# Обработка изображений в OpenCV

Шокуров Антон В.
shokurov.anton.v@yandex.ru
http://машинноезрение.рф

17 октября 2016 г.

Версия: 0.09

**Аннотация**

Растровое изображение. Пиксели. Сглаживание.

## 1 Обработка изображений

В данной заметке будет показано как обрабатывать растровые изображения. В первом подразделе как по-пиксельно обрабатывать изображение.

### 1.1 Растровое изображение

Как было указано в прошлой заметке растровое изображение представляется в виде матрицы. Матрица хранится в памяти построчно, т.е. строчка за строчкой. Последние означает, что данные одной строчки идут вслед за данными предыдущей. Данные каждой строчки соответствуют пикселям её составляющие, при этом каналы хранятся последовательно для каждого из пикселей.

В согласии с вышеописанным представим простейшую программу, которая создает негатив по изображению, т.е. значение каждого из каналов заменяется на "противоположный а именно – вычитается из значения 255. Сначала покажем как это можно осуществить на чистом Си, далее будем видоизменять программу, в том числе, совершенствовать.

**Си-шный способ** Обход всех пикселов в общем случае технически сложен так как необходимо учитывать тип элементов матрицы. Рассмотрим случай, когда он является беззнаковым 8-битным целым числом (CV_8U).

```
1  //I -- входное изображение.
2  //В случае успеха возвращаем 0, иначе -1.
3  int makeNegative(Mat& I)
4  {
5    //Сначала проверим, что у изображения нужный тип.
6    if ( I.depth() != CV_8U)
7    {
8      //Если не подходит тип данных, то выходим
9      //из функции с ошибкой.
10     printf("Тип не поддерживается!\n");
11     return -1;
12   }
13
14   const int channels = I.channels();//Количество каналов.
15   const int rows = I.rows;//Количество строк.
16   const int cols = I.cols;//Количество столбцов.
17
18   int row;//Текущий номер строки.
19   for( row = 0; row < rows; row++)//Обходим все строки.
20   {
21     //Указатель на начало i строки матрицы.
22     //нужного типа
23     p = I.ptr<uchar>( row );
24     int col;//Текущий номер столбца.
25     //Обходим все столбцы:
26     for( col = 0; col < cols; col++)
27     {
28       int channel;//Текущий номер канала.
29       //Обходим все каналы:
30       for( channel = 0; channel < channels; channel++)
31         //Значение каждого канала каждого пикселя
32         p[col*channels + channel] = 255 - //изменяем на
33           p[col*channels + channel];//противоположный.
34     }
35   }
36   //В случае успеха возвращаем 0.
37   return 0;
38 }
```

С точки зрения эффективности память лучше обходить последовательно (связано с устройством кэша), поэтому строчки матрицы как это и показано в программе лучше обходить построчно.

Ввиду того, что каналы в памяти идут подряд, как и сами пикселы, в случае идентичной операции для всех каналов можно избавится от внутреннего цикла:

```
1     int col;//Текущий номер столбца.
2     //Обходим все столбцы и каналы:
3     for( col = 0; col < cols * channels; col++)
4     {
5         //Последовательно обрабатываем каждый из каналов:
6         p[col] = 255 - p[col];
7     }
```

Подчеркну, что это возможно только в том случае, елси все каналы обрабатываются единообразно. В противном случае, пришлось бы как ранее и было показано все-таки каждый из пикселов обрабатывать по отдельности.

Эффективность можно поднять ещё больше, если учитывать расположение строк. Так, существует метод `Mat::isContinuous()`, который позволяет определить идут ли строчку в притык одна к другой. Если да, то матрицу можно обрабатывать как одномерный массив:

```
1   if( I.isContinuous() )
2   {
3     cols *= rows;
4     rows = 1;
5   }
```

**Итератор из C++** Для обхода всех пикселов можно воспользоваться итератором.

```
1  int makeNegative(Mat& I)
2  {
3    if ( I.depth() != CV_8U)
4    {
5      printf("Тип не поддерживается!\n");
6      return -1;
7    }
8    const int channels = I.channels();
9    switch( channels )
10   {
```

```
    case 1:
    {
      MatIterator_<uchar> it, end;
      for( it = I.begin<uchar>(), end = I.end<uchar>();
           it != end; ++it)
        *it = 255 - *it;
      break;
    }
    case 3:
    {
      MatIterator_<Vec3b> it, end;
      for( it = I.begin<Vec3b>(), end = I.end<Vec3b>();
           it != end; ++it)
        {
          (*it)[0] = 255 - (*it)[0];
          (*it)[1] = 255 - (*it)[1];
          (*it)[2] = 255 - (*it)[2];
        }
      break;
    }
    default:
      printf("error in number of channels %d\n",
          channels);
  }
  return 0;
}
```

**Напрямую** У класса `Mat` есть метод `Mat::at`, который позволяет получить доступ к произвольному элементу матрицы.

```
I.at<uchar>(row, col) = 1 - I.at<uchar>(row, col);
```

или в случае 3х компонентной матрицы

```
I.at<uchar>(row, col)[0] = 255 -I.at<uchar>(row, col)[0];
I.at<uchar>(row, col)[1] = 255 -I.at<uchar>(row, col)[1];
I.at<uchar>(row, col)[2] = 255 -I.at<uchar>(row, col)[2];
```

**Современный C++**

**foreach**

**Упражнения** Упражнение. В прошлой заметке было показано как изменить контраст (умножению на константу) и яркость (добавление серого) фактически благодаря операциям над матрицами. Все тоже самое можно сделать написав цикл. Напишите его в качестве тренировки. Подсказка: значения нужно "насыщать"(`saturate_cast<uchar>`).

## 1.2 Изменение цветов

## 1.3 Сглаживание

## 1.4 Считывание видео

Как ранее было показано изображения считываются функцией `imread`. Для считывания видео нужно использовать класс `VideoCapture`.

```
1 //Код
```

В конструкторе указывается путь к картинке.