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: Метод Оцу

Студент гр. 3331506/70401 Козлов Д. А.
Преподаватель Варлашин В. В.
« »_____2020 г.

Санкт-Петербург
2020

Задание

Пользуясь средствами языка C++ и библиотеки OpenCV, реализовать метод Оцу – алгоритм вычисления порога бинаризации для полутонового изображения.

Ход работы

Описание алгоритма

Метод Оцу ищет порог, уменьшающий дисперсию внутри класса, которая определяется как взвешенная сумма дисперсий двух классов:

$$\sigma_\omega^2(t) = \omega_1(t)\sigma_1^2(t) + \omega_2(t)\sigma_2^2(t),$$

где веса ω_i – это вероятности двух классов, разделенных порогом t, σ_i^2 – дисперсия этих классов.

Оцу показал, что минимизация дисперсии внутри класса равносильна максимизации дисперсии между классами:

$$\sigma_b^2(t) = \sigma^2 - \sigma_\omega^2(t) = \omega_1(t)\omega_2(t)[\mu_1(t) - \mu_2(t)]^2,$$

которая выражается в терминах вероятности ω_i и среднего арифметического класса μ_i, которое, в свою очередь, может обновляться итеративно. Эта идея привела к эффективному алгоритму.

Алгоритм:

Пусть дано монохромное изображение $G(i,j), = \overline{1, Height}, j = \overline{1, Width}$.

1. Вычислить гистограмму $p(l)$ изображения и частоту $N(l)$ для каждого уровня интенсивности изображения G.
2. Вычислить начальные значения для $\omega_1(0),\ \omega_2(0)$ и $\mu_1(0),\ \mu_2(0)$.
3. Для каждого значения $t = \overline{1, \max(G)}$ – полутона – вертикальной оси гистограммы:
    1. Обновляем $\omega_1,\ \omega_2$ и $\mu_1,\ \mu_2$
    2. Вычисляем $\sigma_b^2(t) = \omega_1(t)\omega_2(t)[\mu_1(t) - \mu_2(t)]^2$
    3. Если $\sigma_b^2(t)$ больше, чем имеющееся, то запоминаем $\sigma_b^2(t)$ то запоминаем σ_b^2 и значение порога t.

4. Искомый порог соответствует максимуму $\sigma_b^2(t)$.

Значения $\omega_1(t)$, $\omega_2(t)$ и $\mu_1(t)$, $\mu_2(t)$ вычисляются следующим образом:

$$n = \sum_{i=0}^{\max(G)} p(i), \qquad m = \sum_{i=0}^{\max(G)} i \cdot p(i)$$

$$\omega_1(t) = \frac{\sum_{i=0}^{t-1} p(i)}{n}, \qquad \omega_2(t) = \frac{\sum_{i=t}^{\max(G)} p(i)}{n} = 1 - \omega_1(t),$$

$$\mu_1(t) = \frac{\sum_{i=0}^{t-1} i \cdot p(i)}{n \cdot \omega_1(t)} = \frac{\sum_{i=0}^{t-1} i \cdot p(i)}{\sum_{i=0}^{t-1} p(i)} = \frac{\alpha_1}{\beta_1},$$

$$\mu_2(t) = \frac{\sum_{i=t}^{\max(G)} i \cdot p(i)}{n \cdot \omega_2(t)} = \frac{\sum_{i=t}^{\max(G)} i \cdot p(i)}{\sum_{i=t}^{\max(G)} p(i)} = \frac{m - \alpha_1}{n - \beta_1}.$$

Реализация алгоритма

Для описания метода Оцу был создан класс *Otsu*, изображенный на рисунке 1.

```
class Otsu
{
public:
    Otsu();
    ~Otsu();
    int setImage(Mat img);
    Mat getImage();
    int processImage();
private:
    Mat m_image;
    int* m_hist;
    int m_minLuminity;
    int m_maxLuminity;
    void makeHist();
    int otsuThreshold(Mat img);
    void binary(int treshold);
};
```

Рисунок 1 – Класс *Otsu*

Публичный метод *processImage()*, код которого представлен на рисунке 2, осуществляет пороговую бинаризацию изображения m_image с порогом, вычисленным методом Оцу.

```
int Otsu::processImage()
{
    if (m_image.empty())
    {
        return -1;
    }
    makeHist();
    binary(otsuThreshold(m_image));
    return 0;
}
```

Рисунок 2 – Метод *processImage()*

Функции, используемые в методе *processImage()*, рассматриваются ниже.

Для вычисления гистограммы изображения реализован метод *makeHist()*, код которого представлен на рисунке 3.

```
void Otsu::makeHist()
{
    m_minLuminity = m_image.at<uchar>(0, 0);
    m_maxLuminity = m_image.at<uchar>(0, 0);
    for (int i = 0; i < m_image.rows; i++)
    {
        for (int j = 0; j < m_image.cols; j++)
        {
            int luminity = m_image.at<uchar>(i, j);
            if (luminity < m_minLuminity)
            {
                m_minLuminity = luminity;
            }
            if (luminity > m_maxLuminity)
            {
                m_maxLuminity = luminity;
            }
        }
    }
    int histSize = m_maxLuminity - m_minLuminity + 1;
    m_hist = new int[histSize];
    for (int t = 0; t < histSize; t++)
    {
        m_hist[t] = 0;
    }
    for (int i = 0; i < m_image.rows; i++)
    {
        for (int j = 0; j < m_image.cols; j++)
        {
            m_hist[m_image.at<uchar>(i, j) - m_minLuminity]++;
        }
    }
}
```

Рисунок 3 – Метод вычисления гистограммы изображения

Для поиска порога бинаризации была создана функция *OtsuThreshold(Mat img)*, код которой изображен на рисунке 4.

```
int Otsu::otsuThreshold(Mat img)
{
    int m = 0; // m - сумма высот всех бинов, домноженных на положение их середины
    int n = 0; // n - сумма высот всех бинов
    for (int t = 0; t <= m_maxLuminity - m_minLuminity; t++)
    {
        m += t * m_hist[t];
        n += m_hist[t];
    }
    float maxSigma = -1; // Максимальное значение межклассовой дисперсии
    int threshold = 0; // Порог, соответствующий maxSigma
    int alpha1 = 0; // Сумма высот всех бинов для класса 1
    int beta1 = 0; // Сумма высот всех бинов для класса 1, домноженных на положение их середины
    for (int t = 0; t < m_maxLuminity - m_minLuminity; t++)
    {
        alpha1 += t * m_hist[t];
        beta1 += m_hist[t];
        float w1 = (float)beta1 / n;
        // a = a1 - a2, где a1, a2 - средние арифметические для классов 1 и 2
        float a = (float)alpha1 / beta1 - (float)(m - alpha1) / (n - beta1);
        float sigma = w1 * (1 - w1) * a * a;
        // Если sigma больше текущей максимальной, то обновляем maxSigma и порог
        if (sigma > maxSigma)
        {
            maxSigma = sigma;
            threshold = t;
        }
    }
    threshold += m_minLuminity;
    return threshold;
}
```

Рисунок 4 – Метод вычисления порога бинаризации

Для бинаризации изображения с использованием порогового значения, вычисленного ранее, реализован метод *binary(int threshold)*, код которого изображен на рисунке 5.

```
void Otsu::binary(int threshold)
{
    for (int i = 0; i < m_image.rows; i++)
    {
        for (int j = 0; j < m_image.cols; j++)
        {
            if (m_image.at<uchar>(i, j) <= threshold)
            {
                m_image.at<uchar>(i, j) = 0;
            }
            else
            {
                m_image.at<uchar>(i, j) = 255;
            }
        }
    }
}
```

Рисунок 5 – Метод пороговой бинаризации изображения

Сравнение с методом из OpenCV

Для сравнения использовалось монохромное изображение разрешением 1920 на 1080 пикселей (см. рисунок 6).

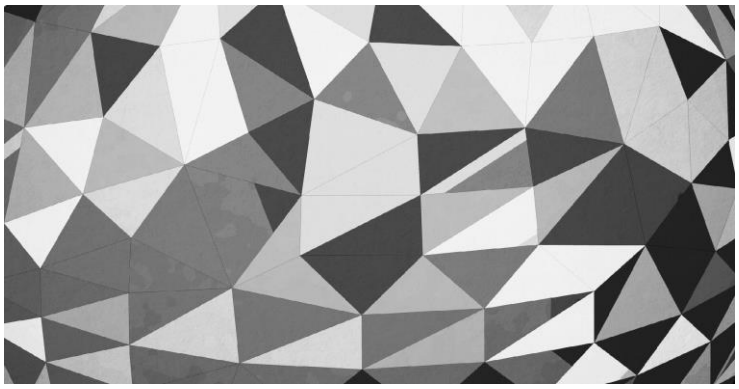

Рисунок 6 – Оригинальное изображение

Изображения, обработанные реализованным фильтром и фильтром из OpenCV, изображены на рисунке 7.

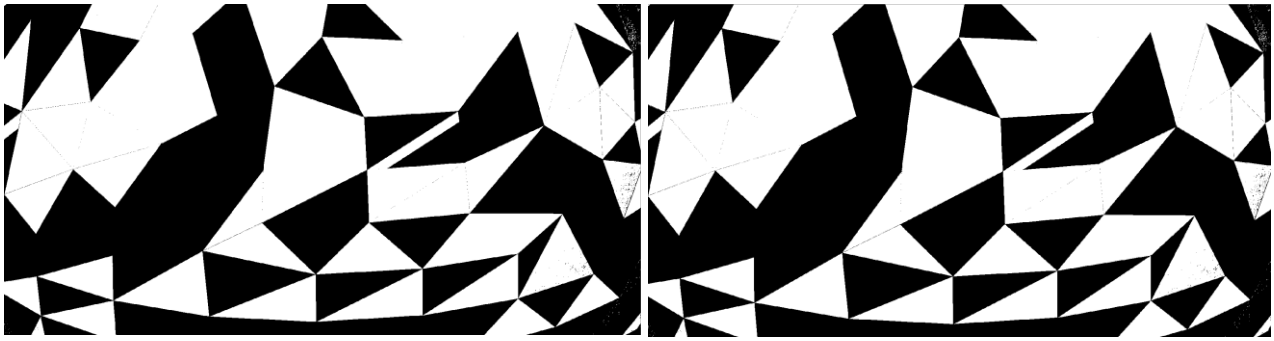

Рисунок 7 – Сравнение результатов

Для точного сравнения изображений реализована функция, код которой представлен на рисунке 8.

```
int compare(Mat img1, Mat img2)
{
    if (img1.total() != img2.total())
    {
        return -1;
    }
    int numOfDiffPix = 0;
    for (int i = 0; i < img1.rows; i++)
    {
        for (int j = 0; j < img2.cols; j++)
        {
            if (img1.at<uchar>(i, j) != img2.at<uchar>(i, j))
            {
                numOfDiffPix++;
            }
        }
    }
    return numOfDiffPix;
}
```

Рисунок 8 – Функция сравнения двух изображений

При сравнении результатов обработки данная функция вернула значение 0, т. е. результаты обработки полностью совпадают.

Для сравнения времени выполнения обработки изображения использовалась функция *clock()* библиотеки time.h, которая возвращает количество тактов процессора, прошедших с момента запуска программы. Поделив это количество тактов на количество тактов в секунду (CLOCKS_PER_SEC), можно получить время в секундах с момента запуска программы.

Таким образом, результаты определения времени бинаризации изображения разрешением 1920x1080 реализованным методом и методом из OpenCV представлены в таблице 1.

Таблица 1 – Сравнение времени выполнения

Конфигурация решения	*Otsu*	OpenCV
Debug	873 мс	12 мс
Release	17 мс	12 мс

Время выполнения обработки реализованным методом в конфигурации Release меньше, чем в Debug в ≈50 раз. Это связано с тем, что в конфигурации Release включена максимальная оптимизация (приоритет скорости) (/O2), а в Debug оптимизация отключена (/Od).

Вывод

В результате выполнения лабораторной работы был реализован метод бинаризации монохромного изображения порогом, вычисленным по методу Оцу. Результаты бинаризации реализованным методом и методом из OpenCV полностью совпали, однако метод из OpenCV более оптимизирован по времени выполнения обработки.