Appendix II. Image processing using the OpenCV library

1. Introduction

OpenCV (Open Source Computer Vision Library: http://opencv.org) is an open-source BSD-licensed library that includes several hundreds of computer vision algorithms of image processing at pixel level to complex methods such as camera calibration, object detection, optical flow, stereovision etc. It also includes tools for data storage and manipulation and graphical user interface functions. The OpenCV library package is freely available and can be downloaded (http://opencv.org/). The online documentation for the various editions of the library is available here: http://docs.opencv.org/ [1].

A framework based on the OpenCV library, called *OpenCVApplication* is available (you can download it from the personal web pages of the teaching staff). The framework is personalized for several versions of the Visual Studio (C++) development environments and several editions of the OpenCV library. The application includes some basic examples for opening and processing images and video streams. It can be used independently (without installing the OpenCV kit).

In the following, some basic data structures and image processing functions of the library are presented, as described in the 2.4.x API documentation (2.x API is essentially a C++ API).

In the OpenCV API, all the classes and functions are placed into the `cv` namespace. To access them from your code, use the `cv::` specifier or `using namespace cv;` directive:

```
...
cv::Mat src_gray, dst;
cv::equalizeHist(src_gray, dst);
...
```

or

```
using namespace cv;
...
Mat src_gray, dst;
equalizeHist(src_gray, dst);
...
```

Some of the current or future OpenCV external names may conflict with STL or other libraries. In this case, use explicit namespace specifiers to resolve the name conflicts:

2. Basic data structures in OpenCV

The Point_ class

`Point_` is a template class that specifies a 2D point by coordinates x and y. You can perform most of the unary and binary operations between `Point` type operators:

```
pt1 = pt2 + pt3;
pt1 = pt2 - pt3;
pt1 = pt2 * a;
pt1 = a * pt2;
pt1 += pt2;
pt1 -= pt2;
```

```
pt1 *= a;
double value = norm(pt); // L2 norm
pt1 == pt2;
pt1 != pt2;
```

You can use specific types for the coordinates and there is a cast operator to convert point coordinates to the specified type. The following type aliases are defined:

```
typedef Point_<int> Point2i;
typedef Point2i Point;
typedef Point_<float> Point2f;
typedef Point_<double> Point2d;
```

The Point3_ class

`Point3_` is a template class that specifies a 3D point by coordinates x, y and z. It supports all the vector arithmetic and comparison operations (same as for the `Point_` class). You can use specific types for the coordinates and there is a cast operator to convert point coordinates to the specified type. The following type aliases are defined:

```
typedef Point3_<int> Point3i;
typedef Point3_<float> Point3f;
typedef Point3_<double> Point3d;
```

The Size_ class

`Size_` is a template class that specifies the size of an image or rectangle. The class includes two members called `width` and `height`. The same arithmetic and comparison operations as for `Point_` class are available.

The Rect_ class

`Rect_` is the template class for 2D rectangles, described by the following parameters:

- Coordinates of the top-left corner: `Rect_::x` and `Rect_::y`
- Rectangle width (`Rect_::width`) and height (`Rect_::height`).

The following type allias is defined:

```
typedef Rect_<int> Rect;
```

The following operations on rectangles are implemented:

- shift: `rect = rect ± point`
- expand/shrink: `rect = rect ± size`
- augmenting operations: `rect += pont, rect -= pont, rect += size, rect -= size`
- intersection: `rect = rect1 & rect2, rect &= rect1`
- minimum area rectangle containing 2 rectangles: `rect = rect1 | rect2, rect |= rect1`
- rectangle comparission: `rect == rect1, rect != rect1`

The Vec class

The `Vec` class is commonly used to describe pixel types of multi-channel arrays. For example to describe a RGB 24 image pixel the following type can be used:

```
typedef Vec<uchar, 3> Vec3b;
```

The following vector operations are implemented:

`v1 = v2 + v3`
`v1 = v2 - v3`
`v1 = v2 * scale`
`v1 = scale * v2`
`v1 = -v2`
`v1 += v2` and other augmenting operations
`v1 == v2, v1 != v2`
`norm(v1)` (Euclidean norm)

The Scalar_ class

`Scalar_` is a template class for a 4-element vector derived from Vec.

The Mat class

The **`Mat`** class represents an n-dimensional single-channel or multi-channel array. It can be used to store real or complex vectors and matrices, grayscale or color images, histograms etc. 2-dimensional matrices are stored row-by-row, 3-dimensional matrices are stored plane-by-plane, and so on.

```
class CV_EXPORTS Mat
{
public:
    // ... a lot of methods ...
    ...
    //! the array dimensionality, >= 2
    int dims;
    //! the number of rows and columns or
    // (-1, -1) when the array has more than 2 dimensions
    int rows, cols;
    //! pointer to the data
    uchar* data;
    // other members
    ...
};
```

The most common ways to create a Mat object are
`create(nrows, ncols, type)` or
using one of the constructors: `Mat(nrows, ncols, type[, fillValue])`

Examples:

```
Mat m1,m2;
// create a 100x100 3 channel byte matrix
m1.create(100,100,CV_8UC(3));
// create a 5x5 complex matrix filled with (1-2j)
Mat m2(5,5,CV_32FC2,Scalar(1,-2));
```

```
// create a 640x480 1 channel byte matrix filled with 0
Mat src(480,640,CV_8UC1, 0);
```

Accessing the elements of a matrix can be done in several ways. Supposing that `src` is a gray-scale image with 480 rows and 640 columns (initialized by opening an image from the disk), the examples below are presenting how a simple processing like the image negative can be implemented:

```
int height = src.rows;
int width = src.cols;
Mat dst = src.(clone);

// the "easy/slow" approach
for (int i=0; i<height; i++)
{
	for (int j=0; j<width; j++)
	{
		uchar val = src.at<uchar>(i,j);
		uchar neg = 255-val;
		dst.at<uchar>(i,j) = neg;
	}
}
```

or

```
// the fast approach
for (int i = 0; i < height; i++)
{
	// get the pointer to row i
	const uchar* SrcRowi = src.ptr<uchar>(i);
	uchar* DstRowi = dst.ptr<uchar>(i);
	//iterate through each row
	for (int j = 0; j < width; j++)
	{
		uchar val = SrcRowi[j];
		uchar neg = 255 - val;
		DstRowi[j] = neg;
	}
}
```

or

```
// the fastest approach using the "diblook style"
uchar *lpSrc = src.data;
uchar *lpDst = dst.data;
int w = src.step; // no DWORD alignment is done !!!
for (int i = 0; i<height; i++)
	for (int j = 0; j < width; j++)
	{
		uchar val = lpSrc[i*w + j];
		lpDst[i*w + j] = 255 - val;
	}
```

The operations implemented on matrices that can be combined in arbitrary complex expressions. Few examples are presented bellow (see the documentation for more examples). In the expressions bellow `A`, `B` stand for matrices (`Mat`), `s` for a scalar (`Scalar`), `alpha` for a scalar (`double`)).

- Addition, subtraction, negation: `A+B, A-B, A+s, A-s, s+A, s-A, -A`
- Scaling: `A*alpha`

- Matrix multiplication: `A*B`
- Transposition: `A.t()` (means A^T)
- Matrix inversion and pseudo-inversion, solving linear systems and least-squares problems: `A.inv([method])` (~ A^{-1}) , `A.inv([method])*B` (~ `X: AX=B`)
- Comparison: `A cmpop B, A cmpop alpha, alpha cmpop A`, where `cmpop` is one of: `>, >=, ==, !=, <=, <`. The result of comparison is an 8-bit single channel mask whose elements are set to 255 (if the particular element or pair of elements satisfy the condition) or 0.
- Bitwise logical operations: `A logicop B, A logicop s, s logicop A, ~A`, where `logicop` is one of : `&, |, ^`.
- Element-wise minimum and maximum: `min(A, B), min(A, alpha), max(A, B), max(A, alpha)`
- Element-wise absolute value: `abs(A)`
- Cross-product, dot-product: `A.cross(B)` `A.dot(B)`
- Matrix initializers: `Mat::eye(), Mat::zeros(), Mat::ones()`

In order to get a region of interest (ROI) from a matrix defined by a Rect Structure use the following statements:

```
Mat image;
Rect ROI_rect;
Mat roi=image(ROI_rect);
```

3. Reading, writing and displaying images and videos

To open an image file stored on the disk the `imread` function can be used. It can handle/decode the most common image formats (bmp, jpg, gif, png etc.):

```
Mat imread(const string& filename, int flags=1 )
```

The input parameters are the `filename` and an optional `flags` parameter specifying the color type of a loaded image:

- CV_LOAD_IMAGE_ANYDEPTH - returns a 16-bit/32-bit image when the input has the corresponding depth, otherwise converts it to 8-bit.
- CV_LOAD_IMAGE_COLOR - always convert the image to a color one.
- CV_LOAD_IMAGE_GRAYSCALE - always converts the image to a grayscale one
- **>0** - returns a 3-channel color image.
- **=0** - returns a grayscale image.
- **<0** - returns the loaded image as is (with alpha channel).

The output is a `Mat` object containing the image for successful completion of the operation. Otherwise the function returns an empty matrix (`Mat::data==NULL`).

To display an image in a specified window, the `imshow` function can be used:

```
void imshow(const string& winname, InputArray mat)
```

In order to control the size and position of the display window the following functions can be used:

```
void namedWindow(const string& winname, int flags=WINDOW_AUTOSIZE )
void moveWindow(const string& winname, int x, int y)
```

To save the image on to the disk the `imwrite` function can be used. The image format is chosen based on the `filename` extension.

```
bool imwrite(const string& filename, InputArray img,
                            const vector<int>& params=vector<int>())
```

The input parameter `params` contains format-specific save parameters encoded as pairs `paramId_1, paramValue_1, paramId_2, paramValue_2, ...` . The following parameters are currently supported:

- For JPEG, it can be a quality (`CV_IMWRITE_JPEG_QUALITY`) from 0 to 100 (the higher is the better). Default value is 95.
- For PNG, it can be the compression level (`CV_IMWRITE_PNG_COMPRESSION`) from 0 to 9. A higher value means a smaller size and longer compression time. Default value is 3.
- For PPM, PGM, or PBM, it can be a binary format flag (`CV_IMWRITE_PXM_BINARY`), 0 or 1. Default value is 1.

The following example illustrates the above mentioned image handling functions by opening a color image, converting it to grayscale and saving it to the disk and displaying the source image and the destination/result image in a separate windows:

```
void testImageOpenAndSave()
{
	Mat src, dst;
	src = imread("Images/Lena_24bits.bmp", CV_LOAD_IMAGE_COLOR); //Read the image

	if (!src.data) //Check for invalid input
	{
		printf("Could not open or find the image\n");
		return;
	}

	//Get the image resolution
	Size src_size = Size(src.cols, src.rows);

	//Display window
	const char* WIN_SRC = "Src"; //window for the source image
	namedWindow(WIN_SRC, CV_WINDOW_AUTOSIZE);
	cvMoveWindow(WIN_SRC, 0, 0);
```

```
	const char* WIN_DST = "Dst"; //window for the destination (processed) image
	namedWindow(WIN_DST, CV_WINDOW_AUTOSIZE);
	cvMoveWindow(WIN_DST, src_size.width + 10, 0);

	cvtColor(src, dst, CV_BGR2GRAY); //converts the source image to grayscale
	imwrite("Images/Lena_24bits_gray.bmp", dst); //writes the destination to file

	imshow(WIN_SRC, src);
	imshow(WIN_DST, dst);

	printf("Press any key to continue ...\n");
	waitKey(0);
}
```

The `VideoCapture` class provides the C++ API for capturing video from cameras or for reading video files. In the following example is shown how you can use it (i.e. performing canny edge detection on every frame and displaying the result in a destination window; the example also shows how you can compute the processing time).

```
void testVideoSequence()
{
	VideoCapture cap("Videos/rubic.avi"); // open a video file from disk
	//VideoCapture cap(0); // open the default camera (i.e. the built in web cam)
	if (!cap.isOpened()) // opening the video device failed
	{
		printf("Cannot open video capture device\n");
		return;
	}

	Mat frame, grayFrame, dst;

	// video resolution
	Size capS = Size((int)cap.get(CV_CAP_PROP_FRAME_WIDTH),
		(int)cap.get(CV_CAP_PROP_FRAME_HEIGHT));

	// Init. display windows
	const char* WIN_SRC = "Src"; //window for the source frame
	namedWindow(WIN_SRC, CV_WINDOW_AUTOSIZE);
	cvMoveWindow(WIN_SRC, 0, 0);

	const char* WIN_DST = "Dst"; //window for the destination (processed) frame
	namedWindow(WIN_DST, CV_WINDOW_AUTOSIZE);
	cvMoveWindow(WIN_DST, capS.width + 10, 0);

	char c;
	int frameNum = -1;

	for (;;)
	{
		cap >> frame; // get a new frame from camera
		if (frame.empty())
		{
			printf("End of the video file\n");
			break;
		}

		++frameNum;

		double t = (double)getTickCount(); // Get the current time [s]

		// Insert your processing here ....
		cvtColor(frame, grayFrame, CV_BGR2GRAY);
		// Performs canny edge detection on the current frame
		Canny(grayFrame, dst, 40, 100, 3);
		// ....
		// End of processing
```

```
            // Get the current time again and compute the time difference [s]
            t = ((double)getTickCount() - t) / getTickFrequency();

            // Print (in the console window) the processing time in [ms]
            printf("Time = %.3f [ms]\n", t * 1000);

            // output written in the WIN_SRC window (upper left corner)
            char msg[100];
            sprintf(msg, "%.2f[ms]", t * 1000);
            putText(frame, msg, Point(5, 20), FONT_HERSHEY_SIMPLEX,
                                        0.5, CV_RGB(255, 0, 0), 1, 8);
            imshow(WIN_SRC, frame);
            imshow(WIN_DST, dst);

            c = cvWaitKey(0);  // waits a key press to advance to the next frame
            if (c == 27) {
                  // press ESC to exit
                  printf("ESC pressed - capture finished");
                  break;  //ESC pressed
            }
      }
}
```

4. Basic operations applied on images

The operations are generic for array type objects (i.e. `Mat`) and if the array is initialized with an image, the result is a pixel level operation. In the case of multi-channel arrays, each channel is processed independently. Some functions allow the specification of an optional mask used to select a sub-array.

- `void` **`add`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`, InputArray` **`mask`**`=noArray(), int` **`dtype`**`=-1)` **-** Calculates the per-element sum of two arrays or an array and a scalar: *src1* is added to *src2* and the result is stored in dst. The function can be replaced with matrix expressions: `dst = src1 + src2;`

- `void` **`addWeighted`**`(InputArray` **`src1`**`, double` **`alpha`**`, InputArray` **`src2`**`, double` **`beta`**`, double` **`gamma`**`, OutputArray` **`dst`**`, int` **`dtype`**`=-1)` – performs the weighted sum of two arrays, and is equivalent with the following matrix expression: `dst = src1*alpha + src2*beta + gamma;`

- `void` **`absdiff`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`)` **-** performs the per-element absolute difference between two arrays or between an array and a scalar.

- `void` **`bitwise_and`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`, InputArray` **`mask`**`=noArray()` – computes the per-element bit-wise conjunction of two arrays (*src1* and *src2*) or an array and a scalar.

- `oid` **`divide`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`, double` **`scale`**`=1, int` **`dtype`**`=-1)` – performs the division operation between the elements of two arrays; the result is multiplied with the scaling parameter.

- `Scalar` **`mean`**`(InputArray` **`src`**`, InputArray` **`mask`**`=noArray())` – calculates the mean value of the array elements, independently for each channel of the array.

- `void` **`max`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`))` – computes the per-element maximum of two arrays: *src1* and *src2*.

`void` **`min`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`)` – computes the per-element minimum of two arrays: *src1* and *src2*.

- `void` **`minMaxLoc`**`(InputArray` **`src`**`, double*` **`minVal`**`, double*` **`maxVal`**`=0, Point*` **`minLoc`**`=0, Point*` **`maxLoc`**`=0, InputArray` **`mask`**`=noArray())` – returns the minimum and maximum value in the *src* array and also their coordinates (the function does not work with multi-channel arrays.).

- `void` **`multiply`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`, double` **`scale`**`=1, int` **`dtype`**`=-1 )` **-** performs the per-element scaled product of two arrays.

5. Morphological operations

In OpenCV the morphological operations work on both binary and grayscale images. Each morphological operation requires a structuring element. This can be created using `getStructuringElement` function:

```
Mat getStructuringElement(int shape, Size ksize, Point anchor=Point(-1,-1)
```

his function creates a structuring element of given shape and dimension. The *shape* parameter controls its shape and can take the following constant values:

- `CV_SHAPE_RECT`
- `CV_SHAPE_CROSS`
- `CV_SHAPE_ELLIPSE`

The *anchor* parameter specifies the anchor position within the element. The default value (-1, -1) means that the anchor is at the center. Note that only the shape of a cross-shaped element depends on the anchor position. In other cases the anchor just regulates how much the result of the morphological operation is shifted.

Morphological dilation can be performed using the *dilate* function:

```
void dilate (InputArray src, OutputArray dst, InputArray kernel, Point
anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const
Scalar& borderValue=morphologyDefaultBorderValue() )
```

Morphological erosion can be performed using the *erode* function:

```
void erode(InputArray src, OutputArray dst, InputArray kernel, Point
anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const
Scalar& borderValue=morphologyDefaultBorderValue() )
```

An example that performs a 2 iterations erosion followed by a 2 iterations dilation using a 3x3 cross-shape structuring element is presented below:

```
//structuring element for morpho operations
Mat element = getStructuringElement(MORPH_CROSS, Size(3, 3));
erode(src, temp, element, Point(-1, -1), 2);
dilate(temp, dst, element, Point(-1, -1), 2);
```

unction `morphologyEx` can be used to perform more complex morphological operations:

```
void morphologyEx(InputArray src, OutputArray dst, int op, InputArray kernel, Point anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const Scalar& borderValue=morphologyDefaultBorderValue() )
```

The `op` parameter specifies the type of the operation which is performed:

- **MORPH_OPEN** - an opening operation
- **MORPH_CLOSE** - a closing operation
- **MORPH_GRADIENT** - a morphological gradient
- **MORPH_TOPHAT** - "top hat"
- **MORPH_BLACKHAT** - "black hat"

6. Thresholding

The thresholding operation in OpenCV are performed with *threshold* function:

```
double threshold(InputArray src, OutputArray dst, double thresh,
                 double maxval, int type)
```

The operation is applied on the *src* image and the resulted binary image is stored in *dst* array. The threshold value is specified by *thresh* parameter and the thresholding method is specified by *type* parameter. The *type* parameter can take one of the following constant values:

- `THRESH_BINARY` $dst(x,y) = \begin{cases} maxval, if\ src(x,y) > TH \\ 0, otherwise \end{cases}$
- `THRESH_BINARY_INV` $dst(x,y) = \begin{cases} 0, if\ src(x,y) > TH \\ maxval, otherwise \end{cases}$
- `THRESH_TRUNC` $dst(x,y) = \begin{cases} TH, if\ src(x,y) > TH \\ src(x,y), otherwise \end{cases}$
- `THRESH_TOZERO` $dst(x,y) = \begin{cases} src(x,y), if\ src(x,y) > TH \\ 0, otherwise \end{cases}$
- `THRESH_TOZERO_INV` $dst(x,y) = \begin{cases} 0, if\ src(x,y) > TH \\ src(x,y), otherwise \end{cases}$

The above values can be combined with `THRESH_OTSU`. In this case, the function computes the optimal threshold value using the Otsu's algorithm (the implementation works only for 8-bit

images) which is used instead of the specified *thresh* parameter. The function returns the computed optimal threshold value.

7. Filters

In OpenCV there are some optimized function designed to perform the filtering operations. These optimizations depend however on the hardware and software architecture of the system.

Some example of functions that perform filtering operations are as follows:

`void medianBlur(InputArray` **`src`**`, OutputArray` **`dst`**`, int` **`ksize`**`)` – implements the median filter

`void blur(InputArray` **`src`**`, OutputArray` **`dst`**`, Size` **`ksize`**`, Point` **`anchor`**`=Point(-1,-1), int` **`borderType`**`=BORDER_DEFAULT )` – implements the normalized box filter (mean filter).

`void GaussianBlur(InputArray` **`src`**`, OutputArray` **`dst`**`, Size` **`ksize`**`, double` **`sigmaX`**`, double` **`sigmaY`**`=0, int` **`borderType`**`=BORDER_DEFAULT )` - implements the gaussian filter.

`oid Laplacian(InputArray` **`src`**`, OutputArray` **`dst`**`, int` **`ddepth`**`, int` **`ksize`**`=1, double` **`scale`**`=1, double` **`delta`**`=0, int` **`borderType`**`=BORDER_DEFAULT ) - p`erforms the filtering with a Laplacian filter using the specified aperture size if `ksize >1`. If `ksize ==1` the 3x3 Laplacian filter have the following elements is applied:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

`void Sobel(InputArray` **`src`**`, OutputArray` **`dst`**`, int` **`ddepth`**`, int` **`dx`**`, int` **`dy`**`, int` **`ksize`**`=3, double` **`scale`**`=1, double` **`delta`**`=0, int` **`borderType`**`=BORDER_DEFAULT )`- computes the first, second, third, or mixed image derivatives (`dx` and `dy` parameters) using an extended Sobel operator. Most often, the function is called with (`xorder` = 1, `yorder` = 0, `ksize` = 3) or (`xorder` = 0, `yorder` = 1, `ksize` = 3) to calculate the first x- or y- image derivative F (see chapter 12 for more details).

`void filter2D(InputArray` **`src`**`, OutputArray` **`dst`**`, int` **`ddepth`**`, InputArray` **`kernel`**`, Point` **`anchor`**`=Point(-1,-1), double` **`delta`**`=0, int` **`borderType`**`=BORDER_DEFAULT )`- this is a generalized function designed to apply the filtering operation with a custom convolution kernel. The kernel elements are defined in *kernel* parameter as a matrix. Default value (-1,-1) for the *anchor* parameter means that the anchor is at the kernel center.

8. References

[1] OpenCV on-line documentation, http://docs.opencv.org/ , cited Dec. 2015.