Trifocal Tensor

Cédric Bidaud
Aurélien Greffard

January 6, 2013

Project state sum-up

- Asked working implemented elements :
  - help
  - options in command line
  - points lists management
  - images display
  - image loading from command line
  - list loading from command line
  - calculation of the tensor
  - transfer (works, but the approximation is not negligible)
- Asked but not working implemented elements :
  - Transfer optimisation. The points given by the transfer function are in the right zone, often very close to their theorical location, but sometimes very far from it.
- Asked but not implemented elements : none
- Not (explicitly) asked and workink implemented elements : error messages
- Not asked and not working (or not implemented) elements : none

Report written with LaTeX.

Chapter 1

Analysis

1.1 Tensor calculation

From the equation extracted and given in the subject :

$$x^k(x'^i x''^3 T_k^{3l} - x'^3 x''^3 T_k^{il} - x'^i x''^l T_k^{33} + x'^3 x''^l T_k^{i3}) = 0^{il}(1)$$

which actually corresponds to a system of several equations. In order to calculate the vector t, we have to rewrite this system as a matrix of type $At = 0$, which means to build this matrix A. Each element of A corresponds to the coefficients in the equations which are just before elements of t. Different loops will allow us to insert correct coefficients in their right places in the matrix.

First step consists in initializing elements of A to 0. Indeed, in each equation of the system, only 12 elements of t on 27 appear, which means others have a null coefficient. Eigen provides some tools to directly initialize matrix A to 0.

Second step consists in filling matrix A. When examining equation (1), we understand this filling will be managed with 4 loops : 3 loops on p, i and l, which select the right line of the matrix, because they are variables which create new equations, then a loop on k, which selects the right column, according to the t coordinate we are working on.

In the subject, i and l vary from 1 to 2, p from 1 to the number of match points, and k from 1 to 3. As lines and columns of our matrices start from 0, we will have i and l varying from 0 to 1, k from 0 to 2 and p from 0 to the number of points -1.

Coefficients we will insert in A will depend on coordinates of x, x' et x" points. Now, these coordinates are saved in our matching points lists, which means in a matrix form. We have to be able to get them back too. For instance, if we consider that points of the first, second and third image are respectively saved in matrices list1, list2 and list3, then k coordinates of the matching point p of the first image is list1(p, k).

Last step consists in calculating t with SVD method, provided by Eigen. Matrix A is decomposed in a 3 matrices product U*D*V, and we can save the tensor values, which correspond to the last column of V (column 26 in our case).

This way, still from equation (1), we get to the following pseudocode :

Algorithm 1 Tensor calculation

```
Initiate A to 0
for $p = 0$ to $nbPoints - 1$ do
    for $i = 0$ to 1 do
        for $l = 0$ to 1 do
            for $k = 0$ to 2 do
                A(4*p + 2*i + l, 9*2 + 3*l + k) += list1(p,k) * list2(p,i) * list3(p,2)

                A(4*p + 2*i + l, 9*i + 3*l + k) -= list1(p,k) * list2(p,2) * list3(p,2)

                A(4*p + 2*i + l, 9*2 + 3*2 + k) -= list1(p,k) * list2(p,i) * list3(p,l)

                A(4*p + 2*i + l, 9*i + 3*2 + k) += list1(p,k) * list2(p,2) * list3(p,l)
            end for
        end for
    end for
end for
A = UDV {SVD decomposition}
$tensor \leftarrow V.col(26)$
```

1.2 Transfer

Now that t is known, we want to find a point from the two others. The principle is the same : first rewrite the system as a matrix, and then fill a matrix B.

Before starting and calculating a 3 coordinates vector just like in the last step, it could be a good thing to notice that the 3rd coordinate of the point we are looking for is known and is equal to 1, as it is its homogeneous coordinate. We only have 2 unknown coordinates then and this time, we have to solve a matrix equation of the form $Bv = b$, with v the 2 coordinates vector.

We should rewrite equation (1) with this last point in mind. To be able to find a point on any image from the two others, we have to examine this equation for each of the 3 cases.

If we are looking for the point x', then x'3 is equal to 1, which leads us to the following equation :

$$x^k x'^i x''^3 T_k^{3l} - x^k x'^i x''^l T_k^{33} = x^k x''^3 T_k^{il} - x^k x''^l T_k^{i3}$$

If we are looking for the point x", then x"3 is equal to 1, which leads us to the following equation :

$$x^k x'^3 x''^l T_k^{i3} - x^k x'^i x''^l T_k^{33} = x^k x'^3 T_k^{il} - x^k x'^i T_k^{3l}$$

Finally, if we are looking for the point x, then x3 is equal to 1, which leads us to the following equations :

if k=3 :

$$x'^i x''^3 T_k^{3l} - x'^3 x''^3 T_k^{il} - x'^i x''^l T_k^{33} + x'^3 x''^l T_k^{i3} = 0^{il}$$

else :

$$x^k (x'^i x''^3 T_k^{3l} - x'^3 x''^3 T_k^{il} - x'^i x''^l T_k^{33} + x'^3 x''^l T_k^{i3}) = 0^{il}$$

This time, we have to fill the matrix B and the vector b. To avoid confusions, we will use MatB and Vecb instead of B and b.

Algorithm 2 Transfer

```
Require: x1 and x2 are known
  Initiate MatB to 0
  Initiate Vecb to 0
  for $i = 0$ to 1 do
    for $l = 0$ to 1 do
      for $k = 0$ to 2 do
        if we search the point x then
          factor = x1(i) * x2(2) * tensor(2, l, k) - x1(2) * x2(2) * tensor(i, l, k) - x1(i) * x2(l) * tensor(2, 2, k) + x1(2) * x2(l) * tensor(i, 2, k)
          if k=2 then
            Vecb(2*i + l) -= factor
          else
            MatB(2*i + l, k) += factor
          end if
        else if we search the point x' then
          MatB(2*i + l, i) += x1(k) * ( x2(2) * tensor(2, l, k) - x2(i) * tensor(2, 2, k) )
          Vecb(2*i + l) -= x1(k) * ( x2(l) * tensor(i, 2, k) - x2(2) * tensor(i, l, k) )
        else if we search the point x" then
          MatB(2*i + l, l) += x1(k) * ( x2(2) * tensor(i, 2, k) - x2(i) * tensor(2, 2, k) )
          Vecb(2*i + l) -= x1(k) * ( x2(i) * tensor(2, l, k) - x2(2) * tensor(i, l, k) )
        end if
      end for
    end for
  end for
  MatB = UDV {SVD decomposition}
  $solution \leftarrow SVD.solve(Vecb)$
```

Chapter 2

The program

In this part, we will explain our objects, structures and functionalities choices.

2.1 Data structures

2.1.1 Tensor

The tensor is an object with a vector of n elements - Eigen::VectorXf - initialized with zeros and methods to easily acces an element - operator () surcharge, and of course fill it and get its elements. Once filled it can also calculate the transfered point from two points clicked on images.

The fill and transfer methods could have been part of the Tensor class, but it was clearer for us to set them apart.

2.1.2 Point lists

The matching points are saved in lists. The first two numbers are the x and y coordinates clicked on the image, the third is the homogeneous coordinate (1). To remain coherent, the points are saved in the same order as the one they are clicked on the images.

In the program, lists are matrices - Eigen MatrixXf. For each matrix corresponds an int that saves its rows count. It's usefull to add a point to a list : as we strangely didn't find any push method for Eigen matrices, we wrote ours. This implied to resize the matrix we wanted to push, that's why we decided to save this number for a practical reason - also usefull for some tests.

2.1.3 Others

To perform the transfer, we have to save the two clicked points and their corresponding images. We define two sets : the firts for the points, in which we stock clicked points, the second initialized with the three images. When an image is clicked, it is removed from this set, and the one which remains is the image where the transfered point must be written.

2.2 Algorithms

As algorithms used to calculate the tensor and the transfer function have already been explained in the first chapter, we will here focus on algorithms related to the good working of the program.

2.2.1 Options

One of the key features asked is the possibility to handle options entered in command line. As we never managed this kind of options, we had to find a solution to efficiently analyse arguments given to the program. We choose to convert argv arguments in an array of strings - std::string. It provided us the usefull strcmp method, which allows to compare a string to another.

Algorithm 3 Options

```
  for each argument do
    for each option do
      if argument corresponds to an option then
        execute the option
      end if
    end for
  end for
```

The list of the different options implemented can be found on the first page.

As the searches for lists or images arguments rely on the same algorithm, we will only describe the one for the images. We are located at the “Execute the option” point.

Algorithm 4 Search for images

```
Require: $externalImages = 0$, the number of loaded images
  if argument corresponds to an option (contains .jpg, .png or .gif) then
    if $externalImages = 3$ then
      too many images loaded, the first three are kept
    else
      load image (if possible) and increment $externalImages$
    end if
  end if
```

This algorithm assures us that the user won’t load too many images. To be sure that he doesn’t load less, we make another verification after browsing all the arguments : if the number of external images is less than 3, the program loads default images, with a message which indicates if there is no or not enough images loaded.

2.2.2 Program states

The program runs with three states :

FILL_LISTS is the standard state, where the user fills the matching points lists by clicking on the images. If the three lists are filled, the tensor calculation can be launched by pressing Enter.

TRANSFERT is the state the program enters if the tensor is successfully calculated. The user must now click two points on two different images to launch the calculation of the coordinates of the transfered point.

SOLUTION is the last state, where the program displays the transfered point according to the two points clicked in TRANSFERT state.

These three states alter the behaviour of input commands (mouse and keyboard) and are checked in the event management loop.

2.3 Optimisations

As we said in the sum-up at the begining of this report, the transfer function is not always accurate. We have several ideas to improve the precision, but didn't manage or didn't have the time to implement them. For example : `http://users.cecs.anu.edu.au/~hartley/Papers/tensor/journal/final/tensor3.pdf` page 9 section Normalization. We will quote here this report :

> Normalization. Before setting out to write and solve the equations, it is a very good idea to normalize the data by scaling and translating the points. The algorithm does not do well if all points are of the form (u1 , u2 , 1) in homogeneous coordinates with u1 and u2 very much larger than 1. A heuristic that works well is to translate the points in each image so that the centroid of all measured points is at the origin of the image coordinates, and then scaling so that the average distance of a point from the origin is 2 units. In this way the average point will be something like (1, 1, 1) in homogeneous coordinates, and each of the homogeneous coordinates will be approximately of equal weight. This transformation improves the condition of the matrix of equations, and leads to a much better solution. Despite the seemingly harmless nature of this transformation, this is an essential step in the algorithm.

Another way to improve our program would be to set up a real errors manager. For the moment, the program just displays an error message then crashes if a problem appears.