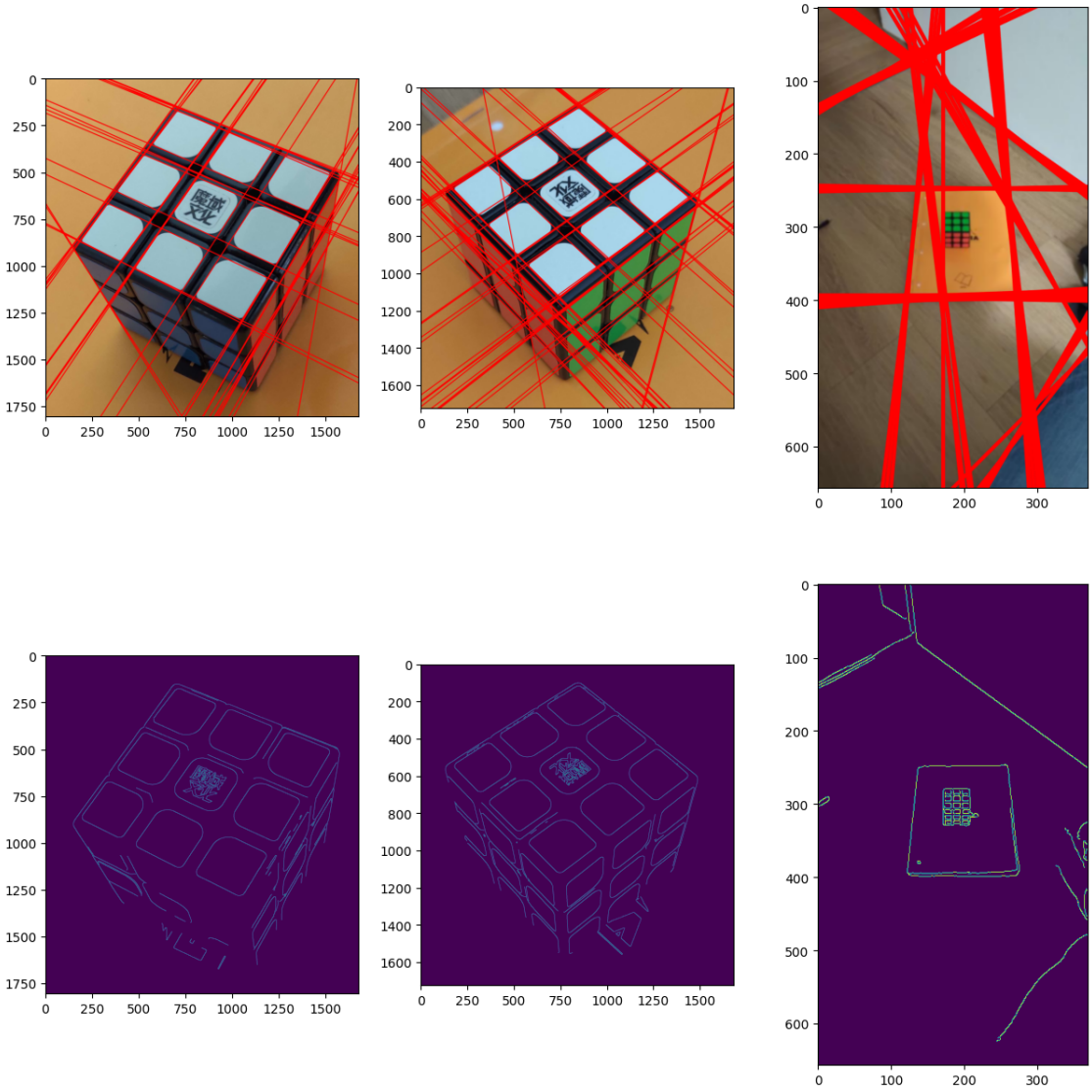
Tp2

December 20, 2023



1 A. Computing vanishing points: questions

1.1 1. The number of vanishing points

Between 0 and 3 finite vanishing points can be found from all possible images of a Rubik's cube. This is because our 3D world, once projected onto a 2D photography only has 3 degrees of freedom for the viewpoint: Left and right, up and down, forward and backward.

For the minimum, the least you can have is 0 - When you take a really zoomed in picture (inside a green square for exemple with no edge visible) because there will be no parallel lines. - When you take a perfect picture of a single face like a 2D square, because the lines on the Rubik's Cube will be parallel and will have the same depth (to the camera), the lines projected onto the image in 2D will be perfectly parallel and will never intersect to give vanishing points.

But for a more realistic picture where you can see only one whole face, there will be probably a rotation so, even if it will be a 2d square with only x and y axis, thoses axis parallel lines will converge to a single point for at least one axis event if it is really far, giving 1 vanishing points.

1.2 2 Is there a relation between number of visible faces and vanishing points?

Yes, there will be one vanishing point per axis transformed, with a maximum of three for a normal camera projection

1.3 3 Can you find 4 vanishing points in Rubik's cube images? If yes, demonstrate with an image.

No, we will never be able to find a 4th vanishing point because the photo takes a 3D image (our world) onto a 2D image, so there are only 3 axes to be deformed by the perspective which means that we will only be able to fid 3 vanishing points at most. That's only

If you change perspective to deform parallels (like a fish eye camera) you may be able to find more than 3 vanishing points, because line will become curved and will be able to intersect more than 3 times. this is not possible with a normal camera, but specialized fish eye cameras can do that and you will be able to find 4 vanishing points.

1.4 4. Can you find a configuration with at least one vanishing point outside the image?

Vanishing points can be outside of the image. Take the following picture for example:

```
x:  -1117.8672526149292 y:  228.17847068402568
x:  -651.1201131192154 y:  -3286.552722011381
x:  89.03729710380304 y:  -234.08720421275828
x:  -17.842700038711765 y:  -261.5503953995988
x:  1988.8707732816579 y:  -2500.387558267078
x:  -182.18499837862794 y:  159.24859092719367
x:  -214.03022087408183 y:  -184.87476397187268
```

1.5 Computing vanishing point

To compute them, we sorted the lines by their angle, then clean the lines given by the hough transform to keep only two lines with approximately the same angle. Then we compute the intersection

of the two lines to find the vanishing point. There is an error, and if more than one group represent the same vanishing point, because the lines are really close in angle, we have multiple points really far from each other.

2 Estimating fundamental matrix

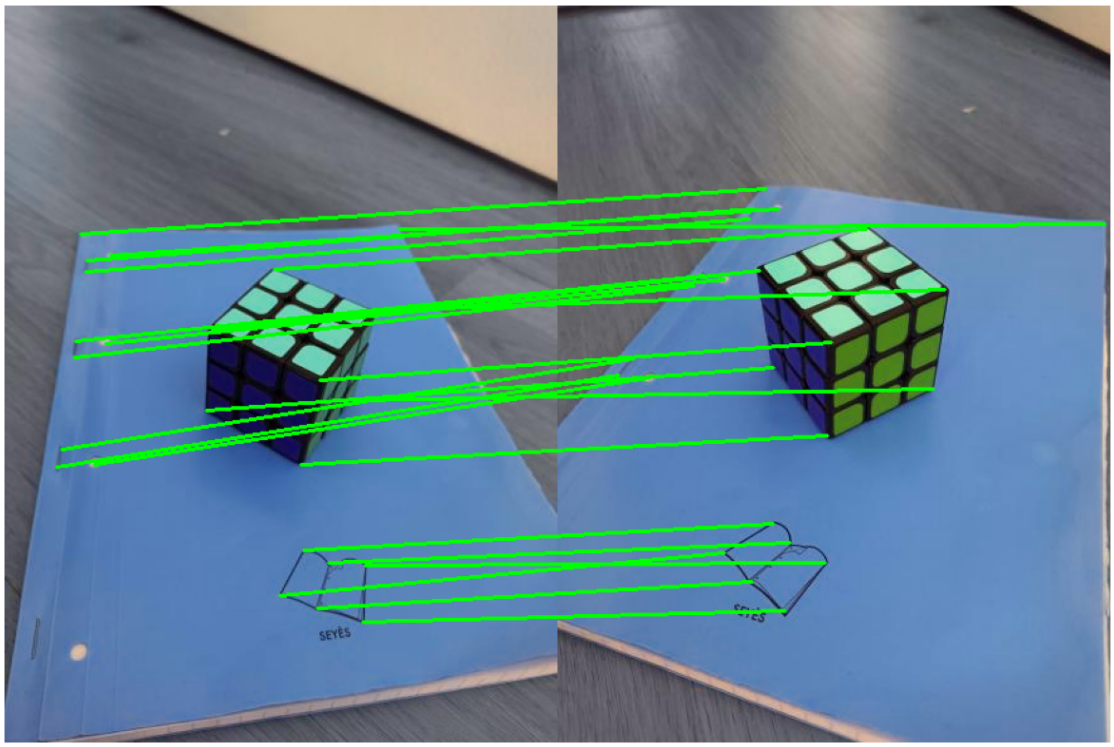
2.1 Point correspondences

We made a simple interface to select points on the image. It's commented by default in the code, but you can uncomment it to use it instead of raw data we hardcoded.

2.2 Estimation

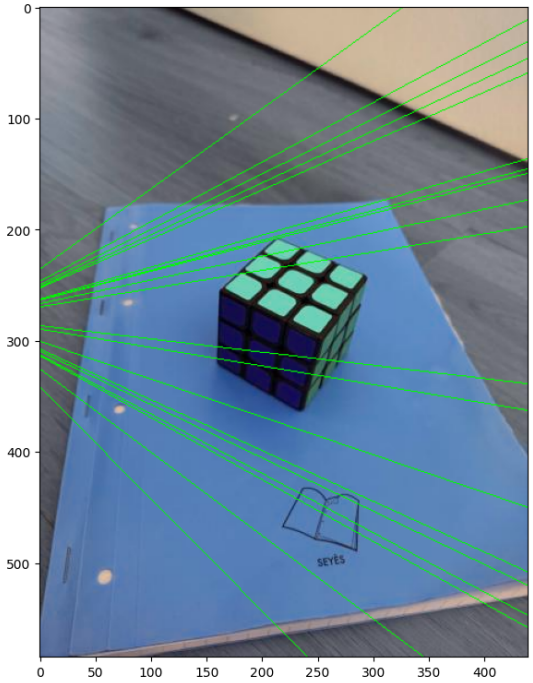
To estimate the fundamental matrix, we used the 8-point algorithm. With a RANSAC that is picking 8 samples each tries, create a matrix, and store the configuration that has the most inliers. We then use this configuration to display the epipolar lines.

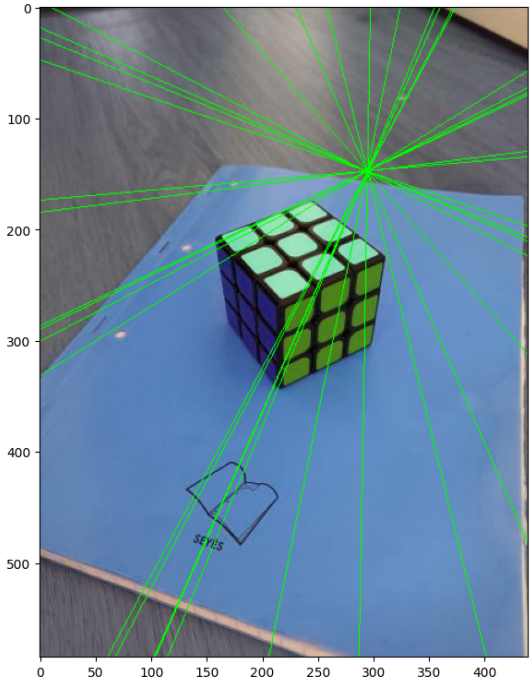
First the user pick the point he needs to match, the more the better because ransac will deal with the outliers.

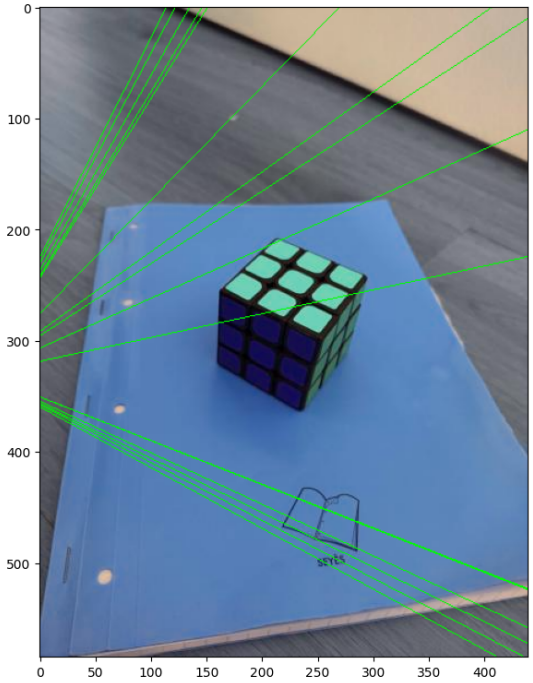


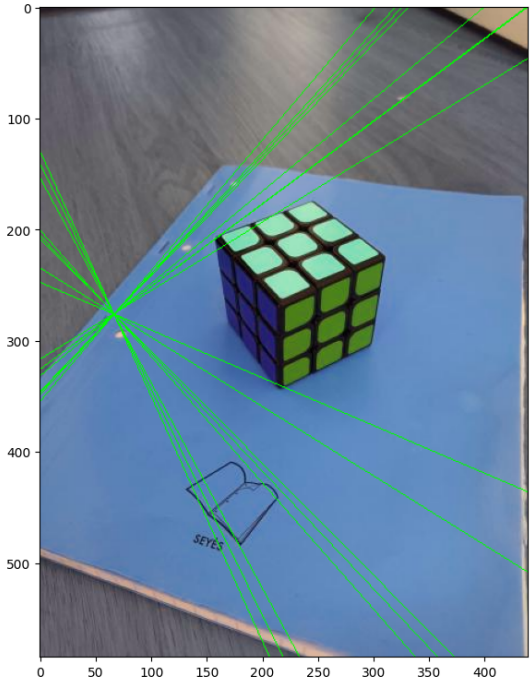
We compute the fundamental matrix with the 8-point algorithm, for 1000 iterations, and we keep the one with the most inliers. We then display the epipolar lines.

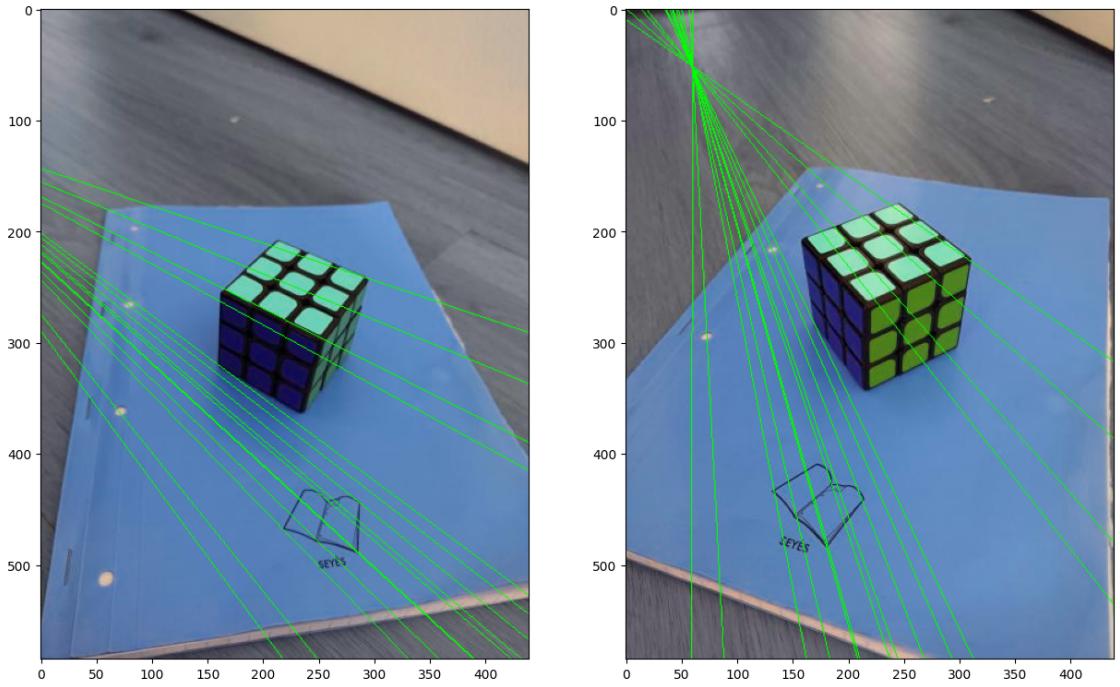
```
fundam quality:  0.002856599069785648  inliners 20
fundam norm quality:  0.0003617362777205764  inliners 16
opencv fundam quality: 6.778408034231215e-05 inliners 15
```


0
100
200
300
400
500
0 50 100 150 200 250 300 350 400
SEYES


0
100
200
300
400
500
0 50 100 150 200 250 300 350 400
SEYES


0
100
200
300
400
500
0 50 100 150 200 250 300 350 400
SEYES


0
100
200
300
400
500
0 50 100 150 200 250 300 350 400
SEYES



As we can see, OpenCV outperforms us, which mean there is something wrong with our computations, they are also pretty consistent between the two images, which means that we may have a problem with the fundamental matrix computation, if so we couldn't find it, it may be a problem with the RANSAC or the 8-point algorithm.

2.3 Normalizing

We tried to normalize the points and it gives us better results, still not as good as opencv but at least it is what we ecpected. It is due to the fact that the fundamental matrix is not scale invariant, so we need to normalize the points to have a better estimation, and it should also be more robust to eventual noise in the points.