Image manipulation

Introduction to OpenCV

Laurent Beaudoin & Loïca Avanthey
Epita

Foreword

During this seminar, we will become familiar with the manipulation of images: how to open them, access the pixels in read or write modes, etc. There are many solutions for manipulating images in many different languages. We will discover the most famous (and perhaps the most complete) library: the OpenCV library, in C / C ++ (also interfaced in python and Java).

1 First steps in OpenCV

1.1 Did you say OpenCV?

OpenCV is an **open-source library** dedicated to **artificial vision** (CV is for Computer Vision). It therefore incorporates **efficient image processing tools** but also **artificial intelligence tools**. It has become an **essential reference** in recent years, both for industry and for research.

Created in 1999 by the will of the **Intel company**, it has not stopped developing and is now supported by a large and particularly active community. It is under **BSD license**.

There are many **online resources** for taming the library. Among other things, we recommend:

- `opencv.org`: this is the reference site where you will find... everything from the library itself to tutorials.
- `docs.opencv.org`: you will find there the documentation of the version installed on your machine (to be consumed without moderation).

Otherwise, for those who want to go further, there are the **books**! We recommend in particular the reference work written by the father of the library, Gary Bradski [KB17] and the one written by Samarth Brahmbhatt [Bra13].

OpenCV is organized in more or less **independent modules**, which makes it possible to compile programs using only what is necessary. Here are the main modules:

- `core`: it is the minimal module which contains all the basic objects and the basic functions,
- `imgproc`: dedicated to basic image processing,
- `highgui`: dedicated to display (GUI),
- `video`: dedicated to reading and writing video,
- `calib3d`: dedicated to camera calibration,
- `features2d`: dedicated to the detection and monitoring of characteristics,
- `ml`: dedicated to machine learning.

For what follows, we will assume to be on a Linux distribution, but OpenCV also works on MacOs, Windows, Android and iOS.

EXERCICE 1 (*Installation*)

Install the latest stable version of OpenCV from the resource sites (preferably use the packages if they are available for your version: `sudo apt-get install libopencv-dev`, `sudo brew install opencv`, `sudo pacman -S opencv fmt hdf5 vtk glew qt5 qtcreator qt5-doc`, etc.).

1.2 First program

1.2.1 Displaying an image: the code

The following code displays an image:

```
#include <iostream>
#include <opencv2/opencv.hpp>

int main ( int argc, char** argv){
   cv::Mat img = cv::imread( argv[1], cv::IMREAD_COLOR);
   if( img.empty() ) {
      std::cerr << "File not found!!!" << std::endl;
      exit(-1);
   }
   cv::namedWindow("Image", cv::WINDOW_AUTOSIZE);
   cv::imshow("Image", img);
   cv::waitKey(0);
   cv::destroyWindow("Image");
   return(1);
}
```

EXERCICE 2 (*Hello World*)

Copy this code into a file `load.cpp`.

Warning, a **copy** from a pdf does not always reproduce **line breaks** properly.

1.2.2 Compilation

The easiest way to **compile** OpenCV programs is to use `CMake` (see appendix C page 24).

EXERCICE 3 (*CMakeLists*)

To do this, in your working directory, create the following `CMakeLists.txt` file:

```
cmake_minimum_required( VERSION 2.8 )
project( exemplesOpencv )
find_package( OpenCV REQUIRED )
include_directories( ${OpenCV_INCLUDE_DIRS} )
#-----------compilation step
add_executable( load.exe load.cpp )
#-----------linking step
target_link_libraries( load.exe ${OpenCV_LIBS} )
```

EXERCICE 4 (*Make*)

Then in the terminal, create the make file, compile and launch the executable:

```
cmake CMakeLists.txt
make
./load.exe ./img.jpg
```

For the next exercices, it will suffice to complete the `CMakeLists.txt` file to launch all the compilations automatically with a simple call to `make`.

1.3 HighGUI

OpenCV makes it easy to **interact** with the operating system to manage the **graphics layer** in conjunction with the user (GUI – Graphical User Interface). This layer is made up of **3 modules**:

- `imgcodecs`, dedicated to (de)compression of images,
- `videoio`, for video capture and encoding,
- `highgui` for the graphical interface.

This **modular** organization makes it possible to be optimal in size for **embedded** applications (where the graphical interface loses its interest).

Let's take the functions of our Hello World:

```
cv::Mat cv::imread(const string& fileName, int flags = cv::IMREAD_COLOR);
```

The `cv::imread()` function allows you to load the `fileName` image. This function begins by identifying the **type of image** by its signature in its header (and not by its extension) and uses the appropriate codec to read the file. It allows you to **load the image** in an instantiation of the `cv::Mat` class with a choice either of a color image (3 layers with `cv::IMREAD_COLOR`) or in gray levels (1 layer with `cv::IMREAD_GRAYSCALE`). Supported formats are: `BMP`, `DIB`, `JPEG`, `JPE`, `PNG`, `PBM`, `PGM`, `SR`, `RAS` and `TIFF`.

Order of stored radiometric values

The radiometric values are stored in memory for each pixel in the Blue Green Red (BGR) order and not RGB (to be faster on graphics card processing)!

```
bool cv::imwrite(const string& fileName, cv::InputArray image);
```

The function `cv::imwrite()` allows to save a matrix (an image) on the hard disk. The tolerated formats are the same as for the `cv::imread()` function. It is possible to specify optional compression parameters, for example for `JPEG`, `PNG` and `PXM` formats.

```
int cv::namedWindow(const string& name, int flags = cv::WINDOW_AUTOSIZE);
```

The function `cv::namedWindow` allows you to easily create a graphic window intended to display an image. The name `name` of the window, which also appears in the title bar, is used as its identifier. The window automatically adjusts its size to its container.

```
void cv::imshow(const string& name, cv::InputArray image);
```

The `cv::imshow` function allows you to load the image `image` into a graphic window identified by `name`.

```
int cv::waitKey(int delay);
```

The function `cv::waitKey` waits `delay` milliseconds for pressing a key on the keyboard. If `delay` is 0, the program waits indefinitely. If no key is pressed during the `delay` time, the function returns 255.

```
int cv::destroyWindow(const string& name);
void cv::destroyAllWindows(void);
```

The `cv::destroyWindow` function allows you to close the `name` window and deallocate the associated memory. The `cv::destroyAllWindows` function does this for all active windows in a single call.

1.4 Data structure for images: dense matrices

An **image** can be considered as a **dense matrix** (in the algebraic sense, that is to say that it contains few zeros, unlike sparse matrices). In OpenCV, the class that corresponds to these matrices is the class `cv::Mat`. It is the heart of the library: practically everything revolves around this class.

An object of type `cv::Mat` contains several attributes such as its dimensions, the type of data (type of primitive like `int` or `cv::Point2D` for example, the number of channels etc.), the pointer to the memory area where the data is physically stored, etc.

The `cv::Point` class (see table 2, appendix A, page 17) *" It allows you to manipulate **a 2D or 3D point**. The attributes x and y (and z if 3D) can be integers or reals. All possible combinations for this object follow the pattern `cv::Point{2,3}{i,f,d}` with i for integer, f for float and d for double. For example, `cv::Point2i` designates an integer 2D point, `cv::Point3f` a floating 3D point."*

Data managment

It is important to understand that the data of a `cv::Mat` object is **not rigidly attached** to its instantiation!

For example, if we **assign** one matrix to another (`m = n`), the data pointer of `m` will point to the same area as that of `n`, all the other attributes of `m` will be **refreshed** and the previously pointed data area will be **automatically deallocated** if it is not referenced by another pointer.

The **data type** of **all boxes** of the matrix is defined by the following type model : `CV_{8U, 16S, 16U, 32S, 32F, 64F}C{1,2,3}` where `8`, `16`, `32` or `64` is the number of bits, `U` for unsigned, `S` for signed, `F` for floating point, `1`, `2` or `3` for the number of channels. For example, `CV_32FC3` means floating type of dimension 3.

1.4.1 Creation of a matrix

To create a matrix, we can:

- either call the **constructor** `cv::Mat` then **create the data zone** with the methods `create()` and **initialize** it with `setTo`,

```
cv::Mat m;
m.create(3, 10, CV_32FC3);
// Set the values to 1 for the 1st channel, 2 for the 2nd and 3 for the 3rd
m.setTo(cv::Scalar(1.0f, 2.0f, 3.0f));
```

- either use the **direct** constructors (cf. table 9, appendix B, page 20),
- either use the constructors **by copy** (cf. table 10, appendix B, page 20),
- either use the **particular** constructors (cf. table 11, appendix B, page 20).

The `cv::Scalar` class (cf. table 4, appendix A, page 17) *" It makes it possible to store **quaternion** type information, that is to say a 4-dimensional vector whose components are in double precision."*

1.4.2 Data access (read / write)

There are **several methods** for **accessing individual values** of a matrix. The simplest is a call to the accessor `at<>()`. To use it, just put in the template the right type of data. The table 12 in the appendix B page 20 illustrates several cases of use of `at<>()`.

An example of use is better than a long speech, the following code:

```
#include <opencv2/opencv.hpp>

int main(int argc, char** argv){
   cv::Mat m(6,6, CV_8UC3);
   unsigned char step = 255/5;
   std::cout << (int)step << std::endl;
   for(int row=0; row<6; row++){
      for(int col=0;col<6; col++){
         m.at<cv::Vec3b>(row,col)[0] = col * step;
         m.at<cv::Vec3b>(row,col)[1] = col * step;
         m.at<cv::Vec3b>(row,col)[2] = col * step;
         std::cout << "(" << (int) m.at<cv::Vec3b>(row,col)[0] << ","
            << (int) m.at<cv::Vec3b>(row,col)[1] << ","
            << (int) m.at<cv::Vec3b>(row,col)[2] << ") ";
      }
      std::cout << std::endl;
   }
   return 0;
}
```

Displays on the screen:

```
(0,0,0) (51,51,51) (102,102,102) (153,153,153) (204,204,204) (255,255,255)
(0,0,0) (51,51,51) (102,102,102) (153,153,153) (204,204,204) (255,255,255)
(0,0,0) (51,51,51) (102,102,102) (153,153,153) (204,204,204) (255,255,255)
(0,0,0) (51,51,51) (102,102,102) (153,153,153) (204,204,204) (255,255,255)
(0,0,0) (51,51,51) (102,102,102) (153,153,153) (204,204,204) (255,255,255)
```

The `cv::Size` class (cf. table 5, appendix A, page 18) *" It allows you to define **the size** of an object. The attributes `width` and `height` can be integer or floats, which gives as possible combinations `cv::Size2{i,f}`."*

The `Cv::Vec` class (cf. table 3, appendix A, page 17) *" It allows you to handle **vectors of small dimensions**. All possible combinations follow the pattern `cv::Vec{2,3,4,6}{b, s, w, i, f, d}` where the numbers represent the number of dimensions, `b` for `unsigned char`, `w` for `unsigned short`, `s` for `short`, `i` for `int`, `f` for `float` and `d` for double. For example: `Vec2b` for a 2D vector of `unsigned char` and `Vec3f` for a 3D vector of floats."*

It is also possible to **retrieve data blocks** directly as a row, a column, the diagonal, etc. The methods available for this are summarized in the table 13 in the appendix B page 21.

1.4.3 Calculations

It is very simple to do **algebraic calculations** with matrices in OpenCV. The table 14 in the appendix B page 21 gives some examples. As a programmer, appreciate the **elegance** of these expressions ;-)! Similarly, there are **methods** which allow you to manipulate the `cv::Mat` object easily for operations like **copy**, **initialization**, **definition of an area of interest** (ROI for *Region Of Interest*) or the **recovery of attributes** (cf. table 15, appendix B, page 21).

Finally, there is a whole **toolbox** of functions allowing you to easily do **advanced operations** on matrices. The online documentation can be improved and especially useful when you already know at least the name of the function. To help you, you will find in the table 16 in the appendix B page 23 a list of functions of interest and a brief description of what they do.

1.5 Other common classes

The `cv::Rect` class (cf. table 6, appendix A, page 18) *“ It allows to define **a rectangle** aligned on the vertical and horizontal. Its attributes relate to the upper left corner of the rectangle and the dimensions (width and height) of it from this corner.”*

The `cv::RotatedRect` class (cf. table 7, appendix A, page 19) *“ It allows you to define **an oriented rectangle** according to any **angle**. Its attributes are the center of the rectangle, its size and its orientation.”*

The `cv::Complex` class (cf. table 8, appendix A, page 19) *“ It allows you to handle **a complex number**. Its attributes can be floats or doubles which gives as possible combinations `cv::Complex{f, d}`.”*

The `cv::InputArray` and `cv::OutputArray` classes *“ They are used to pass arrays as arguments and return values. These classes are justified to make prototypes of OpenCV functions easy to read and manipulate. They cover all possible types of tables. `cv::InputArray` is read-only (`const`). Sometimes prototypes have optional argument (or return) arrays. If you don't use this option, then replace the argument (or return) with `cv::noArray`.”*

2 Image creation and manipulation: it's your turn

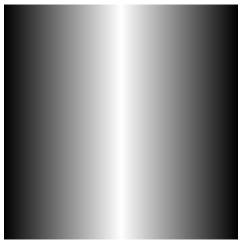
2.1 Creation and manipulation of matrices

EXERCICE 5 (*Longitudinal gradient*)

With OpenCV:

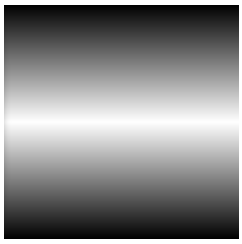
a) Create a matrix of 512×512 pixels of type `unsigned char` on 1 channel.

b) Fill the matrix so as to obtain the gradient illustrated by the following image and display the result.



EXERCICE 6 (*Latitudinal gradient*)

In the same code, create a second matrix to obtain the following gradient and display the result.



EXERCICE 7 (*Mix*)

Still in the same code do the half-sum of the previous matrices and display the result.

EXERCICE 8 (*On a picture*)

Again in the same code:

a) Load any color image and modify your previous code by adapting the width and height of the matrices to those of the loaded image.

b) Convert the matrix of half-sums to three channels thanks to `cv::cvtColor`. Using `cv::multiply` make the multiplication term by term between the result obtained and the image and divide it by 255 to obtain values between 0 and 1 (normalization). Display the result.

2.2 Conversion to grayscale of a color image

EXERCICE 9 (*Homemade conversion*)

With OpenCV:

a) Load and display any image in color which will be called the original image later.

b) Create a matrix of `unsigned char` of the same size as the original image but with a single channel and of which each pixel is the mean of the `BGR` components of the corresponding pixel in the image of origin. Display the result.

EXERCICE 10 (*OpenCV conversion*)

In the same code: create a matrix of `unsigned char` of the same size as the original image but with a single channel and put in this matrix the conversion of the original color image into gray levels via the function `cv::cvtColor`. Display the result. Is it different from the previous result? Save the grayscale image to hard drive.

EXERCICE 11 (*Comparison*)

Still in the same code: to make sure, create a matrix of `unsigned char` of the same size as the original image but with a single channel and put in this matrix the absolute value of 20 times the difference between the average matrix and the matrix from `cv::cvtColor` (the factor 20 is there to make sure to see something on the screen if there are only small values of difference). Display the result. Conclusion?

EXERCICE 12 (*Weighted sum*)

Still in the same code:

a) Create a matrix of `unsigned char` of the same size as the original image but with a single channel and put for each pixel `valGray` of this matrix the following weighted sum, where `valB` is the blue component of the corresponding pixel in the original image, `valG` the green one and `valR` the red one:

$$valGray = 0.114 \times valB + 0.587 \times valG + 0.299 \times valR$$

b) Display the result. Is it different from the one from the function `cv::cvtColor`?

EXERCICE 13 (*Comparison 2*)

Still in the same code: to be sure, create a matrix of `unsigned char` of the same size as the original image but with a single channel and put in this matrix the absolute value of 20 times the difference between the previous matrix and the matrix from `cv::cvtColor`. Display the result. Your final conclusion?

QUESTION 1 (*Take a step back*)

Subsidiary question: why these weighting values?

2.3 Play with BGR components

EXERCICE 14 (***A blue red Ferrari***)

With OpenCV

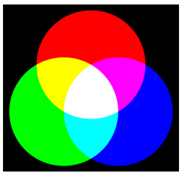
a) Load and display the image of the red Ferrari. Create a copy of the original image in a matrix.

b) How to switch from the left image to the right image very easily?

c) Code the transformation and display the result in addition to the original image.

EXERCICE 15 (***A yellow blue red Ferrari***)

Here is a graph which summarizes the colors obtained by additive synthesis.



a) From this figure, find out what modifications to make to the previous code to obtain the following cars.

b) Check your intuition by displaying the result through your program.

EXERCICE 16

What is the important difference between these 2 codes and which is easily illustrated with this exercise?

```
cv::Mat orig_img = imread(argv[1], cv::WINDOW_AUTOSIZE);
cv::Mat copy_img(orig_img);
```

```
cv::Mat orig_img = imread(argv[1], cv::WINDOW_AUTOSIZE);
cv::Mat copy_img;
copy = orig_img.clone();
```

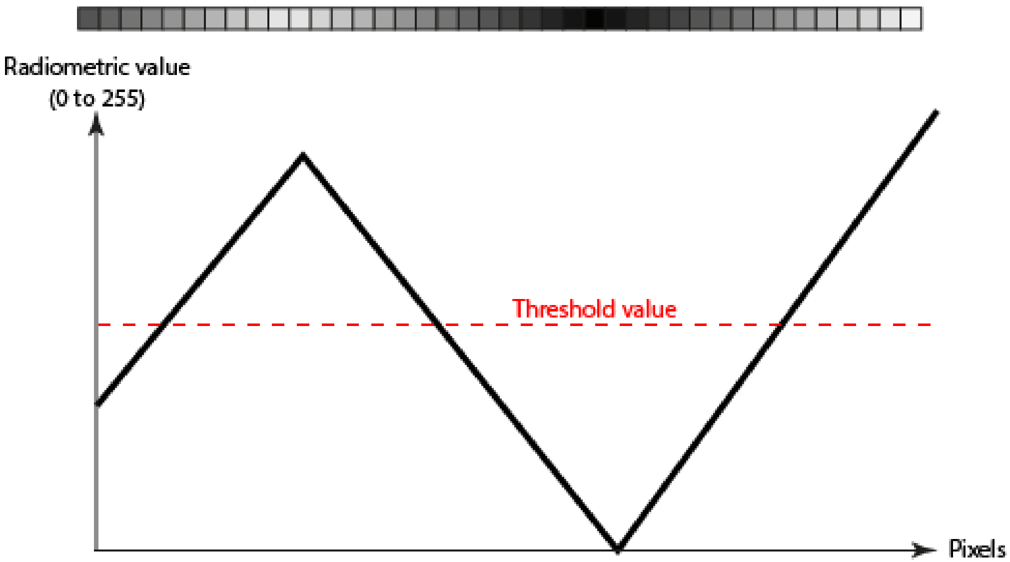
3 First image processing: thresholding

One of the most **simple** image processing consists in applying a **thresholding** on the **pixel values**. In other words, the result of a thresholding will depend on the **comparison** of the pixel value with a given threshold. Thresholding is essential when one wishes to **binarize** a result, such as for example to decide for each pixel whether it belongs to a sought class or not.

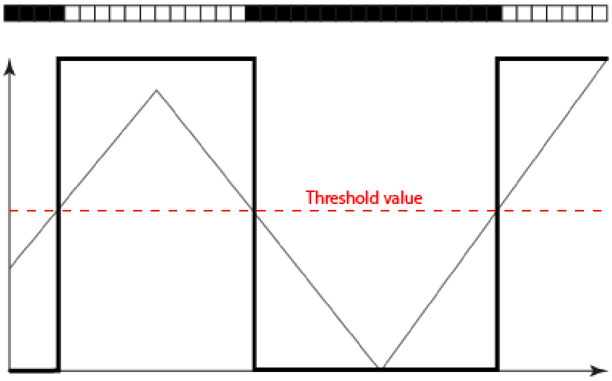
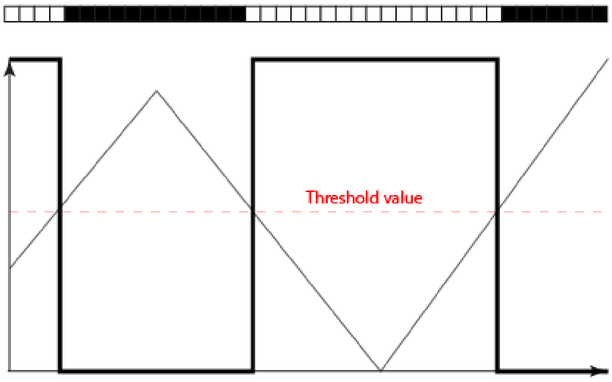
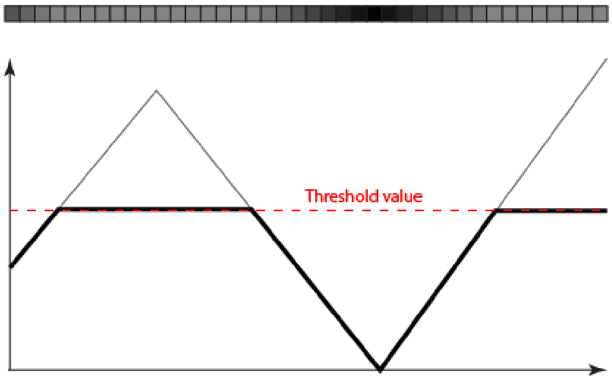
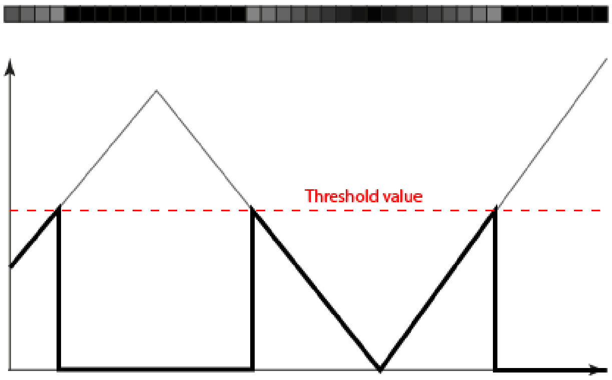
```
double cv::threshold(cv::InputArray src, cv::OutputArray dst, double thresh, double maxValue,
    int thresholdType);
```

The `cv::threshold` function *“ It allows you to make a thresholding of an image `src` at the `thresh` value in several possible ways, according to the value of `thresholdType`. `maxValue` is the saturation value for the `THRESH_BINARY` and `THRESH_BINARY_INV` methods of the following table. The result is put in the image `dst`.”*

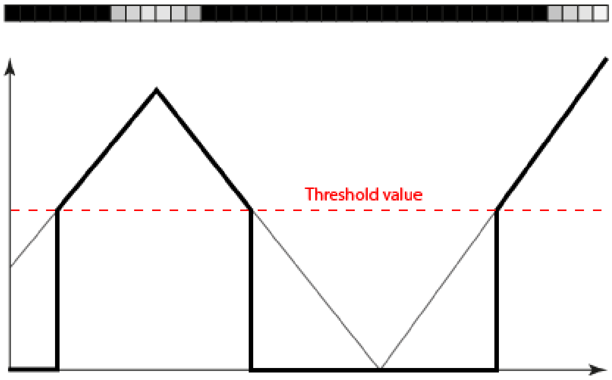
To understand what happens during thresholding, let’s take an example of a line of pixels (1 channel, gray levels with radiometric values between 0 and 255) and its radiometric profile:



The red dotted line shows the threshold value we want to apply. But there are several ways to manage this thresholding (`thresholdType`). We illustrate them in the table below on our pixel line but also on a complete image: Lena (the Mona Lisa of image processing).

Threshold type	Result on the pixel line	Image result
Binary thresholding (`cv::THRESH_BINARY`) ⇒ *Pixel value above the threshold is set to 255, below is set to 0*		
Reverse binary thresholding (`cv::THRESH_BINARY_INV`) ⇒ *Pixel value above the threshold is set to 0, below is set to 255*		
Truncated thresholding (`cv::THRESH_TRUNC`) ⇒ *Pixel value above the threshold is brought back to the threshold value (here 130)*		
Threshold at 0 inverse (`cv::THRESH_TOZERO_INV`) ⇒ *Pixel value above the threshold is brought back to 0*		

Continued on next page...

Threshold type	**Result on the pixel line**	**Image result**
Threshold at 0 (`cv::THRESH_TOZERO`) ⇒ *Pixel value below the threshold is reduced to 0*		

EXERCICE 17 (*In practice*)

With OpenCV :

a) Load the image `lena.jpg` directly in grayscale and display it.

b) Calculate and display a copy of the previous image thresholded at value 130, taking as maximum value 255 and in mode:

   1) binary thresholding,
   2) reverse binary thresholding,
   3) truncated thresholding,
   4) thresholding at zero,
   5) inverse zero thresholding.

4 Image series, video and video stream

4.1 Image set

For a large part of the applications that we saw during the "*Remote Sensing*" course, the image processing algorithms are not applied to a single image but to a large quantity of images. You should therefore be able to open all the images in a folder one after the other.

EXERCICE 18 (*Meteosat image series*)

Modify the code that open an image seen at the beginning of the seminar to open one by one all of the Meteosat images provided (the use of the `dirent` library is recommended). We will assume that all the images are in `png` format, which should make your life easier to sort between the image files (those with this extension) and any others. For each image: you will open it, display it on the screen with the file name as the window title (without the directory), then close it when the user presses a key on the keyboard to switch to the following image.

4.2 Video

In other applications, we no longer work with images but videos, and more precisely with video frames. The following code is used to display a video with OpenCV:

```
#include <iostream>
#include <opencv2/opencv.hpp>

int main(int argc, char** argv){
   cv::VideoCapture cap;
   cap.open(argv[1]);
   if(!cap.isOpened()){
      std::cerr << "Impossible to open " << argv[1] << std::endl;
      return -1;
   }
   cv::namedWindow("Video", cv::WINDOW_AUTOSIZE);
   cv::Mat frame;
   for(;;){
      cap >> frame;
      if(frame.empty()) {printf("ko\n");break;}
      cv::imshow("Video", frame);
      if (cv::waitKey(33)!=-1) break;
      //if (cv::waitKey(33)!=255) break;//opencv3
   }
   cv::destroyWindow("Video");
   return 0;
}
```

EXERCICE 19 (*Displaying videos*)

Copy, compile and test this code.

4.3 Video stream

Sometimes we want to be able to work in real time on video acquisition (video stream) and not a posteriori on recorded files.

4.3.1 Displaying a video stream

The following code is used to display the video stream of a camera with OpenCV:

```
#include <iostream>
#include <opencv2/opencv.hpp>

int main(int argc, char** argv){
   cv::VideoCapture cap;
   cap.open(0);
   if(!cap.isOpened()){
      std::cout << "Impossible to open camera" << std::endl;
      return -1;
   }
   cv::namedWindow("Camera", cv::WINDOW_AUTOSIZE);
   cv::Mat frame;
   for(;;){
      cap >> frame;
      imshow("Camera",frame);
      if(cv::waitKey(33)!=-1) break;
      //if(cv::waitKey(33)!=255) break;//for openCV3
   }
   cv::destroyWindow("Camera");
   return 0;
}
```

The management of a video stream, whether it comes from a file or from a camera, goes through the `cv::VideoCapture` object.

As with opening an image, this high-level function handles many things transparently, such as CODECs.

A **good reflex** before doing anything is to **check** that the video stream is indeed **open** via the `isOpened()` method.

```
cv::VideoCapture cap;
cap.open(const string& fileName); //example 1 : to open a file
cap.open(int deviceNumber); //example 2 : to open a camera
```

To **retrieve the current image**, you can either use the `read` method which returns `false` at the end of the video, or directly use the » operator, but in this case it is necessary to verify that the acquired image is not empty to detect the end of the video.

EXERCICE 20 (*Stream Reading*)

Copy, compile and test the previous code.

4.3.2 Saving a modified video stream

The following code applies a mirror effect around the horizontal to each frame of a video stream and saves the result in a video file.

```
#include <iostream>
#include <opencv2/opencv.hpp>

int main(int argc, char** argv){
   cv::VideoCapture cap;
```

```
    cap.open(argv[1]);
    if(!cap.isOpened()){
       std::cout << "Impossible to open " << argv[1] << std::endl;
       return -1;
    }
    cv::VideoWriter out;
    cv::Size capSize(cap.get(cv::CAP_PROP_FRAME_WIDTH),
          cap.get(cv::CAP_PROP_FRAME_HEIGHT));
    out.open("videoFlip.avi", cv::VideoWriter::fourcc('M','J', 'P', 'G'),
         cap.get(cv::CAP_PROP_FPS), capSize, true);
    if (!out.isOpened()) {
         std::cerr << "Could not open the output video file for write\n";
         return -1;
       }
    cv::namedWindow("Video", cv::WINDOW_AUTOSIZE);
    cv::Mat frame;
    cv::namedWindow("Video flip", cv::WINDOW_AUTOSIZE);
    cv::Mat frameFlip;
    for(;;){
       cap >> frame;
       if(frame.empty()) break;
       cv::imshow("Video", frame);
       cv::flip(frame, frameFlip, 0);
       cv::imshow("Video flip", frameFlip);
       out.write(frameFlip);
       if (cv::waitKey(33)!=-1) break;
       //if (cv::waitKey(33)!=255) break;//for openCV3
    }
    cv::destroyAllWindows();
    return 0;
}
```

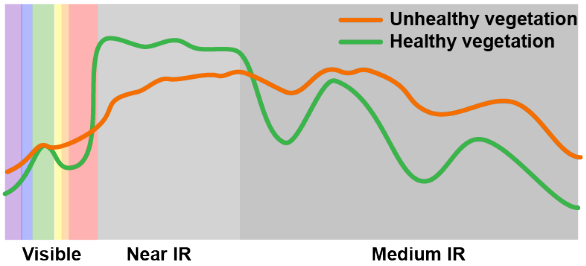
To **record a video**, you must create an object `cv::VideoWriter` and use the method `open` to open the recording stream and `write` for writing.

EXERCICE 21 (***Recording a video stream***)

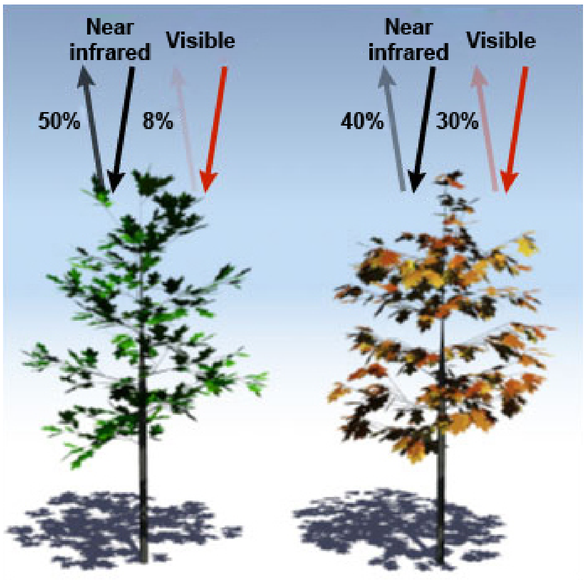
Copy, compile, launch the previous code and record for 15 seconds in a file the video stream coming from the camera of your computer.

5 Concrete application: calculation of the vegetation index (NDVI)

The **vegetation index** is a value which is calculated from **satellite images** and which expresses the **chrorophylian activity**. As we saw during the "Remote sensing" course, the **spectrum** of a green plant (chrorophylian) has a **reflectance** of the order of 20% in the visible band and 45% in near infrared.



[🅭🅯🄎 *L. Avanthey et al, 2018*]

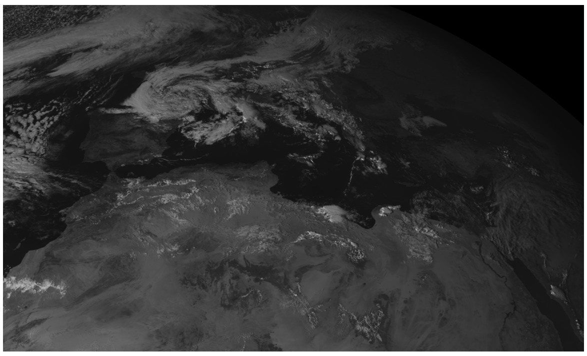


[© *Robert Simmon, Nasa, 2009*]

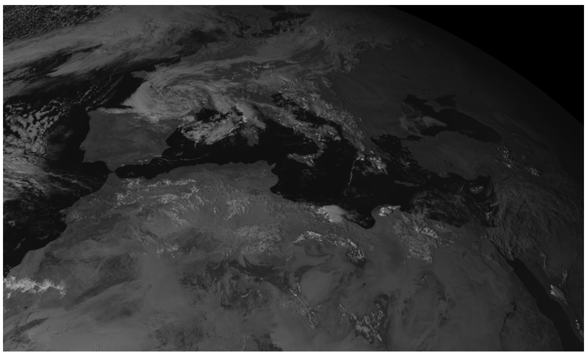
The vegetation index (there are plenty, we will focus on the best known, the NDVI) will therefore exploit this property and is calculated using the formula 1, with IR and VIS the **radiometric values** in the near infrared band ($0.8\mu m$) and in the visible band ($0.6\mu m$) respectively. The values of this index are therefore **between -1 and 1**. The **closer** a value is to **1**, the more it can be assumed that the **chlorophyll activity** is **hight**.

$$NDVI = \frac{IR - VIS}{IR + VIS} \tag{1}$$

We are going to work with portions of images taken by the **Meteosat satellite**. You will find in the archive provided the EUMETSAT technical documentation for the NDVI calculation on the SEVERI sensor [EUM15].



Visible channel (VIS6)
[© *EUMETSAT, 2018*]



Near infrared channel (VIS8)
[© *EUMETSAT, 2018*]

EXERCICE 22 (***One-off calculation of the vegetation index***)

With OpenCV:

a) Load the visible image (VIS6) and the near infrared image (VIS8).

b) Calculate the vegetation index for each pixel using the previous formula.

c) Multiply the results by 255 and display the resulting image (grayscale).

QUESTION 2 (***Critical analysis of your result***)

By observing your result, how can you simply verify that it is consistent?

QUESTION 3 (***Is it sufficient ?***)

What can you say about NDVI in France? What should be done to obtain a satisfactory index for the whole of Europe?

EXERCICE 23 (***One-off calculation of the vegetation index***)

With OpenCV, starting from the previous code, modify it to:

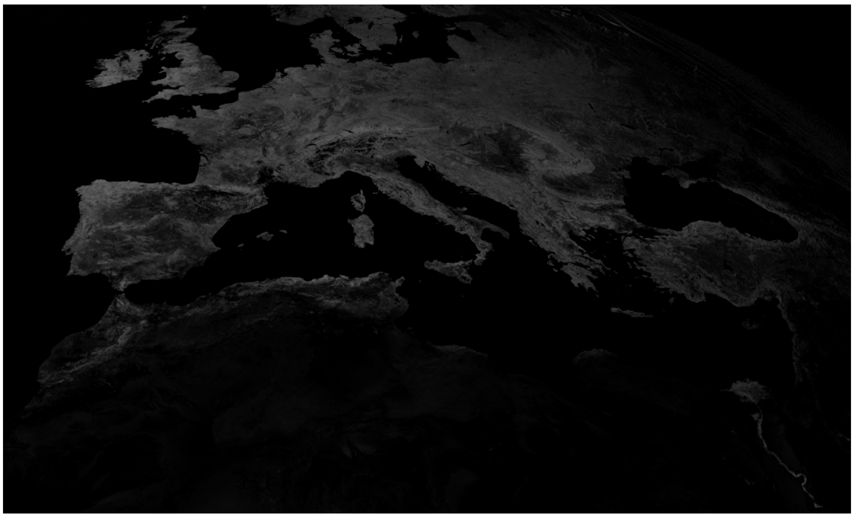
a) Load the next couple (VIS6 and VIS8), and calculate the vegetation index for each pixel as in the previous exercise.

b) Update the resulting image, keeping the max for each pixel between these new NDVI values and the previous ones. Display the result (in the same window as before).

c) Repeat the previous two steps as long as there are couples: at runtime you will see the NDVI complete as you go.

EXERCICE 24 (***Crop***)

With OpenCV, modify your previous code to calculate the NDVI only on a rectangle centered on France and save the result.



Annexe A: Basic object operations

Annexe A.1: Point

Operation	Example
Default constructors	`cv::Point{2,3}{i,f}` (`cv::Point2i`, `cv::Point3f`...)
Copy constructor	`cv::Point3f p2(p1);`
Constructor by values	`cv::Point2i p(2,4);`
Cast	`(cv::Vec3f) p;`
Access to attributes	`p.x`, `p.y`, `p.z`
Scalar product	`float x = p1.dot(p2); double x = p1.ddot(p2);`
Vector product	`p1.cross(p2);` (for 3D points only)
Check if a point is in a rectangle	`p.inside(r);`

Table 2 – *Operations supported by the* `Point` *class.*

Annexe A.2: Vec

Operation	Example
Default constructors	`cv::Vec2s v2s;`, `cv::Vec3f v3f;`, `cv::Vec2b v2b;`
Copy constructor	`cv::Vec3f u3f(v3f);`
Constructor by values	`cv::Vec2f v2f(x0, x1);`, `cv::Vec3f v3f(x0, x1, x2);`
Access to attributes	`v2b[0];`, `v3f[2];`
Vector product	`v3f.cross(u3f);`

Table 3 – *Some operations supported by the* `Vec` *class.*

Annexe A.3: Scalar

Operation	Example
Default constructors	`cv::Scalar s;`
Copy constructor	`cv::Scalar s2(s1);`
Constructor by values	`cv::Scalar s(2.1, 3.4, -5.4, 4.7);`
Access to attributes	`s[0]`, `s[1]`, `s[2]`, `s[3]`
Term by term product (Hadamar)	`s1.mul(s2);`
Conjugate	`s.conj();` (return `cv::Scalar (s[0],-s[1],-s[2],-s[3])`)
Check if it's a real number	`s.isReal();` (return `true` if `s[1]=s[2]=s[3]=0`)

Table 4 – *Operations supported by the* `Scalar` *class.*

Annexe A.4: Size

Operation	Example
Default constructors	`cv::Size sz;`, `cv::Size2i sz;`, `cv::Size2f sz;`
Copy constructor	`cv::Size sz2(sz1);`
Constructor by values	`cv::Size sz(w, h);`
Access to attributes	`s[0]`, `sz.width;`, `sz.height;`
Calculation on an area	`sz.area();`

Table 5 – *Operations supported by the `Size` class.*

Annexe A.5: Rect

Operation	Example
Default constructors	`cv::Rect r`
Copy constructor	`cv::Rect r2(r1);`
Constructor by values	`cv::rect r(x,y,w,h);`
Constructor by origin and size	`cv::Rect r(p, sz);`
Constructor by opposite corners	`cv::Rect r(p1, p2);`
Access to attributes	`r.x`, `r.y`, `r.width`, `r.height`
Calculation on an area	`r.area();`
Extract the upper left corner	`r.tl();`
Extract the lower right corner	`r.br();`
Check if a point is in the rectangle	`r.contains(p);`
Intersection of rectangles	`cv::Rect r3 = r1 & r2;`
Common area for rectangles	`cv::Rect r3 = r1 | r2;`
Offset a rectangle	`cv::Rect rx = r + p`
Enlarge a rectangle	`cv::Rect rs = r + sz`
Check if 2 rectangles are equal	`bool eq = (r1 == r2);`
Check if 2 rectangles are different	`bool neq = (r1 != r2);`

Table 6 – *Operations supported by the `Rect` class.*

Annexe A.6: RotatedRect

Operation	Example
Default constructor	`cv::RotatedRect rr;`
Copy constructor	`cv::RotatedRect rr2(rr1);`
Constructor by 2 opposite corners	`cv::RotatedRect rr(p1,p2);`

Continued on next page ...

Operation	Example
Constructor by values	`cv::RotatedRect rr(p, sz, angle);`
Access to attributes	`rr.center`, `rr.size;`, `rr.angle;`
Return the list of 4 corners	`rr.points(pts[4]);`

Table 7 – *Operations supported by the* ***RotatedRect*** *class.*

Annexe A.7: Complex

Operation	Example
Default constructor	`cv::Complexf z1;`, `cv::Complexd z2;`
Copy constructor	`cv::Complexf z2(z1);`
Constructor by values	`cv::Complexd z2(re0,im1);`
Access to attributes	`z.re`, `z.im;`
Complex conjugate	`z2 = z1.conj();`

Table 8 – *Operations supported by the* ***Complex*** *class.*

Annexe B: Permitted operations for dense matrices

Annexe B.1: Constructors

Operation	Example
Default constructor	cv::Mat; (to use with create() and setTo[])
2D matrix by type	cv::Mat(int rows, int cols, int type);
2D matrix with pre-existing data	cv::Mat(int rows, int cols, int type, void* data, size_t step=AUTO_STEP);
2D matrix with initialization	cv::Mat(int rows, int cols, int type, cv::Scalar& s);
2D matrix by type (variant)	cv::Mat(cv::Size sz, int type);
2D matrix by type (variant with initialization)	cv::Mat(cv::Size sz, int type, const cv::Scalar& s);
nD matrix by type	cv::Mat(int ndims, const int* sizes, int type);
nD matrix by type with initialization	cv::Mat(int ndims, const int* sizes, int type, const cv::Scalar& s);

Table 9 – *Some direct constructors for the* Mat *class.*

Operation	Example
Copy constructor	cv::Mat(const Mat& mat);
Copy of a subset (ROI for Region of Interest)	cv::Mat(const Mat& mat, const cv::Rect& ROI);

Table 10 – *Some constructors per copy for the* Mat *class.*

Operation	Example
Default constructor	cv::Mat; (à utiliser avec create() et setTo[])
Create a matrix filled with zero	cv::Mat::zeros(rows, cols, type);
Create a matrix filled with 1	cv::Mat::ones(rows, cols, type);
Create an identity matrix	cv::Mat::eye(rows, cols, type);

Table 11 – *Special constructors for the* Mat *class.*

Operation	Example
Whole element i of a matrix M	M.at<int>(i);
Element floating in position (i, j) of a matrix M	M.at<float>(i,j);
Whole element at point p (i.e. at coordinates p.x, p.y) of a matrix M	M.at<int>(p);
Floating element of a three-dimensional M matrix	M.at<float>(i,j,k);

Table 12 – *Examples of using the* at<>() *accessor.*

Operation	Example
Table of elements of the line `i`	`M.row(i);`
Table of elements in the column `j`	`M.col(j);`
Table of lines between `i0` and `i1-1`	`M.rowRange(i0,i1);`
Table of columns between `j0` and `j1-1`	`M.colRange(j0,j1);`
Array of elements of the matrix `M`	`M.diag();`
Table of elements included in a rectangle	`M.( cv::Rec(i0,i1,w,h) );`

Table 13 – *Methods to recover data by blocks in an array.*

Operation	Example
Addition; substraction	`m + n; m - n;`
Matrix multiplication	`m*n;`
Matrix inversion (by default, `method = DECOMP_LU`)	`m.inv( method);`
Transposed	`m.t();`
Comparison by element; the result is a matrix `uchar` of 0 or 255	`m>n; m<n; m==n; m>=n m<=n`
Absolute value	`cv::abs(m);`

Table 14 – *Some examples of algebraic calculations under OpenCV.*

Operation	Example
Full copy of the matrix `m0`	`m1 = m0.clone();`
Initialization of a matrix `m` by the scalar `s` at the positions of the non-zero values of the matrix `mask`	`m.setTo(s, mask);`
Define the area of interest of the `m` matrix of size `size` and upper left corner `offset`	`m.locateROI(size, offset);`
Retrieve the type of the matrix (ex: `CV_32FC3`)	`m.type();`
Retrieve the type of each image plane (ex: `CV_32F`)	`m.depth();`
Retrieve the number of channels	`m.channels();`
Get the size `cv::Size` of the matrix	`m.size();`
Check that the matrix does not contain data	`m.empty();`
Convert the elements of m0 into the matrix m1, changing the type of m1 to type with a scale factor and offset	`m0.convertTo(m1, type, scale, offset);`

Table 15 – *Some methods for handling the `cv::Mat` object.*

Function	Use
`cv::abs()`	calculation of the absolute value per element
`cv::absdiff()`	calculation of the absolute value per element of the difference between 2 matrices
`cv::add()`	addition of 2 matrices per element

Continued on next page ...

Function	Use
`cv::addWeighted()`	weighted addition of 2 matrices
`cv::bitwise_and()`	calculates the logical AND operation by element
`cv::bitwise_not()`	calculates the logical operation NO by element
`cv::bitwise_or()`	calculates the logical OR operation by element
`cv::bitwise_xor()`	calculates the logical EXCLUSIVE OR operation by element
`cv::calcCovarMatrix()`	calculates the covariance of a set of vectors
`cv::cartToPolar()`	calculates the angle and magnitude of a vector field
`cv::checkRange()`	checks the validity of the values of a matrix
`cv::compare()`	compare matrices
`cv::completeSymm()`	symmetrically matrix by copying one half on another
`cv::convertScaleAbs()`	scales and calculates the absolute value of a matrix
`cv::countNonZero()`	counts the number of non-zero elements in the matrix
`cv::cvtColor()`	allows to pass a matrix from a color space to another (`cv::COLOR_BGR2GRAY` for example)
`cv::dct()`	calculates the discrete Cosine transform
`cv::determinant()`	calculates the determinant of a square matrix
`cv::dft()`	computes the discrete Fourier transform
`cv::divide()`	calculates the division by element of one matrix by another
`cv::eigen()`	calculates the eigenvalues of a square matrix
`cv::exp()`	calculates the exponentiation by element
`cv::flip()`	flip of a matrix along an axis
`cv::gemm()`	calculates the generalized multiplication of matrices
`cv::idct()`	compute the inverse of the discrete cosine transform
`cv::idft()`	computes the inverse of the discrete Fourier transform
`cv::inRange()`	tests if the elements of the matrix are in defined ranges of values
`cv::invert()`	reverse a square matrix
`cv::log()`	calculates the natural logarithm of a matrix per element
`cv::magnitude()`	calculates the magnitude of a vector field
`cv::LUT`	conversion by element using the values of the LookUp Table
`cv::Mahalanobis()`	calculates the Mahalanobis distance between 2 vectors
`cv::max()`	calculate the maximum per element between 2 matrices
`cv::mean()`	calculates the average value of the matrix
`cv::meanStdDev()`	calculates the mean value and the standard deviation of the matrix
`cv::merge()`	merges multiple single-channel matrices into a multi-channel matrix

Continued on next page ...

Function	Use
`cv::min()`	calculate the minimum per element between 2 matrices
`cv::minMaxLoc()`	find the minimum and maximum of the matrix
`cv::mixChanels()`	mix matrices
`cv::mulSpectrum()`	calculates the multiplication of Fourier spectra
`cv::multiply()`	calculates the multiplication by element of 2 matrices
`cv::mulTransposed()`	calculates the product of a matrix with its transpose
`cv::norm()`	calculates different norms (L1 or L2)
`cv::normalize()`	normalizes by element
`cv::perspectiveTransform()`	calculates the homogeneous coordinates of a list of vectors
`cv::phase()`	calculates the orientations of a vector field
`cv::polarToCart()`	calculates the vector field from angles and magnitudes
`cv::pow()`	calculate power per element
`cv::randu()`	fills a matrix with random numbers using a uniform distribution
`cv::randn()`	fills a matrix of random numbers following a normal distribution
`cv::randShuffle()`	randomly mix the elements of a matrix
`cv::reduce()`	reduce a 2D matrix to a vector
`cv::repeat()`	tile a matrix in another
`cv::sature_cast<>()`	cast without over or underflow
`cv::scaleAdd()`	scales and calculates the sum of 2 matrices
`cv::setIdentity()`	transform a matrix into the identity matrix
`cv::solve()`	solves a system of linear equations
`cv::solvecubic()`	find the real roots of a cubic equation
`cv::solvepoly()`	finds the complex roots of a polynomial equation
`cv::sort()`	sorts rows or columns separately
`cv::sortIdx()`	sorts rows or columns but without modifying the matrix
`cv::split()`	separates a multi-channel matrix into several single-channel matrices
`cv::sqrt()`	calculates the square root per element of a matrix
`cv::substract()`	calculates the subtraction by element between 2 matrices
`cv::sum()`	calculates the sum of all elements of the matrix
`cv::trace()`	calculates the trace of the matrix
`cv::transform()`	apply a transformation on all the elements of the matrix
`cv::tanspose()`	calculates the transpose of the matrix

Table 16 – *Functions for handling `cv::Mat`.*

Annexe C: CMake

CMake is a "production engine" also called "software construction system" (build systems). It was created in 2000 by the Kitware company as part of the Visible Human Project (creation of a database of photographs of sections of the human body). It is supplied under BSD license and it is multilingual (C, C++, Java, etc.) and multiplatform (Linux, MacOS, Windows).

The **project description** (list of source files, libraries, etc.) is done completely **independent of the configuration** of the machine on which the project will be compiled and executed. It is performed in files named `CMakeLists.txt` which are placed in the source directory. It is then the construction system which will be responsible for retrieving information specific to the software configuration of the user's machine and which, thanks to the description file, can then automatically produce a textbf dedicated compilation script: the `Makefile`.

To install the tool on your system (if not already done), go to the official website:

https://cmake.org/install/

References

[Bra13] Samarth Brahmbhatt. Practival OpenCV, hands on projects for computer vision on the Windows, Linux and Raspberry Pi platforms. Technology In Action, 2013. 1

[EUM15] EUMETSAT. Normalised Difference Vegetation Index: Product Guide. https://www.eumetsat.int/website/wcm/idc/idcplg?IdcService=GET_FILE&dDocName=PDF_NDVI_PG&RevisionSelectionMethod=LatestReleased&Rendition=Web, 2015. 15

[GL11] Jean-Michel Géridan and Jean-Noël Lafargue. Processing, le code informatique comme outil de création. Pearson, 2011.

[KB17] Adrian Kaehler and Gary Bradski. Learning OpenCV 3, Computer Vision in C++ with the OpenCV Library. O'Reilly, 2017. 1

[RF14] Casey Reas and Ben Fry. Processing: A Programming Handbook for Visual Designers (Second Edition). MIT Press, 2014.