Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Отчёт

по лабораторной работе №2

Дисциплина: Техническое зрение

Тема: Фильтрация изображения с использованием библиотеки OpenCV

Студент гр. 3331506/70401 Демчева А.А.

Преподаватель Варлашин В.В.

« »_____2020 г.

Санкт-Петербург

2020

Задание

Написать программу, реализующую сглаживание изображения с помощью фильтра Гаусса. Размер ядра — 7x7, якорь в центре. Вариант обработки граничных условий — повторение.

При выполнении использовать средства библиотеки *OpenCV 4.0*.

Выполнение задания

При выполнении задания были использованы возможности классов языка C++. Все функции были написаны в виде методов класса *myGaussianBlur*, интерфейс которого был объявлен в заголовочном файле *myGaussianBlur.h*, а реализация описана в файле *myGaussianBlur.cpp*.

Функция *setSrcImage* задает исходное изображение, копируя его в глобальную переменную класса.

Листинг 1 — Функция setSrcImage

```
void myGaussianBlur::setSrcImage(Mat srcImage)
{
	m_srcImage = srcImage.clone();
}
```

Функция *setKernel* формирует ядро в виде объекта *Mat* с учетом заданных размеров и величины среднеквадратичного отклонения. Элементы ядра вычисляются по формуле:

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

где x и y — расстояние от центра ядра, пикс; σ — среднеквадратичное отклонение, параметр, определяющий степень сглаживания.

Переменные x и y рассчитываются с учетом положения системы координат изображения в OpenCV (левый верхний угол) и положения якоря (в центре).

Результат работы функции показан на рисунке 2.

Листинг 2 — Функция setKernel

```
void myGaussianBlur::setKernel(double sigma, int kernelRows, int kernelCols)
{
	m_sigma = sigma;
	m_kernel = Mat(kernelRows, kernelCols, CV_64F);

	int i = 0;
	int j = 0;

	cout << "Ядро фильтра Гаусса: " << endl;


	for (j = 0; j < m_kernel.rows; j++)
	{
		int y = j - 3;
		for (i = 0; i < m_kernel.cols; i++)
		{
			int x = i - 3;
			m_kernel.at<double>(j, i) = (1 / (2 * CV_PI * pow(m_sigma, 2))) *
					exp(-(pow(x, 2) + pow(y, 2)) / (2 * pow(m_sigma, 2)));
			m_kernelSum = m_kernelSum + m_kernel.at<double>(j, i);
			cout << m_kernel.at<double>(j, i) << "	";
		}
		cout << endl;
	}
	cout << endl << endl;
	return;
}
```

```
Ядро фильтра Гаусса:
0.00145457	0.00149139	0.00151393	0.00152152	0.00151393	0.00149139	0.00145457
0.00149139	0.00152914	0.00155225	0.00156003	0.00155225	0.00152914	0.00149139
0.00151393	0.00155225	0.00157571	0.00158361	0.00157571	0.00155225	0.00151393
0.00152152	0.00156003	0.00158361	0.00159155	0.00158361	0.00156003	0.00152152
0.00151393	0.00155225	0.00157571	0.00158361	0.00157571	0.00155225	0.00151393
0.00149139	0.00152914	0.00155225	0.00156003	0.00155225	0.00152914	0.00149139
0.00145457	0.00149139	0.00151393	0.00152152	0.00151393	0.00149139	0.00145457
```

Рисунок 2 — Результат работы функции setKernel

Get-функции *getSrcImg*, *getExpandedImg*, *getResultImg*, *getDiffImg* осуществляют вывод изображения после каждого этапа обработки.

Листинг 3 — Get-функции класса myGaussianBlur

```
void myGaussianBlur::getSrcImg()
{
    imshow("myGauss source image", m_srcImage);
}

void myGaussianBlur::getExpandedImg()
{
    imshow("myGauss expanded image", m_extImage);
}

void myGaussianBlur::getResultImg()
{
    imshow("myGauss result image", m_resImage);
}
```

```
void myGaussianBlur::getDiffImg()
{
    imshow("difference image", m_diffImg);
}
```

Функция *expandImage* реализует обработку граничных условий в соответствии с правилом *border replicate*. Создается дополнительный объект типа *Mat* — расширенное изображение, в центр которого копируется исходное. Оставшиеся пустыми граничные области заполняются значением интенсивности, равным значению соответствующего крайнего пикселя исходного изображения.

Листинг 4 — Функция expandImage

```
void myGaussianBlur::expandImage()
{
    int myGaussRows = m_srcImage.rows + m_kernel.rows - 1;
    int myGaussCols = m_srcImage.cols + m_kernel.cols - 1;
    int myGaussType = m_srcImage.type();

    m_extImage = Mat(myGaussRows, myGaussCols, myGaussType);

    m_Dot = Point2i((m_kernel.rows / 2), (m_kernel.cols / 2));

    Size2i Size;
    Size.width = m_srcImage.cols;
    Size.height = m_srcImage.rows;
    Rect Rectangle(m_Dot, Size);
    Mat Roi(m_extImage(Rectangle));

    m_srcImage.copyTo(Roi);//Копируем в расширенное изображение исходное


    int leftBorder = (m_kernel.cols / 2);
    int rightBorder = (m_kernel.cols / 2) + m_srcImage.cols - 1;

    int topBorder = (m_kernel.rows / 2);
    int bottomBorder = (m_kernel.rows / 2) + m_srcImage.rows - 1;

    //верхняя граница
    for (int channel = 0; channel < m_srcImage.channels(); channel++) //каналы пикселя
    {
           for (int i = leftBorder; i <= rightBorder; i++)
           {
                  for (int j = (topBorder - 1); j >= 0; j--)
                  {
                         m_extImage.at<Vec3b>(j, i)[channel] =
                                                m_extImage.at<Vec3b>(topBorder,i)[channel];
                  }
           }
    }

    //нижняя граница
    for (int channel = 0; channel < m_srcImage.channels(); channel++) //каналы пикселя
```

```
    {
        for (int i = leftBorder; i <= rightBorder; i++)
        {
            for (int j = (bottomBorder + 1); j < m_extImage.rows; j++)
            {
                m_extImage.at<Vec3b>(j, i)[channel] =
                                m_extImage.at<Vec3b>(bottomBorder, i)[channel];
            }
        }
    }

    //левая граница
    for (int channel = 0; channel < m_srcImage.channels(); channel++) //каналы пикселя
    {
        for (int j = 0; j < m_extImage.rows; j++)
        {
            for (int i = (leftBorder - 1); i >= 0; i--)
            {
                m_extImage.at<Vec3b>(j, i)[channel] =
                                m_extImage.at<Vec3b>(j,leftBorder)[channel];
            }
        }
    }

    //правая граница
    for (int channel = 0; channel < m_srcImage.channels(); channel++) //каналы пикселя
    {
        for (int j = 0; j < m_extImage.rows; j++)
        {
            for (int i = (rightBorder + 1); i < m_extImage.cols; i++)
            {
                m_extImage.at<Vec3b>(j, i)[channel] =
                                m_extImage.at<Vec3b>(j, rightBorder)[channel];
            }
        }
    }

    return;
}
```

Функция *smooth* реализует непосредственно сглаживание — перемещение ядра по исходному изображению, умножение значений интенсивности его пикселей на соответствующий коэффициент ядра, суммирование и запись результата в центральный пиксель.

Листинг 5 — Функция smooth

```
void myGaussianBlur::smooth()
{
    m_resImage = Mat(m_srcImage.rows, m_srcImage.cols, m_srcImage.type());

    double loc_sum = 0;

    int leftBorder = m_kernel.cols / 2;
    int rightBorder = m_extImage.cols - (m_kernel.cols / 2) - 1;

    int topBorder = m_kernel.rows / 2;
```

```
	int bottomBorder = m_extImage.rows - (m_kernel.rows / 2) - 1;


	int exImX = 0; //Координаты пикселя расширенного изображения
	int exImY = 0;

	int resImX = 0; //Координаты пикселя итогового изображения
	int resImY = 0;


	for (m_anchor.y = topBorder; m_anchor.y <= bottomBorder; m_anchor.y++)
	{
		for (m_anchor.x = leftBorder; m_anchor.x <= rightBorder; m_anchor.x++)
		{
			for (int channel = 0; channel < m_resImage.channels(); channel++)
			{
				//Устанавливаем координаты левой верхней точки на изображении,
				к которой будет приложена левая верхняя точка маски
				exImX = m_anchor.x - (m_kernel.cols / 2);
				exImY = m_anchor.y - (m_kernel.rows / 2);

				for (int kerY = 0; kerY < m_kernel.rows; kerY++)
				{
					for (int kerX = 0; kerX < m_kernel.cols; kerX++)
					{
						loc_sum = loc_sum +
						(m_kernel.at<double>(kerY,kerX) *
					      * m_extImage.at<Vec3b>(exImY, exImX)[channel]);
						exImX++;
					}
					//Возвращаемся в начало строки
					exImX = m_anchor.x - (m_kernel.cols / 2);
					//Переходим к следующей строке
					exImY++;
				}

				resImX = m_anchor.x - leftBorder;
				resImY = m_anchor.y - topBorder;

				m_resImage.at<Vec3b>(resImY, resImX)[channel] =
                                                (uint8_t)(loc_sum/m_kernelSum);
				loc_sum = 0;
			}

		}
	}
	return;
}
```

Чтобы убедиться, что написанная программа работает верно, результат применения пользовательского фильтра сравнивается со встроенной функцией *GaussianBlur* библиотеки *OpenCV* с помощью следующих методов класса.

Функция *calcDiff* реализует поэлементное сравнение полученных изображений с помощью метода *absdiff*.

Листинг 6 — Функция calcDiff

```
void myGaussianBlur::calcDiff(Mat builtInImg)
{
	int difRows = m_srcImage.rows;
	int difCols = m_srcImage.cols;
	int difType = m_srcImage.type();

	m_diffImg = Mat(difRows, difCols, difType);

	absdiff(m_resImage, builtInImg, m_diffImg);

	return;
}
```

Функция *squareDeviation* вычисляет среднеквадратичную погрешность фильтрации для каждого канала в соответствии с формулой:

$$\varepsilon = \sqrt{\frac{\sum_{j=0}^{j=M-1}\sum_{i=0}^{i=N-1}(I_{ij} - I_{y\ ij})^2}{N \cdot M - 1}}$$

где N и M — ширина и высота исходного изображения соответственно, пикс; I_{ij} и $I_{y\ ij}$— интенсивности пикселей изображений, полученных после применения библиотечного и пользовательского фильтров Гаусса. Разность $I_{ij} - I_{y\ ij}$ соответствует значению интенсивности пикселей изображения, полученного с помощью функции *absdiff*.

Листинг 7 — Функция squareDeviation

```
void myGaussianBlur::squareDeviation()
{
	double diffSum = 0;
	double difSize = (m_diffImg.rows * m_diffImg.cols);

	for (int channel = 0; channel < m_srcImage.channels(); channel++)
	{
		for (int j = 0; j < m_diffImg.rows; j++)
		{
			for (int i = 0; i < m_diffImg.cols; i++)
			{
			    diffSum = diffSum + pow((m_diffImg.at<Vec3b>(j, i)[channel]), 2);
			}
		}

		double squareDeviation = sqrt(diffSum / (difSize-1));
		cout << "канал  " << channel << endl;
		cout << "Среднеквадратичное отклонение = " << squareDeviation << endl;
	}

	return;
}
```

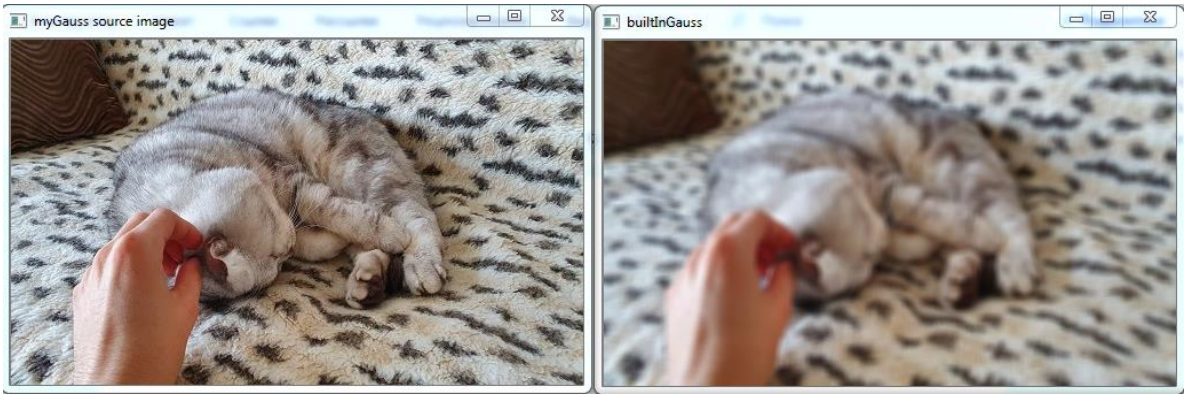
Подсчитывается также число тактов и время, затраченное на выполнение пользовательской и встроенной функции фильтра Гаусса — с помощью функции *clock* и макроса *CLOCKS_PER_SEC*.

Листинг 8 — Функция Main

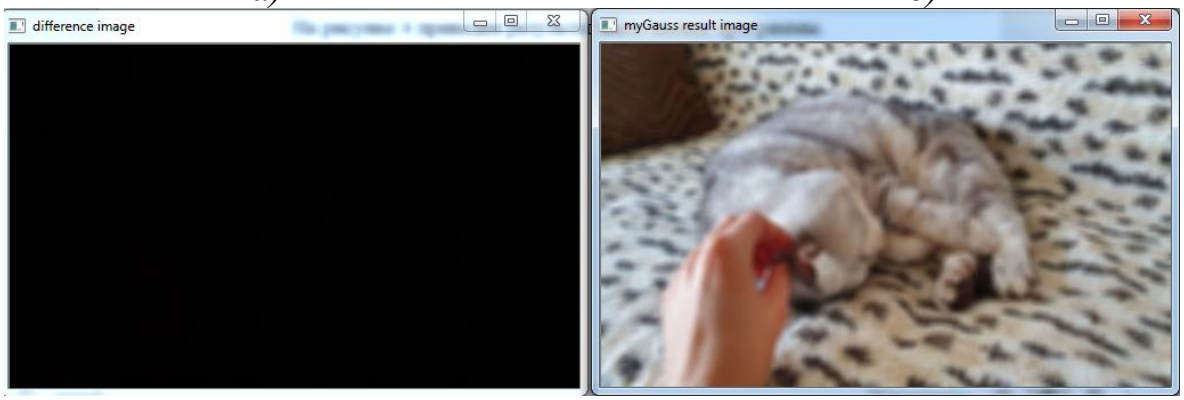
```
int main()
{
	setlocale(LC_ALL, "Russian");

	myGaussianBlur myGauss;

	double sigma = 10;
	int kernelRows = 7;
	int kernelCols = 7;

//Ядро
	myGauss.setKernel(sigma, kernelRows, kernelCols);

//Оригинальное изображение
	Mat image = imread("C:/222.bmp", 1);
	cout << "source image chanels = " << image.channels() << endl;
	myGauss.setSrcImg(image);
	myGauss.getSrcImg();

//Встроенный фильтр
	Mat builtInGaussImg = image.clone();

	int builtInStart = clock();
	GaussianBlur(image, builtInGaussImg, Size2i(7, 7), 10, 10, 1);
	int builtInEnd = clock();

	imshow("builtInGauss", builtInGaussImg);

	cout << "Число тактов для выполнения встроенной функции:  " << (builtInEnd -
builtInStart) << endl;
	cout << "Время выполнения встроенной функции:  " << ((double)(builtInEnd -
builtInStart) / CLOCKS_PER_SEC) << endl;

//Пользовательский фильтр Гаусса
	myGauss.expandImage();
	myGauss.getExpandedImg();

	int myGaussStart = clock();
	myGauss.smooth();
	int  myGaussEnd = clock();
	myGauss.getResultImg();

	cout << "Число тактов для выполнения пользовательской функции: "  << (myGaussEnd -
myGaussStart) << endl;
	cout << "Время выполнения пользовательской функции: " << ((double)(myGaussEnd -
myGaussStart) / CLOCKS_PER_SEC) << endl;

//Вычисляем разницу
	myGauss.calcDiff(builtInGaussImg);
	myGauss.getDiffImg();
	myGauss.squareDeviation();
```

```
        while (waitKey(1) != 27)
        {
                ;
        }

        return 0;
}
```

На рисунке 4 приведен результат выполнения программы: а) — исходное изображение; б), г) — результат применения встроенного и пользовательского фильтров соответственно; в) — разница между изображениями, полученными в результате применения фильтров; г) — сравнение времени выполнения функций и числа затраченных тактов.



а) *б)*



в) *г)*

```
source image chanels = 3
Число тактов для выполнения встроенной функции:  14
Время выполнения встроенной функции:  0.014
Число тактов для выполнения пользовательской функции: 148
Время выполнения пользовательской функции: 0.148
канал  0
Среднеквадратичное отклонение = 1.63152
канал  1
Среднеквадратичное отклонение = 2.32991
канал  2
Среднеквадратичное отклонение = 2.89544
```

д)

Рисунок 4 — Результат выполнения программы

В таблице 1 приведено сравнение скорости выполнения встроенной и пользовательской функций. Как можно заметить, пользовательская функция требует для выполнения почти в 10 раз большего числа тактов и времени.

Таблица 1 — Сравнение встроенного и пользовательского фильтра Гаусса

Параметр сравнения	***Фильтр***	
	Встроенный	***Пользовательский***
Число затраченных тактов	11	109
Время выполнения, с	0.011	0.109

Чтобы сделать разницу между примененными фильтрами более заметной, умножим значение яркости в каждом пикселе разностного изображения на некоторый масштабирующий коэффициент. Получим результат, показанный на рисунке 5.

Разницу между итоговыми изображениями можно объяснить округлением чисел типа *double*, а также различиями в реализации алгоритмов пользовательского и встроенного фильтров. Во встроенной функции нельзя задать положения якоря в ядре, кроме того, она использует разделение ядра для ускорения работы алгоритма.

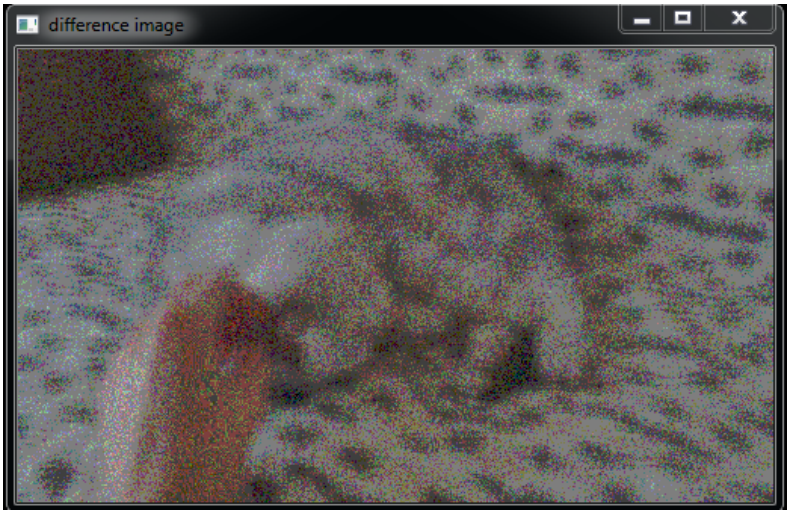


Рисунок 5 — Разница между результатами применения фильтров