Санкт-Петербургский политехнический университет Петра Великого
Институт металлургии, машиностроения и транспорта
Кафедра компьютерных технологий в машиностроении

Лабораторная работа №2

По дисциплине: Техническое зрение

Студент гр. 3331506/70401 Кочурин Р.П.
Преподаватель Варлашин В.В.

« »_____2020 г.

Санкт-Петербург
2020

Цель работы

Приобрести первоначальные навыки обработки изображения с применением библиотек OpenCV.

Задание

Реализовать метод обработки изображения, метод Оцу.

Выполнение

Реализация метода вынесена в отдельную библиотеку, в которой лишь одна функция, которую можно вызвать при подключении библиотеки, а также несколько вспомогательных, являющимися статическими.

Листинг вызываемой функции извне представлен далее:

```
int OtsuMethod(const Mat& src, Mat& dst)
{
	if (src.empty())
	{
		return -1;
	}
	if (dst.empty())
	{
		return -2;
	}
	int histogramm[256];
	BrightnessHistogram(src, histogramm);
	float probability[256];
	int totalPixel = src.cols * src.rows;
	//определение вероятности появления каждой яркости
	for (int i = 0; i < 256; i++)
	{
		probability[i] = (float)(histogramm[i]) / (float)(totalPixel);
	}
	//вероятность появления яркости до уровня k включительно
	float accumulationAmountP[256];
	//накопленная сумма до уровня яркости k
	float accumulationAmountM[256];
	for (int k = 0; k < 256; k++)
	{
		accumulationAmountP[k] = CalculateAccumulationAmountP(probability, k);
		accumulationAmountM[k] = CalculateAccumulationAmountM(probability, k);
	}
	float accumulationAmountMG = accumulationAmountM[255];
	//межклассовая дисперсия при границе k
	float interclassDispersion[256];
	//максимум межклассовой дисперсии
	float maxInterclassDispersion = 0;
	for (int k = 0; k < 256; k++)
	{
		if (accumulationAmountP[k] != 0)
		{
			interclassDispersion[k] = (((float)accumulationAmountMG *
(float)accumulationAmountP[k] - (float)accumulationAmountM[k]) *
				((float)accumulationAmountMG * (float)accumulationAmountP[k] -
(float)accumulationAmountM[k])) /
				((float)accumulationAmountP[k] * (1.0 -
(float)accumulationAmountP[k]));
		}
		else
		{
			interclassDispersion[k] = 0;
		}
		if (maxInterclassDispersion <= interclassDispersion[k])
		{
			maxInterclassDispersion = interclassDispersion[k];
		}
	}
	//определяем количество соответствий максимуму и находим среднее значение
```

```
	int finalThreshold = 0;
	{
		int tempSum = 0;
		int tempAmount = 0;
		for (int i = 0; i < 256; i++)
		{
			if ((int)(maxInterclassDispersion) <= (int)(interclassDispersion[i]))
			{
				tempSum += i;
				tempAmount++;
			}
		}
		finalThreshold = tempSum / tempAmount;
	}
	imageConversion(src, dst, finalThreshold);
	imshow("Before", src);
	imshow("After", dst);
	WAIT;
	return 0;
}
```

В начале данной функции происходит проверка на наличие обрабатываемого файла, то есть является ли переменная *Mat* пустой, и в случае, если является, происходит выход из функции с кодом -1, а также происходит проверка изображения, куда будет записываться обрабатываемое изображения и такая же проверка, но выход происходит с ошибкой -2.

Далее происходит вычисление яркостной гистограммы изображения с помощью функции *BrightnessHistogram*, в которую передаётся изображение и массив для записи гистограммы, описание самой функции будет представлено позже. Создаётся массив *histogram*, в котором содержится 256 переменных, что соответствует 256-ти возможным случаям яркости изображения. В каждой из перемен хранится количество пикселей, которым соответствует *i*-ая яркость (*i* – номер переменной в массиве).

В листинге программы присутствуют комментарии, которые поясняют дальнейшую работу функции.

При вычислении межклассовой дисперсии (*interclassDispersion*) использовалась следующая формула:

$$\sigma_B^2(k) = \frac{\left[m_G P_1(k) - m(k)\right]^2}{P_1(k)\left[1 - P_1(k)\right]},$$

где $m(k)$ – накопленная сумма до уровня яркости k включительно (*accumulationAmountM*), которая задаётся следующим соотношением:

$$m(k) = \sum_{i=0}^{k} ip_i;$$

p_i – вероятность появления i-ой яркости (*probability*);

m_G – накопленная сумма, которой соответствует $m(k)$ при $k = 255$ (*accumulationAmountMG*);

$P_1(k)$ – накопленная сумма вероятностей появления яркости в диапазоне от 0 до k (*accumulationAmountP*), которая задаётся следующим соотношением:

$$P_1(k) = \sum_{i=0}^{k} p_i.$$

Сегментирование изображение происходит по уровню яркости *finalThreshold* следующим образом:

$$g(x,y) = \begin{cases} 1, & \text{если } f(x,y) > k \\ 0, & \text{если } f(x,y) \le k, \end{cases}$$

где $f(x,y)$ – яркость пикселя (x,y) исходного изображения;

$g(x,y)$ – значение пикселя (x,y) обработанного изображения;

k – уровень сегментирования.

Листинг функции вычисления гистограммы представлен далее:

```
static int BrightnessHistogram(const Mat & image, int* value)
{
	for (int i = 0; i < 256; i++)
	{
		value[i] = 0;
	}
	for (int i = 0; i < image.rows; i++)
	{
		for (int j = 0; j < image.cols; j++)
		{
			//[0] - B; [1] - G; [2] - R
			int brightness = (int)(0.3 * (float)(image.at<Vec3b>(i, j)[2]) +
0.59 * (float)(image.at<Vec3b>(i, j)[1]) + 0.11 * (float)(image.at<Vec3b>(i, j)[0]));
			value[brightness]++;
		}
	}
	DrawBrightnessHistogram(image.rows * image.cols, value);
	return 0;
}
```

Вычисление яркости пикселя изображения происходит по следующей формуле:

$$b = 0.3R + 0.59G + 0.11B,$$

где R – значение, соответствующее красному цвету пикселя;

G – значение, соответствующее зелёному цвету пикселя;

B – значение, соответствующее синему цвету пикселя.

После реализации метода Оцу были проведено три тестирования на рисунке 1 представлены результаты первого теста, рисунке 2 – второго.

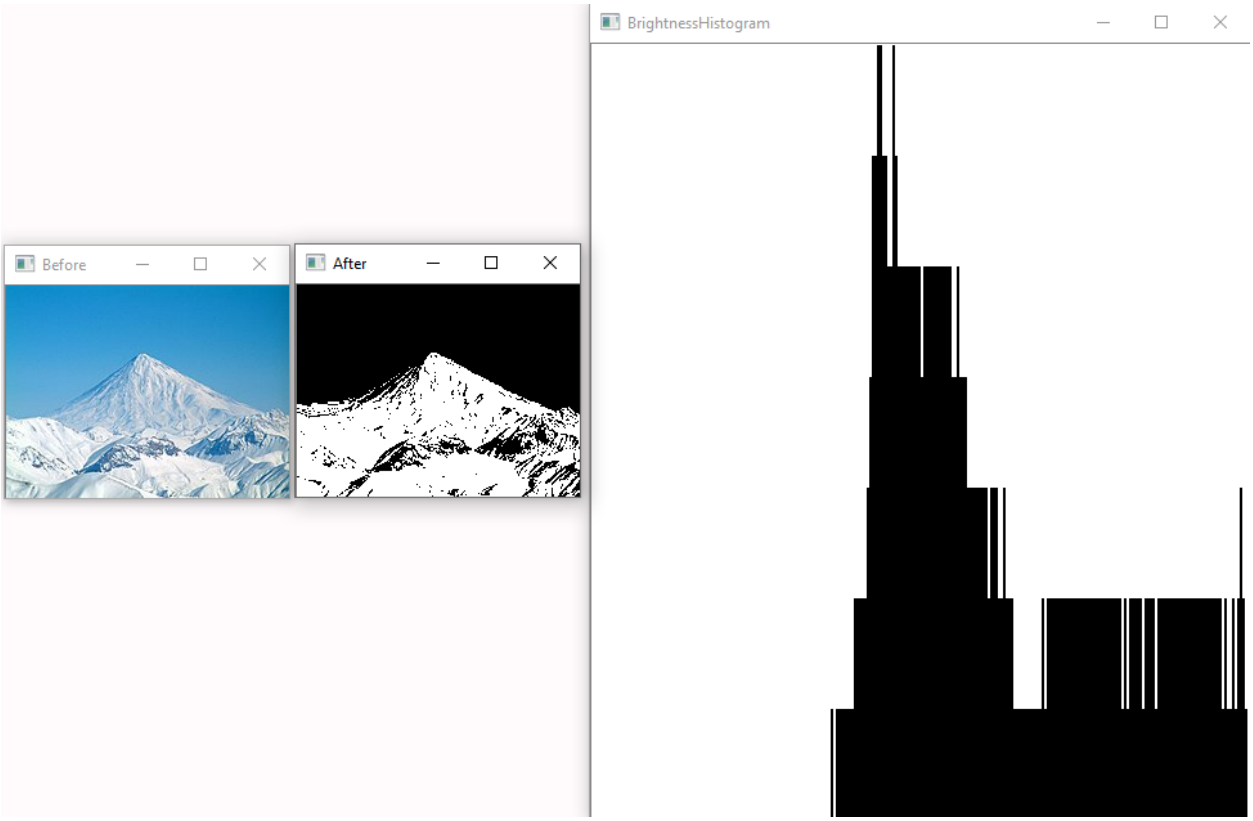

Рисунок 1 – Результаты первого тестирования

Рисунок 2 – Результаты второго тестирования

Далее была проведена проверка корректности работы реализованного алгоритма с помощью стандартной функции метода Оцу. Но так как стандартная функция может работать только с одноканальными изображениями, то проверка также проводилась с изображением, которое было разделено на отдельные цвета и преобразовано в одноканальное изображение.

Для нахождения уровня, используемого в обработке стандартной функцией, использовалась следующая функция:

```
static void searchTresholdStanFun(const Mat& image, const Mat& compared)
{
	Mat dst(image.rows, image.cols, CV_8UC1, Scalar(255));
	cout << "Threshold standart function: ";
	int delta[256];
	for (int i = 0; i < 256; i++)
	{
		delta[i] = 0;
		com_imageConversion(image, dst, i);
		for (int j = 0; j < image.rows; j++)
		{
			for (int k = 0; k < image.cols; k++)
			{
				if (compared.at<uchar>(j, k) != dst.at<uchar>(j, k))
				{
					delta[i]++;
				}
			}
		}
		if (delta[i] == 0)
		{
			cout << i << "; ";
		}
	}
	cout << endl;
	return;
	}
```

В результате проверки корректности работы были получены следующие результаты:

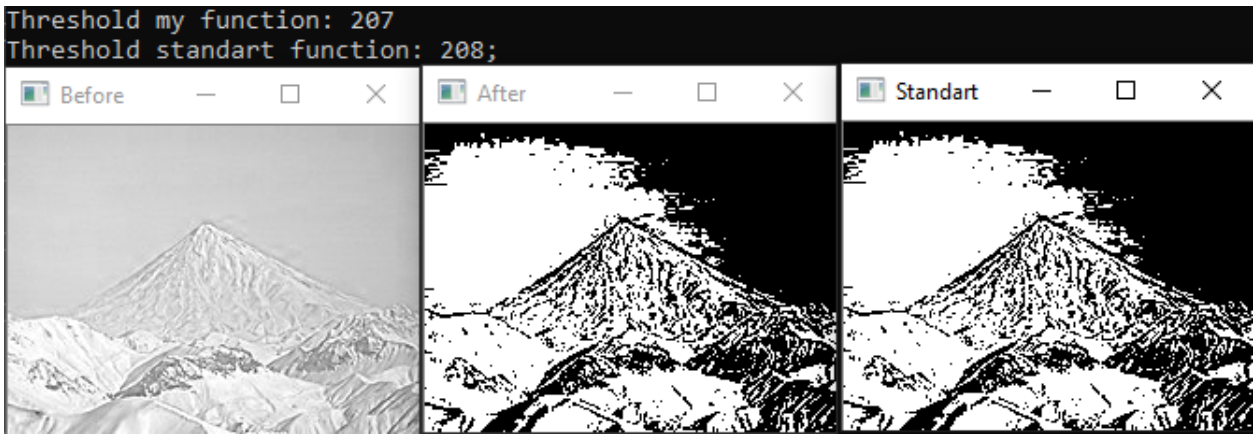

Рисунок 3 – Проверка в синем цвете

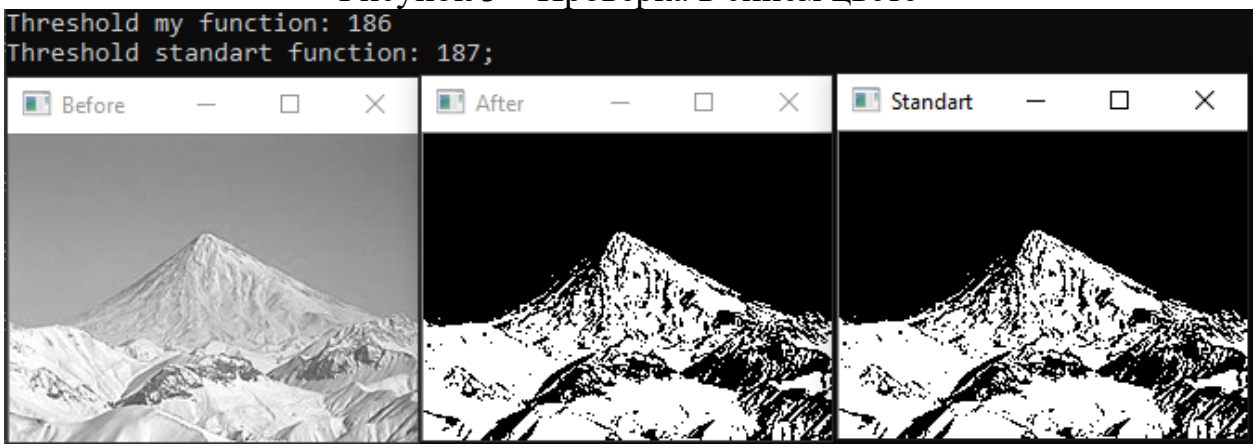

Рисунок 4 – Проверка в зелёном цвете

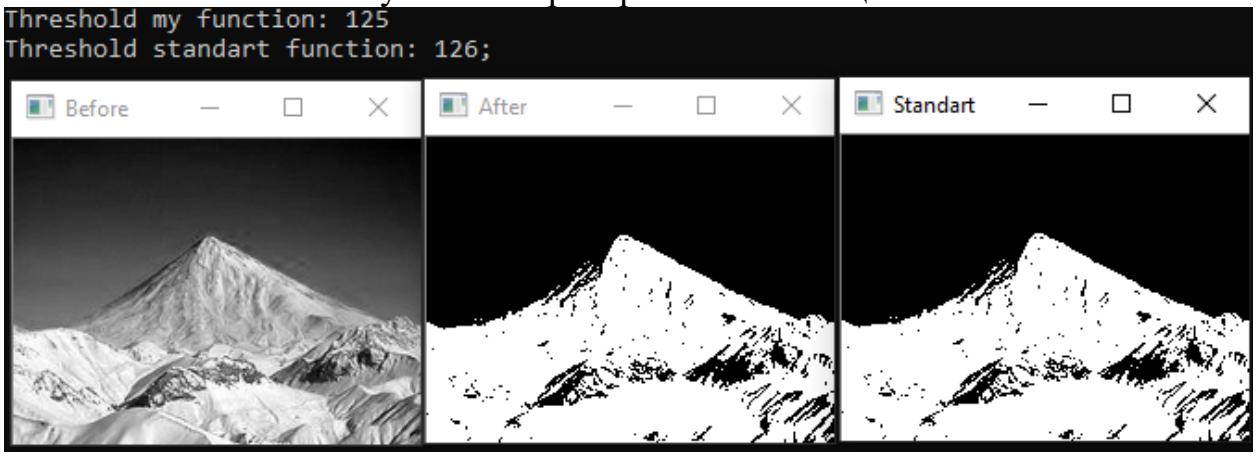

Рисунок 5 – Проверка в красном цвете

Как мы видим изображения получаются очень и очень близкие, а уровни, используемые в моей и стандартной функциях, отличаются на единицу, при этом уровень стандартной больше моего уровня.

Вывод

В ходе выполнения работы были приобретены первичные навыки обработки изображений с использованием библиотек OpenCV, вследствие чего был реализована поставленная задача.

Была проведена проверка корректности работы реализованной функции при помощи сравнения со стандартной функцией, присутствующей в OpenCV. Было подтверждена корректность работы реализованной функцией.