SEAL Team — Exploring Robotics — Computer Vision

Preparing & Improving
(I) Geometric transformations

Laurent Beaudoin & Loïca Avanthey

Épita

Foreword

During this seminar, we will familiarize ourselves with the geometric transformations that we have seen in the previous lesson and we will be particularly interested in borderline cases to properly master the manipulation of these tools.

1 Overthrowning!

In the seminar "*Image Manipulation*" we used the function `cv::flip` to reverse the frames of a video. We will now code our own mirror function.

EXERCICE 1 (*Flip*)

Write a function that allows to apply a horizontal reflection (the top becomes the bottom) on an image whose sensor was positioned upside down during the acquisition and test it on the image `GOPR1857_Nice_2018_09.JPG`.

Remember that we start from the final image: we apply the **reverse transformation**.

No need to process pixel by pixel, you can process **directly by line** (the `row()` and `copyTo()` methods will be useful).

QUESTION 1 (*Flip Matrix?*)

We will do the same thing again, but this time, to apply the horizontal reflection we will use an affine transformation F. Think about the content of the affine matrix (the form of an affine matrix F is recalled on page 3) to apply this transformation.

Be careful not to forget the **translation** to get back to (0,0) in the coordinate system.

EXERCICE 2 (*Flip Matrix!*)

Create your affine matrix and use the `cv::warpAffine` function to apply it to the initial image. Test it on the `GOPR1857_Nice_2018_09.JPG` image.

OpenCV takes care of passing in homogeneous coordinates, the affine matrix is therefore a **floating point matrix** 3×2. OpenCV also takes care of passing it to **inverse transform**.

To multiply two matrices in OpenCV, they must be in float format! As the result will be in floating numbers, don't forget to **convert the final image** to have channels in `unsigned char` to allow the **display**.

2 One sample, two samples,... lots of samples!

We will now see the scale transformations, starting with the simple cases, i.e. those for which the size of the final image is a divisor of the original size. To be sure you understand, you have to code it yourself, so we won't use the `cv:resize` function right away.

EXERCICE 3 (*Thumbnail*)

Write a function that creates an image four times smaller than an original image and test it on the `logo_seal.png` image (do not use the function `cv::warpAffine` here).

Do not forget: reverse transform!!!

EXERCICE 4 (*Pyramid*)

Write the function which allows you to create a series of images with a scale factor of 2, 4, 8 and 16 and test it on the `logo_seal.png` image.

QUESTION 2 (*Matrix thumbnail?*)

Again, the same result can be achieved through an affine transformation. Think about the content of the affine matrix to divide the size by 4.

EXERCICE 5 (*Matrix thumbnail!*)

Create your affine matrix and use the `cv::warpAffine` function to apply it to the initial image. Test with the image `logo_seal.png`.

Give the size of the final image as an argument to the function `cv::warpAffine`.

EXERCICE 6 (*Not a simple case: crash test*)

Test on the `logo_seal.png` image your function written in the exercise 3 by taking a non-integer scale factor, 2.3 for example (note that in this way the size of the final image is no longer a divisor of the original image size). Save the result. What do you observe? Is it consistent with what has been seen with the theory? Do you have an explanation ?

EXERCICE 7 (*Not a simple case: resize and choice of the type of interpolation*)

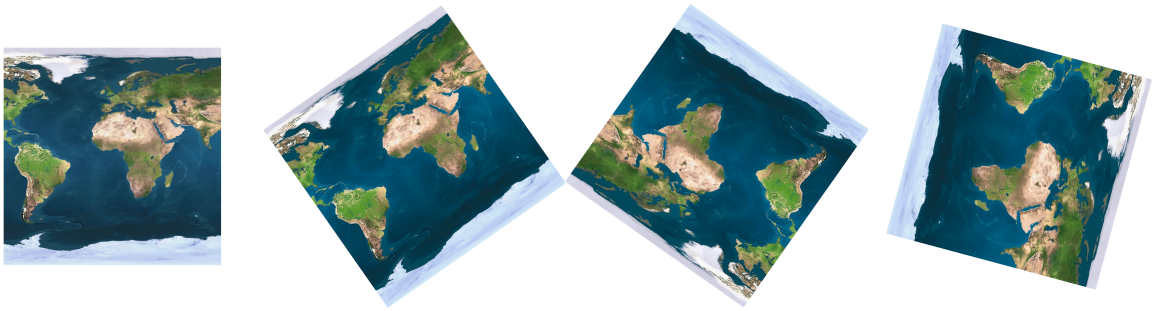
Now we authorize you to test the function `cv::resize` (same scale factor and same image as in exercise 6). Call it with nearest neighbor interpolation (`CV_INTER_NN`), with bilinear interpolation (`CV_INTER_LINEAR`) and with bicubic interpolation (`CV_INTER_CUBIC`). Save the results and compare them (as well as with that of exercise 6).

Do not forget to pass your **scale factor** on the **two axes** (s_x and s_y) as an argument in addition to the size of the final image so that the **interpolation** is **applied**.

After a **visual comparison** of the result images, **display the differences** (multiplied by a factor, so that they are visible) between the images two by two.

3 "You make my head spin": rotations!

In this exercise, to understand the interpolations well, we will code the whole process which makes it possible to rotate the image in the general case (arbitrary center of rotation) without using the ready-made functions of openCV.



To begin, we will complete the theoretical results we have seen: paper / pencil / brain sequence! A general rotation F of an angle α and centered on the point C of coordinates (C_x, C_y) with a scale factor s is written like the product (in homogeneous coordinates):

$$F = \begin{pmatrix} s\cos(\alpha) & s\sin(\alpha) & T_x \\ -s\sin(\alpha) & s\cos(\alpha) & T_y \\ 0 & 0 & 1 \end{pmatrix} = T \times S \times R = \begin{pmatrix} 1 & 0 & T_x \\ 0 & 1 & T_y \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

QUESTION 3 (*Reverse transform*)

Remembering that $F^{-1} = R^{-1} \times S^{-1} \times T^{-1}$, calculate F^{-1} (*check with the result page 6).

To be able to implement the formula found previously, it only remains to calculate the T_x and T_y values of the vector T. In our case, as we saw in the theory, T is the correction that must be made to C_r — result of the central rotation on the point $C(C_x, C_y)$ — so that it returns in (C_x, C_y).

QUESTION 4 (*Translation*)

Calculate $C_r = R * C$ then $T = \overrightarrow{C_r C}$.

Ok, you can write F and F^{-1} explicitly! In what follows, we will take $s = 1$.
Now it's coding time!!!

EXERCICE 8 (*Preliminaries*)

Write a program that takes in parameters a rotation angle in degrees and the name of an image, converts the angle into radians and loads the color original image and displays it.

EXERCICE 9 (*Preliminaries – continued*)

 Copy this image in the center of two images `rotateNN` and `rotateBIL` previously filled with 0 and which will contain the results of your rotation with respectively a nearest neighbor interpolation and a bilinear interpolation.

 The **width** of these two **new images** is equal to $\sqrt{w^2 + h^2}$ where w and h are the width and height of your original image.

 To copy an original image at a (x0, y0) position in a transformed image use: `original.copyTo(transformed(cv::Rect(x0, y0, width, height)));`

EXERCICE 10 (*Nearest neighbor interpolation*)

 Add a function that calculates the α angle rotation of the previous images using the nearest neighbor interpolation. For that :

a) We start by calculating the matrix F^{-1}.

b) Then for each pixel P of the resulting image, the position in the starting image is calculated $(F^{-1} \times P')$.

c) If it is a possible position (i.e. in the starting image), then we calculate the value at this point using the nearest neighbor interpolation on the 3 BGR planes and we store all the results in a temporary image.

d) Once all the pixels have been seen, display the result image.

 **Reminder:** for the nearest neighbor interpolation, you must calculate the integer closest to the decimal value obtained.

EXERCICE 11 (*Bilinear interpolation*)

 Add a function that calculates the α angle rotation of the previous images using bilinear interpolation. For that :

a) We start by calculating the matrix F^{-1}.

b) Then for each pixel P' of the resulting image, we calculate the position in the starting image $(F^{-1} \times P')$.

c) If it is a possible position (i.e. in the starting image), then we calculate the value at this point using bilinear interpolation on the 3 BGR channels and we store all the results in a temporary image.

d) Once all the pixels have been seen, display the result image.

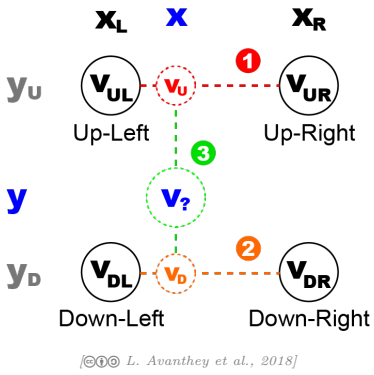
 **Reminder:** for bilinear interpolation, the algorithm is recalled below.

Reminder of the bilinear interpolation algorithm:

1. Calculation of the coefficients a and b from (x_1, y_1) and (x_2, y_2):

$$\begin{cases} a = y_2 - y_1 \\ b = y_1 x_2 - y_2 x_1 \end{cases}$$

2. Calculation of the radiometric value of (x, y):



1) Interpolation along the x axis: **up**
   - Calculate a_u and b_u from (x_L, v_{UL}) and (x_R, v_{UR})
   - Calculate the interpolated value of x such that $v_u(x) = a_u x + b_u$
2) Interpolation along the x axis: **down**
   - Calculate a_d and b_d from (x_L, v_{DL}) and (x_R, v_{DR})
   - Calculate the interpolated value of x such that $v_d(x) = a_d x + b_d$
3) Interpolation along the y axis: **final**
   - Calculate a and b from (y_u, v_u) and (y_d, v_d)
   - Calculate the interpolated value of y such that $v(y) = ay + b$

$\Rightarrow$ $v(x, y) \Leftrightarrow v(y)$

QUESTION 5 (*Analysis*)

What can you conclude by comparing your two result images (nearest neighbor and bilinear)?

3.1 Let's dig a little deeper

At the end of the previous exercise, it is not easy to make a visual difference between the results of the different interpolations. Also, we will in this exercise push the interpolators to their limits. For this, we will use the OpenCV functions which are more optimized than those which we coded in the previous exercise. The objective is to rotate an image 360 degrees but in steps of 1 degree to accumulate the errors of the different interpolations.

EXERCICE 12 (*Cumulative rotations: nearest neighbor interpolation*)

Write a program that:

a) Loads (in color) an image passed as a parameter.

b) Copy this image in the center of a `rotateNN` image previously filled with 0 and which will contain the results of your rotation with an interpolation to the nearest neighbor.

The **width** of this **new image** is equal to $\sqrt{w^2 + h^2}$ where w and h are the width and height of your original image.

c) Calculates the 1 degree rotation matrix using the `cv::getRotationMatrix2D` function.

d) Until the cumulative 360 degree rotation is reached:

- Apply the 1 degree rotation with a nearest neighbor interpolation using the `cv::warpAffine` function.
- Display the result.

You get an "animation" of the cumulative rotations.

EXERCICE 13 (*Cumulative rotations: bilinear interpolation*)

 Complete your previous program with the following:

a) Copy the original image in the center of an `rotateBIL` image previously filled with 0 and which will contain the results of your rotation with a bilinear interpolation (same size as for the `rotateNN` image).Copier l'image originale au centre d'une image `rotateBIL` préalablement remplie de 0 et qui contiendra les résultats de votre rotation avec une interpolation bilinéaire (même taille que pour l'image `rotateNN`).

b) In the loop of cumulative rotations:

- Apply the 1 degree rotation with a bilinear interpolation using the `cv::warpAffine` function.
- Display the result.

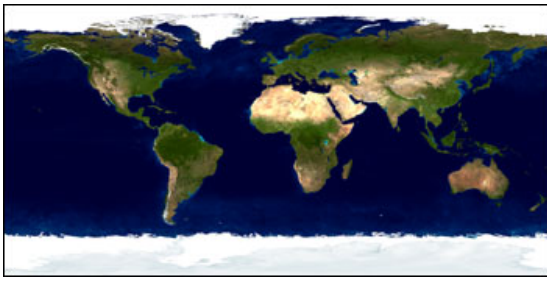
QUESTION 6 (*Analysis*)

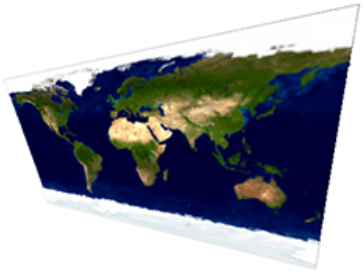
 Analyze the results obtained on the cumulative rotations with the two interpolators. What are your conclusions?

*At question 3 page 3, you are supposed to get: $F^{-1} = \begin{pmatrix} \frac{\cos(\alpha)}{s} & -\frac{\sin(\alpha)}{s} & \frac{T_x \cos(\alpha) - T_y \sin(\alpha)}{s} \\ \frac{\sin(\alpha)}{s} & \frac{\cos(\alpha)}{s} & \frac{T_x \sin(\alpha) + T_y \cos(\alpha)}{s} \\ 0 & 0 & 1 \end{pmatrix}$

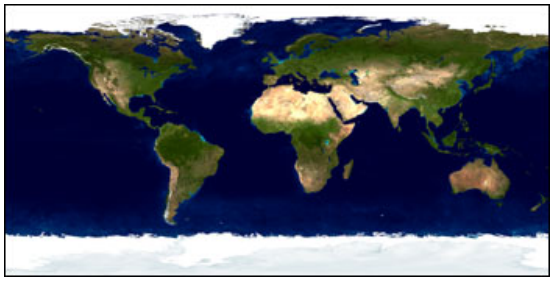
4 Have a good perspective

In this exercise, we will make a perspective transformation, that is to say a transformation that still respects straight lines but no longer the constraints of parallelism (affine) or isometry. In other words, this type of transformation changes a rectangle into any quadrilateral as in the following example.





The aim of the exercise is to project the image on the left onto one of the paintings in the image on the right:



Mathematically, it is necessary to calculate the homography matrix (3×3) which characterizes the transformation we want to applied. For this, we have seen in theory that it suffices to know the correspondences of 4 points (before and after the transformation). In our case, we will take the coordinates of the 4 corners of the image on the left with the coordinates of the 4 corners of a painting in the image on the right. Then, we will use the `cv::getPerspective` function of OpenCV which calculates the matrix and the `cv::warpPerspective` function to apply the perspective.

EXERCICE 14 (*Map room*)

Write a program that:

a) Load in color the `map.jpg` and `room.jpg` images.

b) Create two tables of 4 `cv::Point2f` which respectively contain the 4 corners of the image `map.jpg` – for example in order (0, 0), (0, width-1), (height-1, width-1) and (height-1, 0) – and the coordinates of the 4 corners of one of the paintings in the image `room.jpg` in the same order – for example (59, 102), (115, 115), (115, 164) and (59, 158).

c) Calculates the perspective matrix using the `cv::getPerspectiveTransform` function.

d) Calculates the perspective projection of `map.jpg` using the `cv::warpPerspective` function. We will put at 0 the pixels of the background of the image.

e) Copy only the pixels which correspond to the projection of `map` from the previous image into the `room.jpg` image with the `copyTo` function and the use of a mask.

To calculate the mask image where the useful pixels are at 255 and the others at 0, we convert the perspective image to gray levels and then we threshold it at 1.

For the record, the `room.jpg` image is the map room of the Palazzio Vecchio in Florence. Readers of Dan Brown's novel *Inferno* will appreciate ;-).

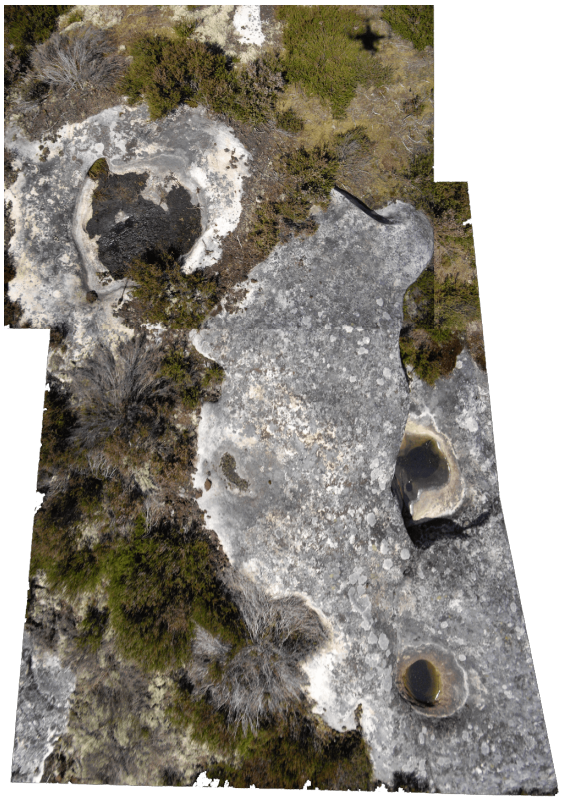
5 First mosaic: geometric part

We are now going to create our first little mosaic! For this we will use three images taken during a drone flight over a study area at Fontainebleau and we will readjust them. Compared to the previous exercise, this implies some small changes: the corresponding points to choose are less obvious and the overlap area being almost zero between the two most spatially distant images, it will be necessary to make a successive registration.

APN1

APN2

APN4



QUESTION 7 (*Corresponding points*)

First, to be able to calculate the homography matrix which links one image with another, you must find the coordinates of 4 corresponding points (you can use the Gimp software for this):

a) Note the coordinates of 4 corresponding points between the `vol4_apn1_IMGP0008` image and the `vol4_apn2_IMGP0008` image.

b) Note the coordinates of 4 corresponding points between the `vol4_apn2_IMGP0008` image and the `vol4_apn4_IMGP0008` image.

The more the points you choose are widely spread, the better the calculated matrix

The quality of the registration will depend on the precision of the coordinates of corresponding points

EXERCICE 15 (*Homographies*)

Write a program that:

a) Open in color the three images `vol4_apn1_IMGP0008.JPG`, `vol4_apn2_IMGP0008.JPG` and `vol4_apn4_IMGP0008.JPG`.

b) Create two arrays of 4 `cv::Point2f` which respectively contain the coordinates of the points in the `vol4_apn1_IMGP0008.JPG` image and the coordinates of the corresponding points in the `vol4_apn2_IMGP0008.JPG` image.

c) Create two other tables of 4 `cv::Point2f` which respectively contain the coordinates of the points in the `vol4_apn2_IMGP0008.JPG` image and the coordinates of the corresponding points in the `vol4_apn4_IMGP0008.JPG` image.

d) Compute the 2 homographic matrices (`cv::getPerspectiveTransform`) for these pairs of images.

EXERCICE 16 (*Registration*)

Complete your program so that it:

a) Create a `mosaic` image which will contain the final mosaic and copy (`copyTo`) to the top left the `vol4_apn1_IMGP0008.JPG` image.

b) In a temporary image of the size of the `mosaic` image, applies the transformation of the `vol4_apn2_IMGP0008.JPG` image in the frame of reference of the `vol4_apn1_IMGP0008.JPG` image thanks to the first calculated homography matrix and the `cv::warpPerspective` function, and copies the result obtained in the `mosaic` image.

c) In a temporary image of the size of the `mosaic` image, applies the transformation of the `vol4_apn4_IMGP0008.JPG` image in the frame of reference of the `vol4_apn2_IMGP0008.JPG` image thanks to the second calculated homography matrix and the `cv::warpPerspective` function, then applies the transformation of the result obtained in the frame of reference of the `vol4_apn1_IMGP0008.JPG` image thanks to the first matrix and copy the final result into the `mosaic` image.

d) Save the `mosaic` image (`cv::imwrite`) and display it on the screen after reducing its size so that you can view it in full (`cv::resize`).

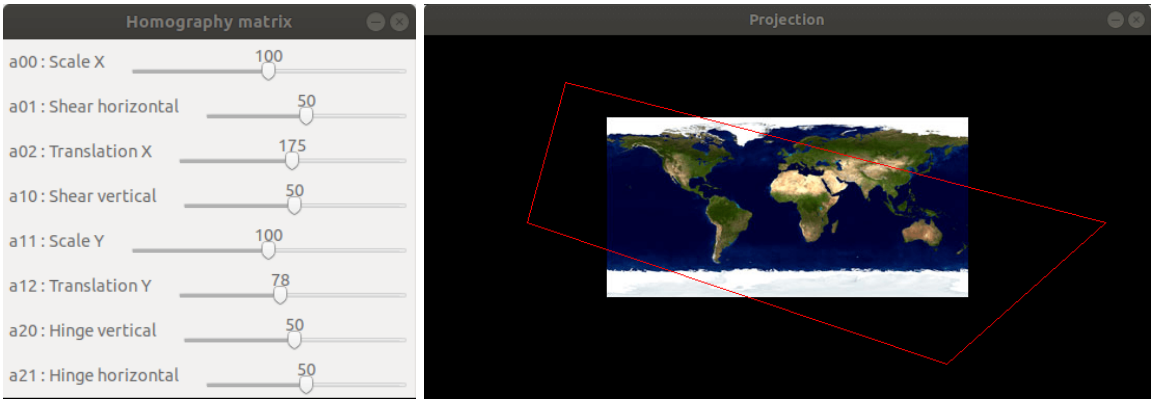
For the **size** of the `mosaic` image, you can take 8000 × 6000 px (and for displaying on screen, decrease it by a factor of 10: 800 × 600 px)

And that's it for this **first mosaic**! We will see in the theoretical part "*Detecting*" that it will be possible to **automatically find the corresponding points** between two images and that we can also use **statistical tools** to **improve the precision of the homography matrix** calculated to compensate for the small errors on the location of the points. The **choice** of the **frame of reference** is also important (here, we have chosen by default that of the first image, but it's rarely the best choice).

The result we get is **far from a finished product**: we can see the connections! We have indeed only applied the geometric transformations, there remains all the **fusion part** which mainly uses **radiometric correction tools**.

6 Control the matrix

The objective of this exercise is to understand how each term in the homography matrix constrains the distortion of an image. For that, we will make a summary GUI to vary these coefficients dynamically and see in real time the induced deformation. Mastery is assessed from the ability to constrain the image to place it in a target area.



6.1 Draw the target area

To materialize the target area, that is to say the area that the source image should occupy in the destination image after using the homography matrix, we will use the `cv::polylines` function from `OpenCV`.

```
void cv::polylines ( cv::Mat img, const Point* const * pts, const int* npts, int ncontours, bool
     isClosed, const Scalar& color, int thickness = 1, int lineType = LINE_8, int shift = 0 )
```

whose parameters are:

- `img` the image where the drawing is made,
- `ncontours` is the number of different contours (or polygons) that the function should draw. In our case, there is only one outline which is the target area,
- `npts` is an array of integers which indicates how many points are made up of each contour. In our case, this table reduces to a box which contains the value 4 for 4 points which are the 4 corners of the target area,
- `pts` is an array of points table which contains all the points of the contours. For example, `pts[i][j]` is the j-th point of the i-th contour.
- `isClosed` specifies if the contour must be closed, so if we must connect the last point with the first,
- `color` to choose the color,
- `thickness` to choose the thickness,
- `lineType` to choose if we draw in the neighborhood 4 or 8 connectivity,
- `shif` to allow non-integer point coordinates.

There are other drawing functions for drawing simple lines, rectangles, circles, ellipses, etc.

The following code illustrates drawing of the target area:

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

int main(int argc, char** argv){

   cv::Mat projection = cv::Mat::zeros(350, 700, CV_8UC3);

   const cv::Point cornersTargetArea[4] = { cv::Point(136, 45), cv::Point(656,180),
                        cv::Point(503,316), cv::Point(99, 180)};
   const cv::Point* arrayOfPolygons[1] = {cornersTargetArea}; //here only one polygon
   const int nbPointsByPolygon[1] = {4};

   cv::polylines(projection, arrayOfPolygons, nbPointsByPolygon, 1, true, cv::Scalar(0, 0, 255),
        1, 8, 0);
   cv::imshow("Projection", projection);

   cv::waitKey(0);
   cv::destroyAllWindows();

   return 0;
}
```

EXERCICE 17 (*Target area*)

Copy the previous code, compile it, and run it to test it.

6.2 Trackbars, or dynamic modification of parameters

`OpenCV` is not designed to make GUIs but a special effort is made to easily interface with other libraries dedicated to this like `Qt` for example. However, you can easily use a useful tool, the `trackBar`, included natively in `OpenCV`, to vary a parameter during the execution of a program.

The key steps in using a `trackbar` are creating the trackbar and using a callback function.

The creation of a `trackBar` is done via the `cv::createTrackbar` function.

```
int createTrackbar(const string& trackbarname, const string& winname, int* value, int count,
    TrackbarCallback onChange=0, void* userdata=0)
```

whose parameters are:

- `trackbarname` is the name of the `trackBar` and therefore its identifier for the rest of the code,
- `winname` the name of the window that contains the `trackbar`,
- `value` is a pointer to an integer to retrieve the current value of the cursor from the `trackbar`,
- `count` which is the maximum value of the `trackbar`,
- `onChange` is the name of the callback function that is executed each time the cursor is moved to the `trackbar`,
- `userdata` to pass any additional parameters to the previous callback function.

The callback function has as prototype: `void Foo(int pos,void* userdata)` with `pos` which is the cursor position of the `trackbar` when the function is called and `userdata` any additional parameters. This function will be called systematically at each variation of the cursor.

The following code illustrates the interest of the `trackbar` on one of the coefficients of the homography matrix (displacement in colones).

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

#define NB_LINES_TRANSFORMED 350
#define NB_COLS_TRANSFORMED 700
#define NB_LINES_MAP 175
#define NB_COLS_MAP 350

cv::Mat map; //map image
cv::Mat projection = cv::Mat::zeros(NB_LINES_TRANSFORMED, NB_COLS_TRANSFORMED, CV_8UC3);
cv::Point2f tabMapPoints[4] = { //array of 4 corners of the map image
      cv::Point2f(0., 0.),
      cv::Point2f( NB_COLS_MAP - 1., 0.),
      cv::Point2f( NB_COLS_MAP -1., NB_LINES_MAP - 1.),
      cv::Point2f(0., NB_LINES_MAP -1.)};
cv::Mat homographyMat = cv::Mat::eye(3, 3, CV_32FC1); //homographic matrix
cv::Point2f tabProjectionPoints[] = { //array of projected map corners in the final image
      cv::Point2f(0., 0.),
      cv::Point2f( NB_COLS_MAP - 1., 0.),
      cv::Point2f( NB_COLS_MAP -1., NB_LINES_MAP - 1.),
      cv::Point2f(0., NB_LINES_MAP -1.)};

int Tx; //Global variables for trackbars
const int tb02Max = NB_COLS_MAP; // Tx value Max

void plotProjection(){//ploting the results
   cv::warpPerspective(map, projection, homographyMat, projection.size(), cv::INTER_LINEAR , cv
       ::BORDER_CONSTANT, cvScalar(0));
   cv::imshow("Projection", projection);
   std::cerr << homographyMat << std::endl;
}

static void tb02( int, void* ){ //callback trackbar function
   homographyMat.at<float>(0,2) = Tx;
   plotProjection();
}

int main(int argc, char** argv){

   map = cv::imread("map.jpg", cv::IMREAD_COLOR);
   if ( map.empty() ) {
      std::cerr << "Impossible to load the map image " << std::endl;
      exit(-1);
   }
   cv::namedWindow("Homography matrix", cv::WINDOW_AUTOSIZE); //windows with trackbars
   cv::Mat homographyWindow = cv::Mat::zeros(1, 350, CV_8UC1);
   cv::createTrackbar("a02 : Translation X" , "Homography matrix", &Tx, tb02Max, tb02);
   cv::setTrackbarPos("a02 : Translation X" , "Homography matrix", 175);//map is centered
   tb02( Tx, 0 );
   cv::imshow( "Homography matrix", homographyWindow );

   cv::waitKey(0);
   cv::destroyAllWindows();

   return 0;
}
```

EXERCICE 18 (*One `trackbar`*)

 Complete your program by adding this code to display a trackbar in a window other than the one where you drew the target area. Compile and test the execution.

6.3 Playing with the homography matrix

EXERCICE 19 (*A `trackbar` for each coefficient*)

 Complete your program to obtain a window now containing 8 `trackbars` in order to be able to dynamically modify the coefficients of the homography matrix (therefore all the coefficients except the lower right diagonal).

 For the conversion of the value of the cursor *val* into value for the matrix we will take as rule:

- $(val - 50)/50$ for the a_{00} and a_{11} coefficients (initialization at 100 and maximum value at 200) as well as for a_{01} and a_{10} (initialization at 50 and maximum value at 100).
- directly *val* for both coefficients a_{02} (initialization at 175 and maximum value at 350) and a_{12} (initialization at 78 and maximum value at 175).
- $(val - 50)/5000$ for the a_{20} and a_{21} coefficients (initialization at 50 for a maximum value at 100).

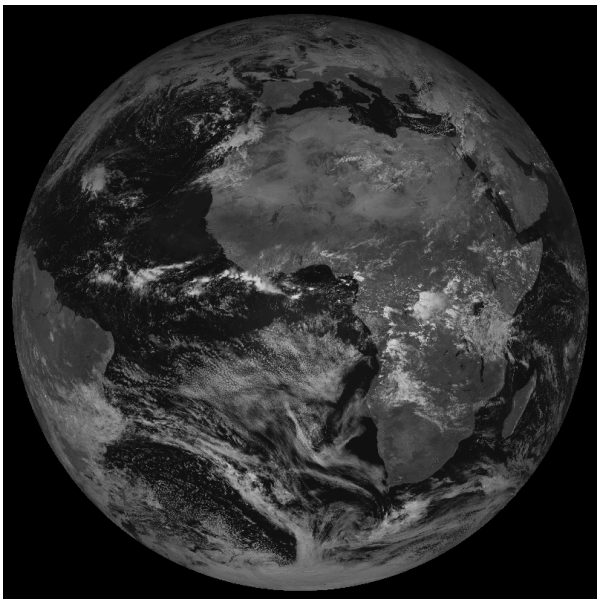
EXERCICE 20 (*Reprojected image*)

 In the other window, where the target area is drawn, add (in the background) the reprojected image.

EXERCICE 21 (*Mastering the coefficients*)

 Compile, execute and find the right coefficients of the homography matrix to project the image in the target area: use your cursors, ready, play!!!

7 Mercator power!!!




In this exercise, we will make a somewhat special geometric transformation which will allow us to project an image taken by the meteorological satellite MSG (on the left) into an image representing the planisphere in Mercator projection (on the right).

7.1 Preparatory functions

Before getting into the very heart of projection algorithms, it is necessary to code a few very useful preparatory functions for the main part.

EXERCICE 22 (*Degrees to radians*)

 Code the `double deg2Radians(double degrees)` function which returns the value in radians of an angle passed in degrees.

EXERCICE 23 (*Radians to degrees*)

 Code the `double rad2Degrees(double radians)` function which returns the value in degrees of an angle passed in radians.

7.2 Conversion functions between image and geographic coordinates

The reference document for performing these operations is the technical document [EUM13] and more precisely the information on pages 24 to 32.

EXERCICE 24 (*Geographic coordinates to image coordinates*)

Code the `int geo2Pix(double lat, double lon, int& line, int& column)` function which allows to pass from `(lat, lon)` geographic coordinates expressed in degrees to `(line, column)` coordinates in the full resolution of the MSG image (3712×3712). In details:

a) Calculate the values of the variables `c_lat`, `rl`, `r1`, `r2`, `r3` and `r3` on page 28 of the document reference.

b) Check that the P point at the geographic coordinates is clearly visible by the satellite (this step is not in the technical documentation). For that, it is enough to check that the angle between the local normal at the P point and the direction of the satellite is smaller than the right angle. In other words, it is enough to verify that the dot product of these vectors is positive and therefore concretely it is necessary to calculate the value of:

```
double dotprod = r1 * (rl * cos(c_lat) * cos(lon - SUB_LON)) - r2 * r2
                 - r3 * r3 * pow(R_EQ/R_POL, 2);
```

with:

- `SUB_LON = 0.0` the longitude of the satellite,
- `R_EQ = 6378.1370` the terrestrial radius in km at the equator in the WGS84 ellipsoid model,
- `R_POL = 6367,7523` the terrestrial radius in km at the pole.

$\Rightarrow$ If `dotprod` is null or negative, we return -1 as the status value.

c) Calculate the values of `x` and `y` with the formulas on page 28 of the reference document.

d) Finally, calculate the values of `line` and `column` from the system of equations in 4.4.4 on page 32 of the reference document with:

- `CFAC = -781648343`
- `LFAC = -781648343`
- `COFF = 1856`
- `LOFF = 1856`

The return value is the status of the function: if it is 0, the calculations have a meaning, if it is -1, then no.

EXERCICE 25 (*Geographic coordinates to image coordinates*)

Code the `int pix2Geo (int line, int column, double& lat, double& lon)` function which allows to pass from `(line, column)` coordinates in the MSG image in full resolution (3712 × 3712) to `(lat, lon)` geographic coordinates expressed in degrees. In details:Codez la fonction `int pix2Geo(int line, int column, double& lat, double& lon)` qui permet de passer des coordonnées `(line, column)` dans l'image MSG en pleine résolution (3712×3712) en coordonnées géographiques `(lat, lon)` exprimées en degrés. Dans le détail :

a) Start by writing a function that calculates (y, x) knowing (l, c). To do this, reverse the system of equations in 4.4.4 page 32 of the reference document. Take these values for the constants:

- `CFAC = -781648343`
- `LFAC = -781648343`
- `COFF = 1856`
- `LOFF = 1856`

b) Let us set $s_d = \sqrt{s_a}$ with `sd` defined on page 29 of the reference document. If the value of s_a is negative or zero, it means that you are beyond the limb and therefore you will return -1 as status.

c) Calculate the values of the variables `sd`, `sn`, `s1`, `s2`, `s3` and `sxy` on page 29 of the reference document.

d) Finally calculate the values in degrees of `(lat, lon)` on page 29 of the reference document.

The return value is the status of the function: if it is 0, the calculations have a meaning, if it is -1, then no.

7.3 The transformation, in images!

You have 3 images at your disposal to test your program:

- `msg.png` : the full resolution image (3712 × 3712) taken by the MSG11 satellite.
- `mercator.jpg` : extract from a planisphere in Mercator projection, of size (800 × 800), centered on the geographic coordinates (0,0) and having for range of latitudes and longitudes -90 ° to +90 ° .
- `msg_contours.jpg` : full resolution image (3712 × 3712) on which are drawn the contours of the continents and the crossings in latitude and longitude in steps of 10 ° .

EXERCICE 26 (*Loads*)

Start by loading each image in its respective matrix: `msgFullRes`, `mercator` and `contours`.

EXERCICE 27 (*Mercator from MSG (reverse transformation)*)

 Code the `void mercator2Msg(cv::Mat msgFullRes, cv::Mat mercator, cv::Mat contours, cv::Mat& mercator2Msg, cv::Mat& mercator2MsgVerif)` function which, for all the pixels in the `mercator` image look for its correspondent in the `msgFullRes` image.

 The `mercator2Msg` and `mercator2MsgVerif` matrices will respectively contain the **result of the projection** of `msgFullRes` and `contours` in the projection of Mercator.

In more detail:

a) Clone `mercator` on `mercator2Msg` and `mercator2MsgVerif`.

b) For each pixel `(lineM, colM)` of `mercator2Msg` and `mercator2MsgVerif`:Pour chaque pixel `(lineM, colM)` de `mercator2Msg` et `mercator2MsgVerif` :

1. Calculate the geographic `(lat, lon)` coordinates corresponding to the `(lineM, colM)` pixel.
2. Calculate the `(lineMsg, colMsg)` coordinates corresponding to `(lat, lon)` with the `geo2Pix` function and retrieve the status. Continue if the status is different from -1.
3. To verify that the projection went well, we will test our result with the contours: if the `(lineMsg, colMsg)` pixel of the `contours` image is not zero, the `(lineM, colM)` pixel of the image `mercator2MsgVerif` is worth the `(lineMsg, colMsg)` pixel of `contours`.
4. Check that the pixel is on Earth. To do this, calculate its distance from the center of the image and continue only if it is smaller than the half-width of the image minus 60 pixels of margin. If this is the case, the `(lineM, colM)` pixel of `mercator2Msg` is equal to the `(lineMsg, colMsg)` pixel of `msgFullRes`.

EXERCICE 28 (*Verification*)

 Display the images `mercator2MsgVerif` and `mercator2Msg`.

7.4 Let's dig a little deeper

We saw in the theory that it was more judicious to use the inverse transformation (we start from the resulting image to find the corresponding values in the starting image), because it is more efficient. But is this the only reason? Let us test and look at what the result of the direct Mercator transformation gives (that is to say that we start this time from the starting image to fill the resulting image).

EXERCICE 29 (*MSG to Mercator (direct transformation)*)

Code the `void msg2Mercator(const cv::Mat msgFullRes, const cv::Mat mercator, const cv::Mat contours, cv::Mat& msg2Mercator, cv::Mat& msg2MercatorVerif)` function.

The `msg2Mercator` and `msg2MercatorVerif` matrices will contain respectively the result of the projection of `msgFullRes` and `contours` in the projection of Mercator.

In more detail:

a) Clone `mercator` on `msg2Mercator` and `msg2MercatorVerif`.

b) Calculate the `step` subsampling coefficient as the ratio between the width of `msgFullRes` by that of `msg2Mercator`.

c) For all the `(lineMsg, colMsg)` pixels of the `msgFullRes` image (remembering that to pass from one line to another or from one column to another you must increment the counter not by 1 but by `step` because it is the subsampled image that we project):

1. Check that the pixel is on Earth as in the previous function.
2. Calculate the geographic coordinates using the `pix2Geo` function and retrieve the status.
3. If the status is not equal to -1, calculate the `(lineM, colM)` coordinates in the `mercator` image which correspond to the `(lat, lon)` geographical coordinates that we have just calculated.
4. To verify that the projection went well, we will test our result by displaying the contours (i.e. non-zero pixels) of the `contours` image in the `mercator` image. For this, if the `(lineMsg, colMsg)` pixel in the `contours` image is not zero, then the `(lineM, colM)` pixel of `msg2MercatorVerif` is worth the value of `(lineMsg, colMsg)` pixel in the `contours` image.
5. The `(lineM, colM)` pixel of `msg2Mercator` is worth the `(lineMsg, colMsg)` pixel of `msgFullRes`.

EXERCICE 30 (*Verification*)

Display the `msg2MercatorVerif` and `msg2Mercator` images and compare with the results obtained with the inverse transformation. What do you conclude?

References

[EUM13] EUMETSAT. LRIT/HRIT Global Specification. Technical report, Coordination Group for Meteorological Satellites, 2013. 15