Object pose estimation

Mautone Alberto

Contents

1 Introduction **3**

2 Structure of the code **4**
2.1 The header file 4
2.2 The .cpp file 6
2.3 The main file 8

3 Parameters used **9**
3.1 Canny parameters 9
3.2 Match method 12

4 Results **13**

1 | Introduction

The aim of this code is to better detect the **pose** of an object on some image tests, given a set of templates or models and returning a .txt file containing the best 10 object templates that better fit on the given scenario (test images). To do so, I chose to follow the approach of extracting **edges** from template and test images and then match them thanks to a **template matching tecnique**: this means that I haven't used the given masks, since it wouldn't have been helpful. Actually, when I apply the **Canny** algorithm to my images, I lose any area, and masks have a white area to be compared. Just for curiosity, I tried to use masks anyway, but clearly I've got very bad results. Edge-based template matching is an approach largly used since it can bring to very **robust** results: also it is supposed to be **fast** (it depends on many factors, including PC performance. I have an old laptop and, for this reason, it took more than one hour to get all the three .txt files).

2 | Structure of the code

To implement such program, I chose to use an ausiliar class, named "**PoseEstimation**". Of course it has a header file (.h) which contains the class constructor and a .cpp file where I've implemented all the functions defined in the header. Clearly this class is then used in the main.cpp file.

2.1 The header file

The header file "PoseEstimation.h" is the one containing the class **constructor**: it takes as explicit parameters two strings defined as class variables. This two strings are actually **two paths**: one leading to the directory containing all the templates images and the other one to the directory containing all the test images for that specific object we are taking in consideration. I need these paths because I want to **store** all the template and test images inside the constructor itself: to achieve this purpose I used two more class variables which are, of course, two vectors of Mat, one for the templates (so in our case will contain 250 images) and one for the tests (10 images). Since they are vectors, they don't need in input a pre-fixed size, so I can add all the images I need. Both vectors are filled thanks to two

functions defined in this file (*add_models_images(string)* and *add_test_images(string)*) and implemented in the .cpp one. Another vector is also defined, but it is a vector containing **String**. Why do I need it? Since in output I want to get also the **name** of the 10 best templates I need something to store all the models name. So, this vector<String> contains all the paths leading to a specific model of the template set. So, for instance, if the first image in the template directory is named "model0.png" then the vector<String> model_name will have:
model_name[0] = "C:\...\models\model0.png"
Also this vector is filled with a funcion (*add_models_name(string)*) implemented in the .cpp file .
Finally I can access to each one of these vector by the implementation of three simple *get...()* function, which just return the corresponding vector I'm requesting based on the function name.
Three more functions are defined: one of them returns an "**MValues**" object. This is a **struct** defined at the beginning of the file, and it is very useful to store all the info related to a model - test match: actually it contains:

- A string **name**, containing the name of the model;
- A double **s**, containing the level of accuracy of the match;
- A Point **y**, containing the traslation of the template with respect the test image.

The other function (*extract_best_ten(string)*) gives in output the .txt file containing the best 10 matches with all the requested informations. The name of the .txt file is

given as **explicit parameter** (string).
Finally, the last function is a function that **tests** all the best 10 template on each of the image test: it gives in output the specific template and the detected position (highlighted by a green **rectangle**) on the related test image. All the informations are **extracted** from the final .txt file produced by the previous function.

2.2 The .cpp file

Premise In this file I have **commented** all the output windows and their related "imshow(...)" and the function rectangle(...) which draw a rectangle around the detected object. This is done to avoid the **continous spamming** of opening and closing windows through the whole run of the code. If anyway you're interested in seeing **every single** output (and not only the best 10), they can be seen by simply decommenting the windows-parts and the waitKey(0) function. Anyway remember that, thanks to the "*test_result(...)*" function, used in the main file, it is shown only the **best 10 matches**.

In this file I've just implemented what defined in the header one. After the inizialitation of some useful **variables** (mainly Mats) I start by implementing one ausiliar function: *sort_by_s(...)*. It is very useful to order all the matching done basing on their accuracy (which is the "**s**" value defined in the struct MValues). Thanks to this function, I can extract only the **best 10 matching** per object. The three functions that fill the vectors defined in the header file are all implemeted by using the *glob(...)* func-

tion and a simple for loop: they all return the respective vector properly filled (one for the template images, one for the test images and one for the template names).
The next implemeted function is the **principal** one: *extract_best_ten(string)* that actually uses the other defined function (*Matching_Method(...)*) taken from the documentation and modified in some parts (more details **later** on the report). *extract_best_ten(string)* creates a MValues vector to store all the **matching infos** for all the 250 templates I'm evaluating for each test image, and, before changing the test image, it is totally cleared. For not loosing information, before clearing, I just write on the related **.txt file** all I need to know about the best 10 matches (model name and translation among x and y axis). To write on a .txt file I just use an **ofstream object**, having care to specify *ios::app* in order to keep everything I've written inside the text file, without deleting anything. To get the matching infos for each template, after some checks about the emptyness of the images and about the number of channels of the test image I'm considering, I just apply the *Matching_Method(...)* function, implemented as last function.
The *Matching_Method(...)* function uses some code from the OpenCV documentation: what I changed in the definition is the **return value** (in my case is a MValues) and the **explicit parameters** by adding a string s, which is gonna be useful to extract the template name. About the implementation I removed the **normalization** part (else I would have gotten all one and zeros values for the matching accuracy), set only one matching method (in particular I chose the fourth one: TM COEFF), removed the mask

part, and added the part related to the model name. After that I've got all the infos I **store** them in a MValues object.
Finally the last function is the one testing all the results obtained and stored in three .txt files (one per object). What it does is just some **string manipulations** on each line of each .txt file. After extracting the image test name and the model image name, it just get the images by following the **paths** given as explicit parameters: the last part is to get **shifts** among x and y axis and, after that's done, I just store all this integer values inside a **vector**. Thanks to a counter "**i**", I'm able to get all the shifts (and so rectangle corners) I need to draw the proper **green rectangle** around the detected object on the test images.

2.3 The main file

Finally the main files just uses all the stuffs defined above: in particular I create a **PoseEstimation object** for each one of the object I'm considering (can, driller and duck), and, after setting the proper paths, I just use the extract_best_ten(string) function in order to create my .txt file as requested and test them. **So the main file has to be modified in order to set proper paths on the user PC** (hard coding).

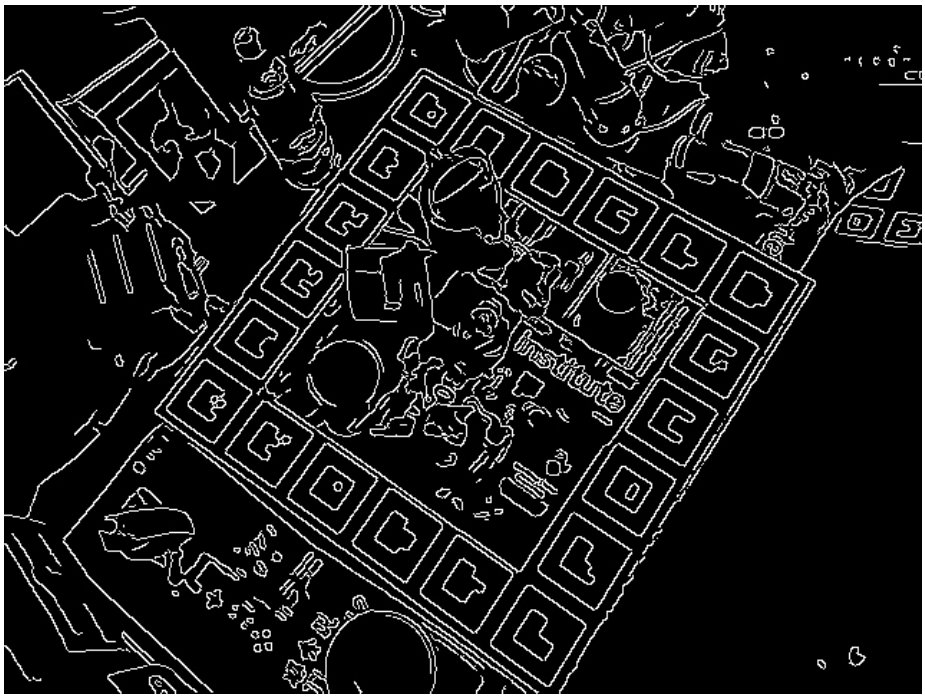
3 | Parameters used

Through the development of the code I had to do **many tests** to see where parameters changing would have brought me. Basically I had to deal with **Canny** thresholds parameters, trying to adjust them to make the project the most stable I could, in order to be run on different dataset (even different from the three given).

3.1 Canny parameters

These parameters refers to the value of the **double threshold** used by Canny: the low one and the high one. At **first** I tryed to use a random tool to test the most possible (reasonable) values, but, due to my laptop performance, I couldn't go on this way. So I chose to keep a **relationship** between the low and high threshold by setting the high one **2 times bigger** than the low one. The hardest part was that to find a value that could be good for **all the three** datasets given: clearly I didn't get 10/10 on each test image for each object, but, after all, I've reached good results. I have tried to have the most **similar** edge detected image both for the template and the test image, but, at the same time, this value should have been good for all the 3 objects. The first thing I did was to set parameters for

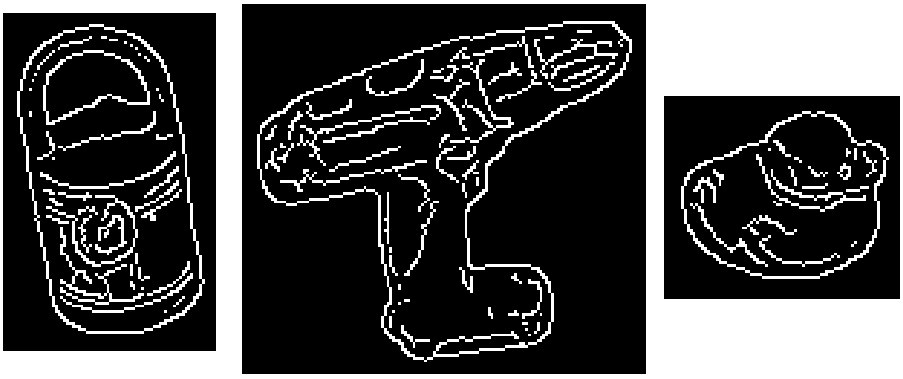
the edge-detection in the test images since they are, apart from the orientation, pretty the **same scenario**. I left the kernel size (sigma) to its default value (3), and I have set the two threshold values one at **60** and the other one at **80**. This way I didn't lost many details on the **whole** image, but of course, I lost some details **inside** each single object: my pourpose is that to select only the shape of the object with some **small details inside**, in order to match it with the shape of the template image.



Canny edge detector applied to a test image.

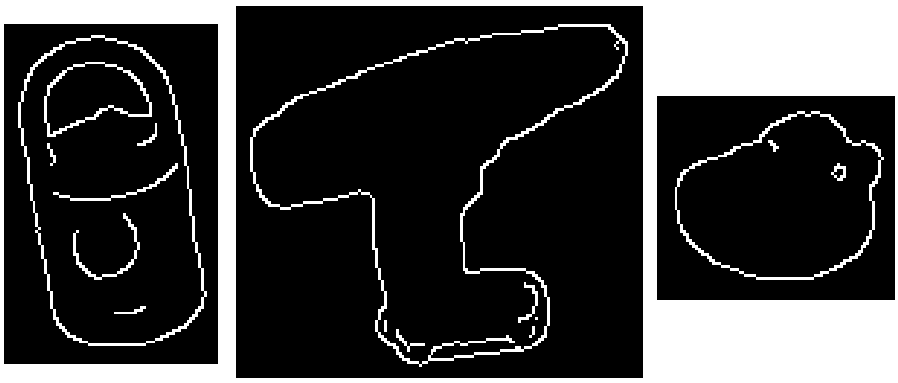
For what is concerned the template Canny detector, I tried many different parameters: lower ones gave me a **higher-detailed** edge image, while higher one a **less detailed** image. Since I chose to keep some details inside each object on test images, **in a first moment**, I went for **low**

values of the low threshold (remember that the high one is 2 times the low one). The point was to choose which specific value to use: the range was around **6-15**. Actually the higher values of this range (so about 13, 14 and 15) gave me **bad results on the "can"** object but **better on the "duck"** object, and viceversa for the lower values. At the same time "**driller**" results were **really really bad**, since I couldn't get any right pose estimation.



Canny edge detector applied to some template images with LT = 11 and HT = 22.

So my final choice was to select 40 as (low) threshold value, completely changing my first idea: making different tests for all the three objects I found out that 40 and 80 are the values that make me reach the **greatest** result (so far). All cans are properly detected, while better results are obtained for the other two objects. **Unluckly**, due my laptop performances I couldn't have tried a large range of values.



Canny edge detector applied to some template images with LT = 40 and HT = 80.

3.2 Match method

For the template matching I had to choose among **6 strategy**, identified as number (0 - 5) in the function *matchTemplate(...)*. In particular, two method, which are **SQDFF and SQDIFF_NORM** are suitable for using **masks**, but, as said before, I didn't find a sense on using masks in edge images. So I had to choose between four options: to be honest I tried **them all** before picking up the number four, which is **TM_COEFF** and this is why with this method I got the best results in terms of **proper pose detection and position inside the test images**.

4 | Results

So, overall the code **works well**, since it can recognize almost every pose for every test image for every object. In particular, basing on the **results** shown by the "test_results(...)" function I can do the following stats for each object, where **testx: y/10** means that for the image test number **x**, the code right detects **y** template poses (and centers it on the respective object on the test image) out of **10** (where 10 is the number of the test images).

- Can:

  test0: 3/10

  test1: 6/10

  test2: 3/10

  test3: 4/10

  test4: 6/10

  test5: 8/10

  test6: 2/10

  test7: 7/10

  test8: 1/10

  test9: 1/10

- Driller:

test0: 1/10

test1: 0/10

test2: 1/10

test3: 3/10

test4: 0/10

test5: 1/10

test6: 1/10

test7: 0/10

test8: 1/10

test9: 0/10

- Duck:

  test0: 0/10

  test1: 5/10

  test2: 2/10

  test3: 0/10

  test4: 0/10

  test5: 1/10

  test6: 1/10

  test7: 5/10

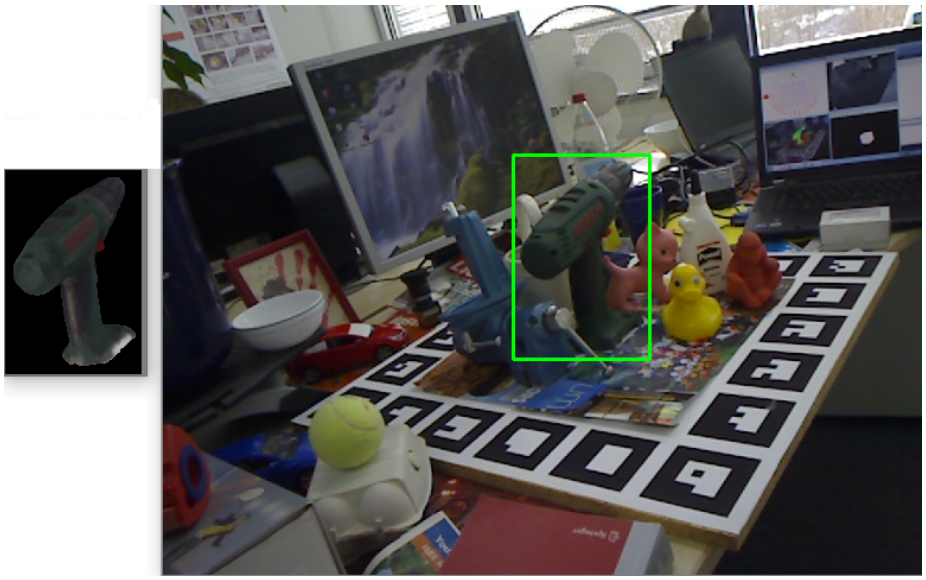
  test8: 0/10

  test9: 8/10

As reported, I've got the **best results on the can**, while the **worst on the driller**. Basically even though the pose was almost right, it wasn't recognized on the driller on the test images, but in **other parts**: clearly this isn't enough to consider it a good match. So I guess a **different**

threshold for Canny would have been helpful, but at the same time, I know that changing values will also **effect** the other two objects. I tried with different matching methods obtaining worse results, so I kept the TM_COEFF one. Overall, for the driller, I get **at least one right match** in 6 images out of 10 which is not a (very) bad result.
For what concern the duck I have noticed that "**planar**" ducks are never detected in the right position on test images: this is way they are almost **always detected on the black and white shapes around the tables**, which I guess, reminds more of the canny-edged template duck image. Maybe using a mask approach, instead of a Canny approach, would have given me better results. Overall I've got **at least one right match in 6 images out of 10**, so the same result of the driller. However, in the duck analysis I have **more right results per image**, especially in the last one, where I've got **8 right duck poses out of 10**. Here are some results example:

The first best result for the can image test number 0.



The first best result for the driller image test number 0.

The first best result for the duck image test number 1.