1 General remarks

In order to assess your work quickly, we ask you to follow some simple rules:

- You must send an archive named YOURNAME_LabX.tar.bz2 attached to an e-mail with AIVI in the subject
- We shall be able to decompress this archive with `tar xvjf YOURNAME_LabX.tar.bz2` and get a directory YOURNAME_LabX. This directory will contain all your work (source code, figures, report if any).
- We must be able to compile your work quickly. Thus provide all the necessary source code files AND the CMakeLists.txt.
- DO NOT SEND useless files, such as the videos, the executable, the build directory, the lab pdf file, __MACOSX directories, ...

Some recommendations for the source code:

- Clean your code before sending it. In particular, remove all the dead code. It may be a good idea to also make functions when necessary and add some comments.
- Correct the warnings produced during compilation. Warnings are here to help you. In particular, warnings about uninitialized values should hint you that you will probably not get the expected correct result.
- Check for errors. In particular, if you make system calls (open a file, create a directory, ...), check for returned error codes, in order to not fail silently.
- Do not hard code paths. We will probably not have the same paths on our computer.

2 OpenCV

OpenCV is a library aimed at real-time computer vision. It was first developped by Intel and is now maintained by the russian company Itseez (bought by Intel).

OpenCV provides lots of functionnalities (image processing, object detection, camera calibration, classification, ...). Some of these functionalities may run on the GPU (via CUDA or OpenCL code).

The code is mainly in C++ (although bindings exist for python and java). These labs will be in C++.

These labs require OpenCV version 2.4.x. They will not compile with OpenCV 3.x.

You can usually check OpenCV version with the following command:

```
pkg-config --modversion opencv
```

Currently on the CREMI computers, the version 2.4.9 of OpenCV is installed. The documentation is available here: `http://docs.opencv.org/2.4.9/`.

3 CMake

For these labs, CMake is used to configure the build process.

CMake is a software that manages the build process, in conjunction with the native build environment (make, ninja, Xcode, Visual Studio, ...). On linux, we will use it to generate a Makefile.

It is recommended to use out-of-place builds, that is we will build our software in a different directory than where the sources are (for example a *build* directory). This way we can remove all the compiled files easily (we just have to remove the *build* directory), or we can compile the sources with different configurations (*Release* and *Debug* mode for example, see below).

CMake uses one or several files named CMakeLists.txt to describe the build process. In these labs, we have one CMakeLists.txt in the source directory. This file describes in particular which source files to compile and that the executable depends on OpenCV.

When you call the *cmake* executable (from the command line), you just have to pass the path to the CMakeLists.txt file.

If we are in the source directory, we can configure (that is, produce a Makefile) and build our project this way:

```
mkdir build
cd build
cmake ..
make
```

Here we make a directory *build* in which we will build our project. When we are in this *build* directory, the CMakeLists.txt is in the parent directory, that is "*..*". The *cmake ..* step will produce the Makefile, and then we do *make* to compile the project.

If you change one source file, you just have to recompile, i.e., just to type *make*.

One interesting parameter of the configuration process is to specify if we want to compile in *Debug* or *Release* mode. In *Debug* mode, you will be able to debug your code (with gdb for example) and if you have *assert* in your code, the conditions will be checked. In *Release* mode, the *assert* will be ignored/skipped, your code will also be much difficult to debug, but it should execute faster. To choose the compilation mode during configuration, you can type for example:

```
cmake .. -DCMAKE_BUILD_TYPE=Release
```

It is recommended to first build your project in *Debug* mode during the design phase. Then, when your program works correctly, build in *Release* mode, to produces the results faster.

Graphical interfaces are also available for cmake: *ccmake* and *cmake-gui*. From the command line, you can use *ccmake*.

To see which commands are used during the build process, you can type the following:

```
make VERBOSE=1
```

During the configuration process, CMake uses a cache to keep some information. If you want to configure your project again, from scratch, you can remove the following file and sub-directory from your *build* directory : *CMakeCache.txt* and *CMakeFiles*. For example:

```
rm -rf CMakeCache.txt CMakeFiles
```

4 Lab1

4.1 Reading a video file

There is an example in the OpenCV documentation almost doing this: see `http://docs.opencv.org/2.4.9/modules/highgui/doc/reading_and_writing_images_and_video.html#videocapture`. We just have to modify this example a bit to read from a video filename and remove useless operations.

Here is an example of code playing a video and saving each color frame:

```
#include <sstream>
#include <iomanip>

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>


int
main(int argc, char **argv)
{
  if(argc != 2) { //[A]
    std::cerr<<"Usage: "<<argv[0]<<" videoFile"
             <<"\n";
    return EXIT_FAILURE;
  }

  const char *videoFilename = argv[1];

  cv::VideoCapture cap(videoFilename);  //[B]
  if ( ! cap.isOpened()) {
    std::cerr<<"Error: Unable to open file "
             <<videoFilename<<"\n";
    return -1;
  }
```

```
  const size_t deltaT = delta;

  unsigned long frameNumber = 0;
  for ( ; ; ) {

    cv::Mat frameBGR; //[C]
    cap >> frameBGR;  //[D]
    if (frameBGR.empty()) {
      break;
    }

    sd::stringstream ss; //[E]
    ss<<''frame_''
      <<std:setfill('0')<<std::setw(6)<<frameNumber
      <<''.png'';
    cv::imwrite(ss.str(), frameBGR); //[F]

    ++frameNumber;
  }

  cap.release(); //[G]

  return EXIT_SUCCESS;
}
```

A : Minimal verification of program parameters. Reminder: argc is the number of parameters passsed to the program, including the program name itself as first paramter.

B : We use a `VideoCapture` object (from the highgui module) to read the video file. See `http://docs.opencv.org/2.4.9/modules/highgui/doc/reading_and_writing_images_and_video.html#videocapture`. All types of OpenCV are in the namesapce `cv`.

C : `Mat` is for Matrix. It is one of the main types of OpenCV. It is used in particular to store images.

D : Here, operator `>>` of `VideoCapture` is used to get a new frame from the video, as a `Mat`. **Images are in BGR format by default in OpenCV. That is, images have three channels: first blue, second green, third red**. Images are stored in memory as dense arrays of pixels (that is `BGR, BGR, BGR, ...`) and not as planes per channel (that is a plane for Blue `B, B, B, ...`, a plane for Green `G, G, G, ...`, and a plane for Red `R, R, R, ...`). See `http://docs.opencv.org/2.4.9/modules/core/doc/basic_structures.html#mat` for more information on `Mat` class.

E : We want to save each image with a different name (otherwise we would overwrite the same file each time). We use a stringstream object to build the filename with the frame number in it. Here, `setfill()` and `setw()` are stream manipulators: they are used to always have 6 digits whatever the frame number. For example, we will have 000001 for the frame 1.

F : `imwrite` (also in the highgui module) is used to write a `Mat` to a file.

G : we close the video file before exiting. Actually, this call is useless as it will be automatically called by the `VideoCapture` object destructor.

4.2 Δt

We want to compute the various measures (MSE, PSNR, ...) between two luminance frames distance of Δt frames ($\Delta t > 0$). We have to convert the RGB frames to luminance frames, that is keep only the Y channem of frames converted to YCrCb. And we have to store Δt frames. We will then compute the measure between the first frame and the last frame of our container. To implement such a FIFO (*first-in first out*), we can use the C++ *queue* container.

Here is an example of code playing a video and calling a function to compute the Mean Squared Error (MSE) between two luminance frames distant of Δt frames:

```
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>


int
main(int argc, char **argv)
{
  if(argc != 3) { // [A]
    std::cerr<<"Usage: "<<argv[0]<<" videoFile deltaT"
             <<"\n";
    return EXIT_FAILURE;
  }

  const char *videoFilename = argv[1];

  const int delta = atoi(argv[2]);
  if (delta <= 0) {
    std::cerr<<"Error: deltaT must be strictly positive\n";
    return EXIT_FAILURE;
  }

  cv::VideoCapture cap(videoFilename);
  if ( ! cap.isOpened()) {
```

```
    std::cerr<<"Error: Unable to open file "
             <<videoFilename<<"\n";
    return -1;
  }

  const size_t deltaT = delta;
  std::queue<cv::Mat> previousFrames; //[B]

  unsigned long frameNumber = 0;
  for ( ; ; ) {

    cv::Mat frameBGR;
    cap >> frameBGR;
    if (frameBGR.empty()) {
      break;
    }


    cv::Mat frameYCbCr;
    cv::cvtColor(frameBGR, frameYCbCr, CV_BGR2YCrCb);//[C]

    std::vector<cv::Mat> planes(3); //[D]
    cv::split(frameYCbCr, planes);
    const cv::Mat Y = planes[0];

    if (previousFrames.size() >= deltaT-1) {  //[E]
      cv::Mat prevY = previousFrames.front();
      previousFrames.pop();

      const double MSE = computeMSE(Y, prevY);

      std::cout<<frameNumber<<" "<<MSE<<"\n";
    }

    previousFrames.push(Y);

    ++frameNumber;
  }

  cap.release();

  return EXIT_SUCCESS;
}
```

A : Minimal verification of program parameters. We have one more param-

eter here: Δt.

B : `queue` of `Mat`. `Mat` is for Matrix. It is one of the main types of OpenCV. It is used in particular to store images. This queue will help us to store the Δt frames.

C : We convert the current frame from `BGR` color space to `YCrCb` color space.

D : We are interested only in the `Y` channel (luminance/luma) of the `YCrCb` frame. We split the matrix in three individual planes (Y, Cr, Cb) and keep only the plane corresponding to `Y` channel.

E : Here, we have enough (Δt) frames. We can compute the MSE between the current frame and the first in the FIFO.

4.3 MSE

We want to compute the Mean Square Error (MSE) between two images. To do that we have to traverse both images and compute the the square of the difference between the pixels at same coordinates in both images (and sum the result).

There are several ways to access the pixels (and thus compute the MSE) in OpenCV. Each way has some pitfalls.

4.3.1 Pixel access

A first method to compute the MSE, that we can qualify of low level method, consists in accessing all the image pixels, pixel by pixel.

Simple version The following code gives a simple implementation of a function computing the MSE, pixel by pixel.

```
double
computeMSE(cv::Mat m1, cv::Mat m2)
{
  assert(m1.size() == m2.size()
         && m1.type() == m2.type()
         && m1.type() == CV_8UC1); //[A]

  unsigned long sum = 0; //[B]
  for(int i = 0; i < m1.rows; ++i) {
    for(int j = 0; j < m1.cols; ++j) {
      const int diff = m1.at<uchar>(i, j)
                     - m2.at<uchar>(i, j); //[C]
      sum += diff * diff;
    }
  }
  const double MSE = sum/(double)(m1.rows*m1.cols); //[D]
```

```
    return MSE;
}
```

Some remarks on this code:

A : this is a good practice to check that the matrix size and type are really what is expected. Indeed, OpenCV will not warn you if you access a matrix with the wrong element type or out-of-bounds. The macro `assert()` (available in `cassert` header) can help. Here, in particular, we check that the type of the matrices or images is `CV_8UC1`. That is we have only one channel (represented by `C1`) and each element (or pixel) of the channel is of type `unsigned char` or `uchar` (represented by `8U`, for 8-byte unsigned). Other types are available. For example, our BGR frames are of type `CV_8UC3` : 3 channels and each pixel of type `uchar`. We could also have for example a matrix of type `CV_32FC4`, that is 4 channels and each pixel of type `float` (represented by `32F`).

B : watch out for the type used for accumulation. We will store *square of differences* thus positive values, and we will accumulate a lot of them (*m1.rows*m1.cols*), so we should use a big enough (unsigned) type.

C : here my matrices are of type `CV_8UC1`, so elements are of type `unsigned char` or `uchar`. We access elements/pixels with the `at<uchar>()` matrix method. Accessing pixels with `at<>()` is not necessarily the fastest method (see *Advanced version* below). Watch out for the type used to store the difference. The two values are unsigned chars, but the difference will be signed and potentially bigger than unsigned char. So we should use a big enough signed type.

D : watch out for the divide operation. We have used integer types (for `sum` and `m1.rows*m1.cols`) but we want a floating point divide.

One remark on performances:

- OpenCV 2D Matrices are stored row-by-row in memory. That is, first we have all the data (that is the values of each column) for row 0, then all the data for row 1, and so on. This layout has an influence on how you should access your matrix for maximum performance. Indeed, when you access a given address, your processor will put in its cache(s) some of the data at the following addresses. Then if you access these following addresses data, access will be faster as it is already in the cache. In our case, as 2D Matrices are stored row-by-row, when we access an element of a given row, the cache will hold some of the following elements of the row. Thus if we traverse our matrices row-by-row, we take advantage of the cache, and it is faster than to access them column-by-column (for example). So if you look at the code above, it takes care to traverse the matrix row-by-row. That is we have a first for loop on the rows, then a for loop on the columns. **You should respect this order in your code when you**

traverse the whole image: the first for loop is on the rows, the second for loop is on the columns.

Advanced version The following code gives a more efficient implementation of a function computing the MSE, pixel by pixel. It does not use the rather costly `at<>()` matrix method, and takes into account that pixels allocated for matrices are contiguous in memory.

To fully understand this code, I encourage you to read OpenCV documentation about `cv::Mat` here `http://docs.opencv.org/2.4.9/modules/core/doc/basic_structures.html#mat`, and in particular the paragraph beginning with: "`The next important thing to learn about the array class is element access`".

```
double
computeMSE(const cv::Mat &m1, const cv::Mat &m2)
{
  assert(m1.size() == m2.size()
         && m1.type() == m2.type()
         && m1.type() == CV_8UC1);

  unsigned long sum = 0;

  int rows = m1.rows;
  int cols = m1.cols;
  if (m1.isContinuous() && m2.isContinuous()) { //[A]
    cols *= rows;
    rows = 1;
  }
  for (int i = 0; i<rows; ++i) { //[B]
    const unsigned char *p1 = m1.ptr<unsigned char>(i);//[C]
    const unsigned char *p2 = m2.ptr<unsigned char>(i);
    for(int j = 0; j<cols; ++j) {
      const int diff = p1[j]-p2[j];
      sum += diff * diff;
    }
  }
  const double MSE = sum / (double)(m1.rows*m1.cols);
  return MSE;
}
```

- The same remarks than for the simple version of the code apply, in particular about types.

A : first, try to understand the code without this `if`. This `if` block is a clever way to traverse the matrix data as one big row if the data is

contiguous. Instead of traversing `rows` rows of `cols` pixels, we traverse `1` row of `cols*rows` pixels if the data is contiguous.

B : the way we traverse the image data has an impact on performances. Here, we take advantage of the processor cache. **The first for loop is on the rows, the second for loop is on the columns**.

C : Here, we get a pointer on the row `i`. We can then access directly the j-th column with an array access. It is faster than accessing with `at<uchar>()` (that has to do a multiplication).

4.3.2 Matrix operations

We can also compute the MSE using high level matrix operations.

One way to compute the MSE is the following:

```
double
computeMSE(cv::Mat m1, cv::Mat m2)
{
   assert(m1.size() == m2.size()
          && m1.type() == m2.type()
          && m1.type() == CV_8UC1);

   cv::Mat m;
   cv::absdiff(m1, m2, m); //[A]

   m.convertTo(m, CV_32S); //[B]
   m = m.mul(m);

   const double sum = (cv::sum(m))[0]; //[C]

   const double MSE = sum / m.total();   //[D]
   return MSE;

}
```

Some remarks about this code:

A : `cv::absdiff` is used to compute the absolute difference between two matrices. The type of the result matrix m is the same than $m1$ and $m2$, that is `CV_8UC1`, and there is no loss of precision or truncation as the absolute difference of two unsigned char is still an unsigned char.

B : as the square of the values of m (the differences) will not fit in an `unsigned char`, we first convert the matrix to a larger type. The next command `m = m.mul(m)` compute the square of values and they fit in a *int* type.

C : `cv::sum` is used to compute the sum of all channels of a given matrix. It returns a vector and as we have only one channel, we take its first element.

D : `m.total()` is equivalent to `m.rows*m.cols`.

This code may be slower than the pixel-by-pixel version as we traverse several times the matrices.

The following code shows an other implementation

```
double
computeMSE(cv::Mat m1, cv::Mat m2)
{
  assert(m1.size() == m2.size()
         && m1.type() == m2.type()
         && m1.type() == CV_8UC1);

  m1.convertTo(m1, CV_32FC1);
  m2.convertTo(m2, CV_32FC1);

  cv::Mat e;
  cv::subtract(m1, m2, e); //[A]

  e = e.mul(e);

  const double sum = (cv::sum(e))[0];

  const double MSE = sum / e.total();
  return MSE;
}
```

A : Here, we first convert each matrix to a floating point type and then compute the difference with `cv::subtract`.

The following code shows an other implementation

```
double
computeMSE(cv::Mat m1, cv::Mat m2)
{
  const double l2 = cv::norm(m1, m2, cv::NORM_L2); //[A]

  const double MSE = (l2*l2)/(m1.rows*m1.cols);
  return MSE;
}
```

A : Here, we use the `cv::norm()` function to compute the L2 norm of $m1 - m2$.

The following code shows an **incorrect** implementation

```
double
computeMSE_WRONG(cv::Mat m1, cv::Mat m2)
{
  assert(m1.size() == m2.size()
         && m1.type() == m2.type()
         && m1.type() == CV_8UC1);

  cv::Mat m;
  cv::pow(m1-m2, 2, m);   //[A]

  const double MSE = cv::sum(m)[0] / m.total();
  return MSE;
}
```

A : there is two errors here. First we compute a temporary matrix $m1 - m2$ that will have the same type than m1 and m2 (`CV_8UC1`) and thus will not have enough precision to store the difference. Then we compute the square with `cv::pow()`, that will store its result in a matrix of the same type than $m1 - m2$, that is `CV_8UC1`, that will not have enough precision to store the square value.

4.4 PSNR

PSNR is easily computed from MSE. We can have the following implementation:

```
double
computePSNR(double MSE)
{
  assert(MSE > 0);
  const double PSNR = 10*log10((255*255)/MSE);
  return PSNR;
}
```

4.5 Entropy

Same as for MSE, there are several ways to compute the Entropy.

4.5.1 Pixel access

Here are two implementations of Entropy computation with pixel accesses.

Simple version Slow implementation with `at<>()` calls for pixel access.

```
double
computeEntropy(cv::Mat m)
{
  assert(m.type() == CV_8UC1);

  //- compute histogram
  const int size = 256;
  long hist[size] = {0}; //[A]

  for (int i = 0; i < m.rows; ++i) {
    for(int j = 0; j < m.cols; ++j) {
      hist[m.at<uchar>(i, j)] += 1;
    }
  }

  //- compute entropy
  double entropy = 0.0;
  for (int i=0; i<size; ++i) {
    if (hist[i] > 0) {
      const double pi = (hist[i])/(double)(m.rows*m.cols);

      entropy -= pi * log2(pi);
    }
  }

  return entropy;
}
```

A : Here `hist` is a simple static array. We use the special initialization that sets all the values of the array to zero.

Advanced version Here is a most efficient version of the same code:

```
double
computeEntropy(cv::Mat m)
{
  assert(m.type() == CV_8UC1);

  //- compute histogram
  const int size = 256;
  long hist[size] = {0};

  int rows = m.rows;
  int cols = m.cols;
  if (m.isContinuous()) {
```

```
    cols *= rows;
    rows = 1;
  }
  for (int i = 0; i < rows; ++i) {
    const unsigned char *p = m.ptr<unsigned char>(i);
    for(int j = 0; j < cols; ++j) {
      hist[p[j]] += 1;
    }
  }

  //- compute entropy
  double entropy = 0.0;
  const double norm = 1. /(double)(rows*cols);
  for (int i=0; i<size; ++i) {
    if (hist[i] > 0) {
      const double pi = (hist[i]) * norm;

      entropy -= pi * log2(pi);
    }
  }

  return entropy;
}
```

4.5.2 Matrix operations

Entropy computation can also be implemented with higher level matrix operations.

```
double
computeEntropy(cv::Mat m)
{
  assert(m.type() == CV_8UC1);

  //- compute histogram
  const int histSize = 256;
  const float range[] = { 0, 256 } ;
  cv::Mat hist;
  const float* histRange = { range };
  const bool uniform = true;
  const bool accumulate = false;

  cv::calcHist(&m, 1, 0, cv::Mat(), hist, 1, &histSize,
               &histRange, uniform, accumulate); //[A]
```

```
  //-compute entropy
  hist /= m.rows*m.cols; //[B]

  cv::Mat logP;
  cv::log(hist, logP); //[C]
  cv::divide(logP, log(2), logP);

  double entropy = -(cv::sum(hist.mul(logP)))[0]; //[D]

  return entropy;
}
```

A : Here we use *cv::calcHist()* to compute the histogram. See `http://docs.opencv.org/2.4.9/modules/imgproc/doc/histograms.html#calchist` for details about the parameters. The produced *hist* will be a matrix of size 256x1 (i.e., a column vector) and of type `CV_32F`.

B : Here the matrix is divided by a scalar. As *hist* is already of type `CV_32F`, there is no problem of truncation.

C : As there is no *log2* matrix operation in OpenCV, we compute $log(x)/log(2)$.

D : Here we compute $hist * logP$ with *mul* method of *cv::Mat*, and then sum all its elements with *cv::sum()*, and take the first element of the returned vector.