**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ТВ**

отчет

**по лабораторной работе №6**

**по дисциплине «Интеллектуальные видеосистемы»**

Тема: СЕГМЕНТАЦИЯ ДВИЖУЩИХСЯ ОБЪЕКТОВ НА ОСНОВЕ ОЦЕНКИ ПОЛЯ ВЕКТОРОВ ДВИЖЕНИЯ

|  |  |  |
| --- | --- | --- |
| Студент гр. 8571 |  | Темный Д.А. |
| Преподаватель |  | Поздеев А.А. |

Санкт-Петербург

2023

**Цель работы**

Целью лабораторной работы является знакомство с алгоритм сегментации объектов на основе оценки поля векторов движения при отсутствии глобального движения. В соответствии с заданием необходимо разработать программу, моделирующую алгоритм сегментации объектов на основе векторов движения. Исходными данными являются два кадра видеопоследовательности со статичным фоном, на которых присутствует пара разнонаправленно перемещающихся объектов. В программе должны быть реализованы следующие процедуры с выводом результата на экран:

* загрузка пары кадров,
* разбиение кадров на блоки,
* реализация корреляционного алгоритма поиска векторов движения (с выводом векторов поверх изображения), с использованием в качестве целевой функции по указанию преподавателя MAD, или MSE (среднюю квадратичную ошибку) и одну из процедур для организации поиска (по указанию преподавателя): полный перебор, 3SS, 4SS, логарифмический, или ортогональный поиск.
* фильтрация поля векторов движения (демонстрация результата на изображении),
* кластеризация векторов и сегментация объектов (с отображением результата на кадре).

**Теоретические сведения**

Для сегментации и сопровождения объектов по признаку движения может быть использована оценка поля векторов движения. Это поле описывает видимое перемещение объектов в кадре. Векторы движения позволяют не только детектировать факт перемещения объекта, но и оценить его скорость и направление.

Наличие информации о направлении и скорости позволяет решить большое число прикладных задач:

* сегментировать объекты на сложном фоне, находящиеся в непосредственной близости друг к другу;
* разрешить ситуации окклюзии при сопровождении путем выявления объекта, находящегося на переднем плане;
* построить модель движения по совокупности векторов (на основе информации, полученной в одном кадре) и осуществить сопровождение объекта.

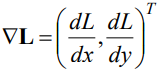
Аппарат векторов движения (векторов оптического потока) был разработан для применения в стандартах видеокомпрессии MPEG. Однако, в видеоаналитике векторы движения также стали очень популярны и востребованы. В задачах компьютерного зрения вектор движения показывает смещение фрагмента изображения в кадре *t+*1 относительно кадра *t*.

Для определения векторов движения основным является уравнение оптического потока, полученное на основе допущения о постоянстве яркости *L(x,y,t)* точки (пикселя) при движении



Так как x=f(t) и y=f(t), вычисления нужно проводить по формуле сложной производной:



где *L(x,y,t)* – яркость пикселя с координатами x и y в момент времени t; 〈…〉 обозначает скалярное произведение векторов, а , – векторградиент;  – вектор скорости (оптического потока).

Вместо производных по времени и пространству используют их целочисленные приближения



и вычисляют приращения яркости в соседних пикселях по вертикали и горизонтали, а также в соседнем кадре, т. е. приравнивают Δ*x* и Δ*y* одному пикселю, а Δ*t* - одному кадру (рис. 1).

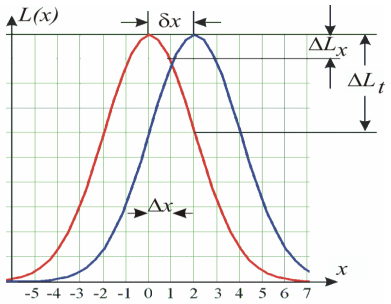


Рис. - Иллюстрация к расчету вектора движения

Из анализа уравнения оптического потока можно сделать следующие выводы (которые объясняют ограничения метода):

1. Уравнение оптического потока является недоопределенным и позволяет найти только сонаправленную с яркостным вектором-градиентом  компоненту векторов оптического потока. Ортогональная с яркостным вектором-градиентом компонента может принимать любые значения, не изменяя скалярного произведения, и поэтому не может быть определена однозначно.

Для полной оценки векторов оптического потока необходимо ввести требование гладкости – близости скоростей у группы соседних пикселей. Принятие решения о размере такой группы называют проблемой апертуры. Рис. 6.1. Иллюстрация к расчету вектора движения

1. Однозначное определение векторов оптического потока возможно только в случае, если компоненты яркостного вектора градиента отличны от нуля, т.е. имеют место изменения яркости по горизонтали и вертикали. В случае гладкой поверхности достоверную оценку векторов найти нельзя.
2. Уравнение оптического потока предполагает постоянство яркости при движении точки вдоль траектории. Подсветки, тени, блики (которые часто возникают при видеонаблюдении на открытом воздухе), низкая детальность, прозрачные и зеркальные поверхности реальных объектов нарушают это утверждение, что затрудняет вычисления и приводит к ошибкам при определении векторов оптического потока.

Перечисленные факторы обусловливают высокую вероятность появления ошибочных, так называемых «аномальных векторов», которые не соответствуют реальному перемещению, существующему в видеопоследовательности, и могут быть не сонаправленными с движением объекта, к которому они принадлежат. Аномальные векторы в задачах видеокомпрессии приводят только к уменьшению степени сжатия, в то время как в системах видеоаналитики их следствием является разделение объектов интереса на части (ошибка сегментации), снижение точности моделей движения, потеря объекта при сопровождении.

***Корреляционные методы*** для поиска векторов движения определяют векторы оптического потока на основе смещений, при которых достигается максимальное соответствие фрагментов изображения текущего и предыдущего кадров. Определение наилучшего соответствия выполняется путем поиска максимума корреляционной функции.

Наиболее часто используют метод сопоставления блоков, принятый во многих стандартах видеокодирования. Метод состоит из следующих основных шагов:

* текущий кадр делится на неперекрывающиеся квадратные блоки размером *M*×*N* пикселей;
* для каждого блока формируется область поиска в предыдущем кадре, которая имеет размер (2*d*+*M*+1)×(2*d*+*N*+1) пикселей, где *d* – это максимально возможное смещение в горизонтальном и вертикальном направлениях (рис. 2);
* выполняется совмещение блоков текущего кадра с блоками предыдущего кадра видеопоследовательности, определяется, какому блоку в области поиска текущий блок лучше всего соответствует, и оценивается величина смещения положения блока в текущем кадре относительно предыдущего – вектор движения.

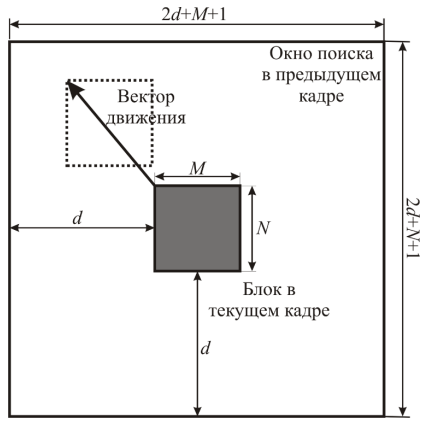


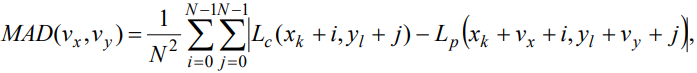
Рис. - Совмещение блоков

Считается, что все пиксели блока претерпевают одинаковое перемещение и им приписывается один и тот же вектор движения.

Задача определения векторов движения в этом случае решается путем минимизации целевой функции, характеризующей степень соответствия (совпадения) двух блоков, на множестве различных положений обрабатываемого блока в области поиска.

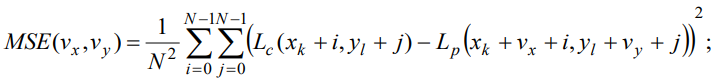
***Формирование целевой функции***, оценивающей степень соответствия между блоком текущего кадра и блоком предыдущего кадра, может быть выполнено в нескольких вариантах:

1. средняя абсолютная разность (MAD):

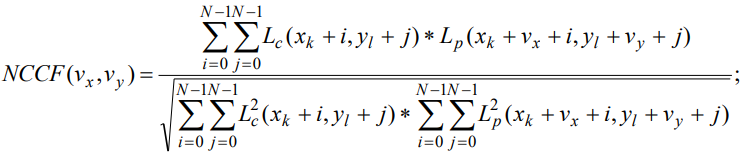


где *Lc*(…) и *Lp*(…) – яркости пикселя в текущем и предыдущем кадре соответственно; (*xk, yl*) – координаты пикселя левого верхнего угла текущего блока; *N·N* – размер блока; (*vx, vy*) – один из возможных векторов движения;

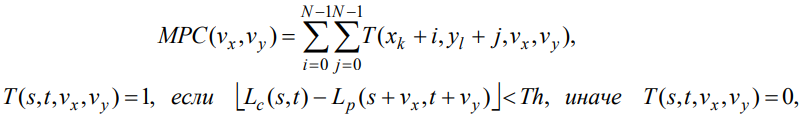
1. среднеквадратическая ошибка (MSE):



1. нормированная функция взаимной корреляции (NCCF):



1. максимальное число соответствующих пикселей (MPC):



где *Th* – предопределенный порог.

В отличие от предыдущих критериев блок наилучшего соответствия, найденный с использованием *MPC*, тот, который дает самое большое значение целевой функции. Обычно в качестве целевой функции используется *MAD*, так как она дает характеристику, близкую к характеристике *MSE*, но не требует операций умножения.

***Совмещение блоков.*** Самым простым и надежным алгоритмом, позволяющим выполнить совмещение блоков, является полный перебор *FS* (*full search*), но из-за большого объема вычислений он обладает низкой скоростью. Разработано большое число алгоритмов, которые направлены на оптимизацию стратегии поиска минимума. Их можно разделить на два класса:

алгоритмы, уменьшающие число вычислений при определении целевой функции;

алгоритмы, уменьшающие число контрольных точек в области поиска.

Примерами, относящимися к первому классу, являются алгоритм неполного определения целевой функции, алгоритм остановки на полпути (*PDS*) и остановки на полпути c нормировкой (*NPDS*).

В рамках второго класса можно выделить три группы алгоритмов:

1. основанные на свойстве унимодальности целевой функции; примерами являются алгоритм поиска тремя итерациями (*3SS*), алгоритмы логарифмического, ортогонального и поперечного поиска, алгоритм поиска по квадрантам;
2. учитывающие одновременно унимодальность минимизируемой функции и скорость оцениваемого движения; к этой группе относятся алгоритмы двух видов: ориентированные на работу с видеопоследовательностями, в которых преобладает медленное движение, такие как блочный градиентный поиск, четырехшаговый алгоритм (*4SS*), а также производящие предсказание характера оцениваемого движения (быстрое/медленное) и далее использующие наиболее эффективный для данного вида движения подход, – гибридные алгоритмы.
3. предсказывающие начальное приближение: алгоритмы с предсказанием и иерархический поиск.

В качестве примера рассмотрим работу алгоритма *3SS*. Многие из вышеперечисленных подходов будут отличаться от него только расположением точек для вычисления значений целевой функции, или числом шагов.

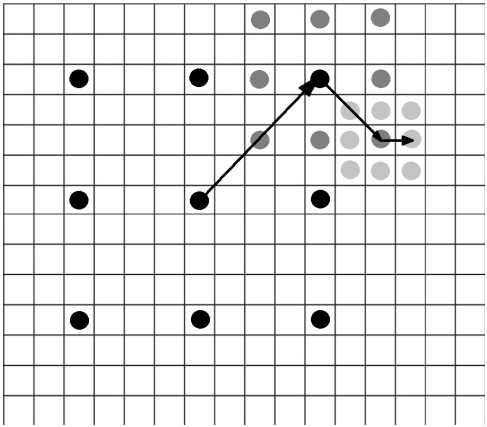


Рис. - Иллюстрация алгоритма 3SS

Для каждого блока в кадре t на первом этапе вычисляется целевая функция (*MAD*, *MSE*, или *NCCF*) в девяти точках, то есть мера корреляции с девятью блоками в кадре *t*+1 (черные точки на рис. 3), отстоящих друг от друга на некоторый шаг *d* и формирующих квадрат. Затем, уменьшив шаг *d* вдвое, вычисляют еще 8 значений целевой функции относительно точки, принадлежащей блоку, который имеет наибольшую корреляцию с блоком кадра *t* – минимум целевой функции (темно-серые точки на рис. 3). На последнем, третьем, шаге операцию повторяют, снова уменьшив шаг *d* в два раза. В результате, для каждого блока в кадре t находят наиболее похожий на него блок в кадре *t*+1 и соответственно определяют вектор движения.

Разумеется, ни один алгоритм не гарантирует безошибочного нахождения векторов движения. На практике негативное влияние на корректность определения векторов могут оказывать шумы, блок может сместиться на расстояние, выходящее за границы области поиска. Все эти факторы приводят к появлению ошибочно найденных векторов, которые затрудняют сегментацию.

Для снижения влияния аномальных векторов движения на результат обработки, полученное поле векторов целесообразно обработать – выполнить пространственную или временную фильтрацию.

***Рекурсивная векторная медианная фильтрация*** обеспечивает высокую эффективность обработки поля векторов в пределах одного кадра (рис 4). Под медианой множества векторов понимается такой вектор из рассматриваемого множества, у которого сумма расстояний до всех других минимальна. Расстояние между двумя векторами, *U*(*xu*, *yu*) и *V*(*xv*, *yv*) вычисляется на основе нормы *L2*:



В применении к задаче удаления аномальных векторов под медианной фильтрацией понимается замена каждого вектора движения векторной медианой множества, составленного из самого вектора и восьми его ближайших соседей. При вычислении расстояний используются только ненулевые векторы, т.е. каждый ненулевой вектор превращается в векторную медиану, вычисленную с помощью его восьми ненулевых соседей (рис. 4). Это делается для того, чтобы избежать замены ненулевых векторов на нулевые, когда в этом соседстве доминируют нулевые векторы.

Пространственная медианная фильтрация поля векторов движения позволяет повысить точность сегментации объектов интереса за счет устранения аномальных векторов движения и вызванных ими разрывов внутри сегментированного объекта. Однако объекты, которые по размерам меньше маски медианного фильтра, могут быть потеряны.

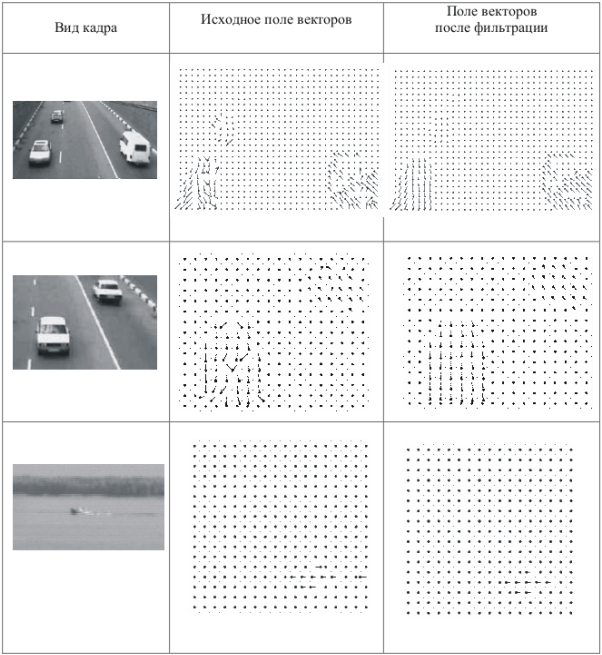


Рис. - Медианная фильтрация полей векторов движения

На заключительном этапе алгоритма сегментации объектов на основе векторов движения найденные векторы, прошедшие фильтрацию, объединяют в кластеры по признаку сонаправленности и пространственной связанности.

***Алгоритм кластеризации*** состоит из следующих шагов.

1. Последовательно перебирают все блоки изображения, для блоков с ненулевым вектором движения проверяют наличие метки, если метка отсутствует, то для данного блока формируют новую метку.
2. Вокруг блока с созданной меткой рассматривают 8 его соседей. Если встречается ненулевой вектор и принимается решение, что векторы сонаправленные, то текущая метка присваивается соседнему блоку и шаг 2 повторяется. В противном случае повторяют шаг 1.

Процедуру выполняют до тех пор, пока существуют неразмеченные блоки. Результатом обработки являются сегментированные движущиеся объекты.

Таким образом, алгоритм сегментации объектов на основе векторов движения состоит из следующих шагов:

* разбиение изображения на блоки;
* определение векторов движения;
* пространственная фильтрация поля векторов движения;
* кластеризация блоков, для которых определены ненулевые векторы движения, определение их принадлежности к соответствующим объектам.

**Сегментация объектов**

Исходные кадры представлены на рис. 5 и 6.



Рис. 5 – Первый кадр



Рис. – Второй кадр

Для поиска векторов движения использовался алгоритм 3SS с разделением на блоки по 5 пикселей. В качестве целевой функции использовалась функция MAD (рис. 7).



Рис. 7 – Найденные вектора движения

Вектора после проведения фильтрации изображены на рис. 8.



Рис. – Вектора движения после фильтрации

Так как полученные векторы имеют большой разброс, то в качестве критерия сонаправленности векторов для их маркировки был взят угол между векторами меньше 90 градусов и наличие восьмисвязности. Результат маркировки и покраски распознанных движущихся объектов представлен на рис. 9.



Рис. – Сегментация распознанных объектов

**Вывод**

В ходе работы был реализован алгоритм сегментации движущегося объекта в OpenCV: найдены вектора движения перемещающихся объектов на двух кадрах видеопоследовательности, произведена их фильтрация и сегментация объектов.

В результате объект, меняющий свое положение между кадрами, был корректно обнаружен, однако присутствуют паразитные вектора, направленные в заметно отличающихся от векторов объекта направлениях из-за пикселей, фактически занятых объектом после перемещения, которые были ошибочно найдены в ближайшем окружении.

**Листинг Кода**

**main.cpp:**

#include <iostream>

#include <string>

#include <unordered\_map>

#include <opencv2/opencv.hpp>

#include <utils/utils.h>

#include <clusterization.hpp>

int main(int argc, char\* argv[]) {

    int blockSize = 5;

    cv::Mat src\_prev\_colour;

    cv::Mat src\_cur\_colour;

    if ( argc <=2 ) {

        src\_prev\_colour = cv::imread("p1\_1.jpg");

        src\_cur\_colour = cv::imread("p1\_2.jpg");

    } else if ( argc == 3 ) {

        src\_prev\_colour = cv::imread(argv[1]);

        src\_cur\_colour = cv::imread(argv[2]);

    }

    std::unordered\_map<std::string, cv::Mat> output\_imgs;

    output\_imgs["vectors"] = src\_cur\_colour.clone();

    output\_imgs["filteredVectors"] = src\_cur\_colour.clone();

    output\_imgs["clusterized"] = src\_cur\_colour.clone();

    getVectorsImg(src\_prev\_colour, src\_cur\_colour, blockSize, output\_imgs);

    std::cout << std::endl;

    std::string output\_folder = "output";

    std::string create\_folder = "mkdir " + output\_folder;

    system(create\_folder.c\_str());

    output\_folder += "/";

    cv::imwrite(output\_folder + "vectors.jpg", output\_imgs["vectors"]);

    cv::imwrite(output\_folder + "filteredVectors.jpg", output\_imgs["filteredVectors"]);

    cv::imwrite(output\_folder + "clusterized.jpg", output\_imgs["clusterized"]);

    int confirm = 0;

    std::cout << "\nShow all? (0/1) ";

    std::cin >> confirm;

    if (confirm) {

        show\_img("inputPrev", src\_prev\_colour);

        show\_img("inputCur", src\_cur\_colour);

        show\_img("vectors", output\_imgs["vectors"]);

        show\_img("filteredVectors", output\_imgs["filteredVectors"]);

        show\_img("clusterized", output\_imgs["clusterized"]);

        cv::waitKey(0);

    }

    std::cout << "\n";

}

**clusterization.hpp (функции обработки):**

#include <opencv2/opencv.hpp>

#include <utils/utils.h>

namespace lab6

{

    struct Point : public cv::Point {

        using cv::Point::Point;

        Point(cv::Point &point)

        {

            x = point.x;

            y = point.y;

        }

        friend bool operator<(const Point& l, const Point& r)

        {

            return l.x < r.x ? true : (l.x == r.x ? l.y < r.y : false);

        }

        friend bool operator>(const Point& l, const Point& r)

        {

            return r < l;

        }

    };

    struct MoveVector {

        MoveVector(){}

        MoveVector(cv::Point \_start, cv::Point \_end) {

            start = \_start;

            end = \_end;

            coords = \_end - \_start;

        }

        // codirectional vectors

        friend bool operator==(const MoveVector& l, const MoveVector& r)

        {

            if ((l.coords.x / r.coords.x) == (l.coords.y / r.coords.y))

            {

                return 0 < (l.coords.x \* r.coords.x + l.coords.y \* r.coords.y);

            }

            return false;

        }

        double distance(const MoveVector& r)

        {

            return std::sqrt(std::pow(coords.x - r.coords.x, 2) + std::pow(coords.y - r.coords.y, 2));

        }

        double length()

        {

            return std::sqrt(std::pow(coords.x, 2) + std::pow(coords.y, 2));

        }

        // if positive --> <90 degrees

        double getScalarMult(const MoveVector& r)

        {

            return (coords.x \* r.coords.x) + (coords.y \* r.coords.y);

        }

        // if cos > 0.7 --> <45 degrees

        double getCos(MoveVector& r)

        {

            return (getScalarMult(r) / (length() \* r.length()));

        }

        void update(const cv::Point& newCoords)

        {

            coords = newCoords;

            end = start + newCoords;

        }

        cv::Point start;

        cv::Point end;

        cv::Point coords;

    };

}

void filterVectors(int stepSize, std::map<lab6::Point, lab6::MoveVector> &vectors, std::map<lab6::Point, lab6::MoveVector> &vectorsOut, bool fullFill)

{

    vectorsOut = vectors;

    std::cout << std::endl << "Filtering vectors... " << std::endl;

    for (int i = vectors.begin()->first.x + stepSize; i < vectors.rbegin()->first.x; i += stepSize)

    {

        for (int j = vectors.begin()->first.x + stepSize; j < vectors.rbegin()->first.y; j += stepSize)

        {

            if(vectors[lab6::Point(i, j)].coords == cv::Point(0, 0))

                continue;

            double minDist = INT\_MAX;

            cv::Point minCoords = cv::Point(0, 0);

            for (int vX = -stepSize; vX <= stepSize; vX += stepSize)

            {

                for (int vY = -stepSize; vY <= stepSize; vY += stepSize)

                {

                    if(vectors[lab6::Point(i + vX, j + vY)].coords == cv::Point(0, 0))

                        continue;

                    double curDist = 0;

                    for (int ii = -stepSize; ii <= stepSize; ii += stepSize)

                    {

                        for (int jj = -stepSize; jj <= stepSize; jj += stepSize)

                        {

                            if ((ii == vX && jj == vY) || vectors[lab6::Point(i + ii, j + jj)].coords == cv::Point(0, 0))

                                continue;

                            curDist += vectors[lab6::Point(i + vX, j + vY)].distance( vectors[lab6::Point(i + ii, j + jj)] );

                        }

                    }

                    if (curDist < minDist)

                    {

                        minDist = curDist;

                        minCoords = vectors[lab6::Point(i + vX, j + vY)].coords;

                    }

                }

            }

            if (fullFill)

            {

                for (int vX = -stepSize; vX <= stepSize; vX += stepSize)

                {

                    for (int vY = -stepSize; vY <= stepSize; vY += stepSize)

                    {

                        vectorsOut[lab6::Point(i + vX, j + vY)].update(minCoords);

                    }

                }

            } else {

                vectorsOut[lab6::Point(i, j)].update(minCoords);

            }

        }

        progressbar((float)(vectors.rbegin()->first.x - stepSize), i, 10);

    }

}

void drawVectors(const cv::Mat &input\_img, cv::Mat &output\_img, std::map<lab6::Point, lab6::MoveVector> &vectors, int shift)

{

    output\_img = input\_img.clone();

    cv::Point shiftPoint(shift, shift);

    std::cout << std::endl << "Drawing silly little arrows... " << std::endl;

    int cur = 0;

    for (auto &vector : vectors)

    {

        cv::arrowedLine(output\_img, vector.second.start + shiftPoint, vector.second.end + shiftPoint, cv::Scalar(255, 0, 0, 255));

        progressbar((float)(vectors.size()), ++cur, 5);

    }

}

double getMSECorrelation(const cv::Mat &previous, const cv::Mat &current)

{

    double sumMSE = 0;

    cv::Mat diff;

    cv::absdiff(previous, current, diff);

    for (int i = 0; i < diff.cols; ++i)

    {

        for (int j = 0; j < diff.rows; ++j)

        {

            sumMSE += std::pow(diff.at<uchar>(j, i), 2);

        }

    }

    return (sumMSE / (diff.rows \* diff.cols));

}

double getMADCorrelation(const cv::Mat &previous, const cv::Mat &current)

{

    double sumMAD = 0;

    cv::Mat diff;

    cv::absdiff(previous, current, diff);

    for (int i = 0; i < diff.cols; ++i)

    {

        for (int j = 0; j < diff.rows; ++j)

        {

            sumMAD += diff.at<uchar>(j, i);

        }

    }

    return (sumMAD / (diff.rows \* diff.cols));

}

cv::Point compare\_3SS(const cv::Mat &previousBlock, const cv::Mat &currentFrame, int step, int iteration, cv::Point centralBlock)

{

    std::map<double, cv::Point> blocksResults;

    for (int i = centralBlock.x - step; i <= centralBlock.x + step; i += step)

    {

        for (int j = centralBlock.y - step; j <= centralBlock.y + step; j += step)

        {

            double result = getMSECorrelation(previousBlock, currentFrame(cv::Rect(i, j, previousBlock.cols, previousBlock.rows)));

            blocksResults[result] = cv::Point(i, j);

        }

    }

    if (iteration == 3) {

        return blocksResults.begin()->second;

    } else {

        return compare\_3SS(previousBlock, currentFrame, step/2, iteration+1, blocksResults.begin()->second);

    }

}

cv::Point compare\_FS(const cv::Mat &previousBlock, const cv::Mat &currentFrame)

{

    cv::Mat result;

    int result\_cols = currentFrame.cols - previousBlock.cols + 1;

    int result\_rows = currentFrame.rows - previousBlock.rows + 1;

    result.create(result\_rows, result\_cols, CV\_32FC1);

    matchTemplate(currentFrame, previousBlock, result, cv::TemplateMatchModes::TM\_CCOEFF);

    normalize(result, result, 0, 1, cv::NORM\_MINMAX, -1, cv::Mat());

    double minVal;

    double maxVal;

    cv::Point minLoc;

    cv::Point maxLoc;

    minMaxLoc(result, &minVal, &maxVal, &minLoc, &maxLoc, cv::Mat());

    return minLoc;

}

void fillArea(const cv::Mat &input\_img, cv::Mat &output\_img, const lab6::Point &topLeftCoords, std::vector<int> &color, int blockSize)

{

    for (int i = 0; i < blockSize; ++i)

    {

        for (int j = 0; j < blockSize; ++j)

        {

            cv::Vec3b pixel\_val = input\_img.at<cv::Vec3b>(topLeftCoords.y + j, topLeftCoords.x + i);

            for (int k = 0; k < pixel\_val.channels; ++k)

                output\_img.at<cv::Vec3b>(topLeftCoords.y + j, topLeftCoords.x + i)[k]

                    = pixel\_val[k] + color[k] < 255 ? pixel\_val[k] + color[k] : 255;

        }

    }

}

void recursiveClassify(cv::Mat &mask, std::map<lab6::Point, std::vector<int>> &colorizedVectors, std::map<lab6::Point, lab6::MoveVector> &vectors, std::pair<const lab6::Point, lab6::MoveVector> &curVector, int blockSize, int iter)

{

    for (int i = -blockSize; i <= blockSize; i += blockSize)

    {

        for (int j = -blockSize; j <= blockSize; j += blockSize)

        {

            lab6::Point neighbourVector(curVector.first.x + i, curVector.first.y + j);

            if (neighbourVector == curVector.first || mask.at<uchar>(neighbourVector) != 0)

                continue;

            if (vectors.count(neighbourVector) != 0 && vectors[neighbourVector].coords != cv::Point(0, 0))

            {

                if (curVector.second.getCos(vectors[neighbourVector]) >= 0.0)

                {

                    mask.at<uchar>(neighbourVector) = mask.at<uchar>(curVector.first);

                    colorizedVectors[neighbourVector] = colorizedVectors[curVector.first];

                    if (iter < 1)

                        recursiveClassify(mask, colorizedVectors, vectors, std::pair<const lab6::Point, lab6::MoveVector>(neighbourVector, vectors[neighbourVector]), blockSize, iter+1);

                }

            }

        }

    }

}

void classify(const cv::Mat &input\_img, cv::Mat &output\_img, std::map<lab6::Point, lab6::MoveVector> &vectors, int blockSize)

{

    output\_img = input\_img.clone();

    cv::Mat mask = cv::Mat::zeros(input\_img.size(), CV\_8U);

    std::map<lab6::Point, std::vector<int>> colorizedVectors;

    std::cout << std::endl << "Adding some colours... " << std::endl;

    int cur = 0;

    for (auto &vector : vectors)

    {

        if (vector.second.coords == cv::Point(0, 0))

            continue;

        if (mask.at<uchar>(vector.first) == 0)

        {

            mask.at<uchar>(vector.first) = std::rand()%254;

            for (int k = 0; k < 3; ++k)

                colorizedVectors[vector.first].push\_back(std::rand()%254);

        }

        recursiveClassify(mask, colorizedVectors, vectors, vector, blockSize, 0);

        fillArea(input\_img, output\_img, vector.first, colorizedVectors[vector.first], blockSize);

        progressbar((float)(vectors.size()), ++cur, 5);

    }

    cv::imwrite("output/mask.jpg", mask);

}

void getVectorsImg(const cv::Mat &previous\_img, const cv::Mat &current\_img, int blockSize, std::unordered\_map<std::string, cv::Mat> &output\_imgs)

{

    cv::Mat src\_prev;

    cv::Mat src\_cur;

    cv::cvtColor(previous\_img, src\_prev, cv::COLOR\_BGR2GRAY);

    cv::cvtColor(current\_img, src\_cur, cv::COLOR\_BGR2GRAY);

    cv::copyMakeBorder(src\_prev, src\_prev, blockSize\*7, blockSize\*7, blockSize\*7, blockSize\*7, cv::BORDER\_CONSTANT, cv::Scalar(0));

    cv::copyMakeBorder(src\_cur, src\_cur, blockSize\*7, blockSize\*7, blockSize\*7, blockSize\*7, cv::BORDER\_CONSTANT, cv::Scalar(0));

    cv::Mat prevBlock;

    cv::Mat curBlock;

    std::map<lab6::Point, lab6::MoveVector> vectors;

    std::map<lab6::Point, lab6::MoveVector> vectorsFiltered;

    cv::Point destination;

    std::cout << "Finding vectors..." << std::endl;

    for (int i = blockSize\*7; i < src\_cur.cols - blockSize\*7; i += blockSize)

    {

        for (int j = blockSize\*7; j < src\_cur.rows - blockSize\*7; j += blockSize)

        {

            prevBlock = src\_prev(cv::Rect(i, j, blockSize, blockSize));

            vectors[lab6::Point(i, j)] = lab6::MoveVector(cv::Point(i, j), compare\_3SS(prevBlock, src\_cur, 4\*blockSize, 1, cv::Point(i, j)));

            // vectors[lab6::Point(i, j)] = lab6::MoveVector(cv::Point(i, j), cv::Point(i - blockSize, j - blockSize) + compare\_FS(prevBlock, src\_cur(cv::Rect(i - blockSize\*3, j - blockSize\*3, blockSize\*3, blockSize\*3))));

        }

        progressbar((float)(src\_cur.cols - blockSize\*8), i);

    }

    cv::Mat src\_bordered;

    cv::copyMakeBorder(previous\_img, src\_bordered, blockSize\*7, blockSize\*7, blockSize\*7, blockSize\*7, cv::BORDER\_CONSTANT, cv::Scalar(0));

    filterVectors(blockSize, vectors, vectorsFiltered, false);

    drawVectors(src\_bordered, output\_imgs["vectors"], vectors, blockSize/2);

    drawVectors(src\_bordered, output\_imgs["filteredVectors"], vectorsFiltered, blockSize/2);

    classify(src\_bordered, output\_imgs["clusterized"], vectorsFiltered, blockSize);

    for (auto &imgs: output\_imgs)

    {

        imgs.second = imgs.second(cv::Rect(blockSize\*7, blockSize\*7, previous\_img.cols, previous\_img.rows));

    }

}