataset/3.jpg")
    # imageC = imutils.resize(imageC, width=width)

    # (result2, vis) = stitcher.stitch([result, imageC], showMatches=True)

    # show the images
    cv2.imshow("Source", data.imgA)
    cv2.imshow("Cutted", data.imgB)
    cv2.imshow("Warped", data.imgB_warped)
    cv2.imshow("Result", result)
    cv2.imshow("Matches", data.matches_visualized)

    # cv2.imshow("Result 2", result2)
```

```
    # cv2.imwrite(f"D:/repositories/stud/cv_experiments/results/{firstimg_name}_{
        ↪ secondimg_name}/matches.jpg", vis)
    # cv2.imwrite(f"matches.jpg", vis)
    # cv2.imwrite(f"result.jpg", result)

    cv2.waitKey(0)
    cv2.destroyAllWindows()


if __name__ == "__main__":
    ap = argparse.ArgumentParser()

    # String path to images directory
    ap.add_argument("-dir", required=True, help="Set working directory")

    # A boolean flag of operation type
    ap.add_argument("-op",
                    required=True,
                    type=str,
                    choices=['two', 'set', "test", "mult", "var-imgs"],
                    help="Type of operation")

    ap.add_argument("-i",
                    required=False,
                    type=int,
                    help="Number of operations")

    ap.add_argument("-rep",
                    required=True,
                    type=str,
                    help="Report(s) directory")

    args = ap.parse_args()

    directory_path = f"images/{args.dir}/"

    if args.op == 'two':
        stitch_two_images(directory_path)
    elif args.op == 'set':
        stitch_sequence(directory_path)
```

```
elif args.op == 'test':
    test_cutted_part(directory_path)
elif args.op == 'mult':
    if args.i is None:
        test_iterative(directory_path, 1, args.rep)
    else:
        test_iterative(directory_path, args.i, args.rep)
elif args.op == 'var-imgs':
    if args.i is None:
        test_iterative(directory_path, 1, args.rep, single_img=False)
    else:
        test_iterative(directory_path, args.i, args.rep, single_img=False)
```