PROJECT WORK BY FRANCESCO CERRI, matr. 951972

francesco.cerri2@studio.unibo.it

FOR IMAGE PROCESSING AND COMPUTER VISION M 2021/2022 COURSE

HELD BY PROF. L. DI STEFANO

# COLOR BASED BACKGROUND REMOVAL AND OBSTACLES IDENTIFICATION FOR AUTONOMOUS ROVER NAVIGATION

TABLE OF CONTENTS

**LIST OF FIGURES**

**CODE AND COLORS SCHEME**

Since no code snippets will be included in this document, this part is intended to ease the search for reference for a given part of this work inside the code; whenever it is necessary, a note inside the text will point to a footer, that will display where the code includes that salient part, specifying the script and relative line.

Since, as will be shown in the following page, the code is intended to be used inside the ROS (Robot Operative System) framework, which could not be assumed as a standard package installed, I decided to provide a version that can (limitedly for what in concern available sensor streams and interoperability between script s and completeness of the code) work in analogous way.

Page footer will show (when appliable, obviously somewhere the ROS-free code will lack some features) the reference using this color code:

- Red color for ROS-able code
- Blue color for ROS-free code

Scripts that include ROS-able code are: noise_simulation.py, preprocessor.py, image_filter.py, object_detector.py, while ROS-free classes are totally encompassed in cv_proj_cerri.py script.
Note that all of the aforementioned scripts run in Python3.

INSTRUCTIONS TO RUN THE CODE

The folder containing this document will serve as root directory for the project.
Script cv_proj_cerri.py must be run from the folder scripts/ (it uses relative path to load the samples).
samples are contained in the media/video/samples/ folder; the parent folder contains the video, a prerecorded navigation camera stream.

A window showing some statistics like currently used filter, morphology operations level and object detection method; terminal will display some statistics, for example resolution of the video stream, and a full list of available options.

**Window's dimension can be increased/ decreased using keys "+" and "-" respectively**.

**INTRODUCTION AND GOALS**

What is the European Rover Challenge?

This code was developed with the purpose of being used by Alma Mater's **Alma-X Rover Team** in the upcoming European Rover Challenge this September.
ERC is an international student competition, where teams challenge themselves to (build and) operate an autonomous rover, inside a terrain which has some common (by morphology and appearance) features with respect to Martian surface (more about this in CHAPTER 3: TASK AND ENVIRONMENT ANALYSIS)

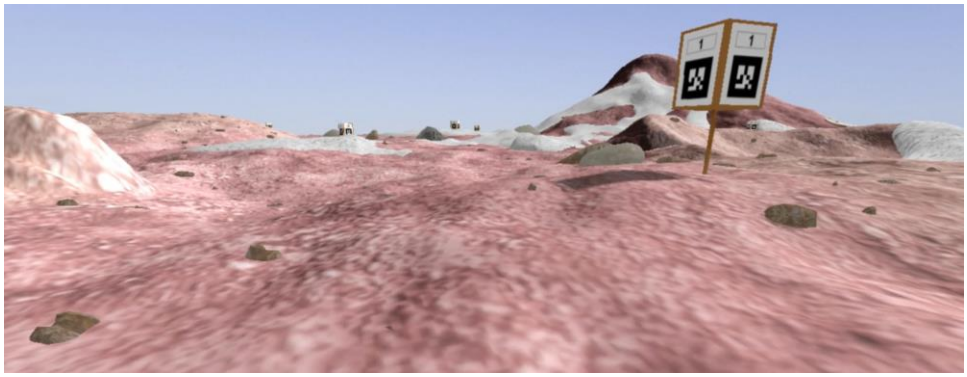

*IMAGE 1: ERC's MARSYARD AND SOME FEATURES*

Goal

This code provides a tool to help our team tackle the two main goals for the modality of participation we decided to attend: autonomously drive a given wheeled rover inside the challenge terrain, sensing the obstacles which cannot be traversed, and identify some interesting objects.
I hence decided to deal with this task simplifying the analysis by removing the more information possible from the image: by being able to mask the ground from the image, only remaining areas can be tested for our purposes.

What is ROS?

ROS is a collection of software frameworks, available for python and c++, typically used for robot programming, providing the capabilities of a (virtual) operative systems, with the purpose of synchronization and message passing between various processes (called Nodes); synchronization is controlled by a master node (in ROS1, fully distributed in ROS2) and it is possible through message exchange using some specific, named, broadcast communication channels (called Topics).
Basic units of ROS programming are Subscriber objects, which register to a topic and spawn a sleeping thread, ready to execute a certain callback function once a message is received on that channel and Publishers, on-demand sending a message on a given topic.

Without diving into much detail, ROS enables the user to easily exploit multiprocessing capabilities of a machine, without introducing too much overhead, controlling and synchronizing timing and rendez-vous and passing data between simpler scripts, enclosing it in a middle layer of abstraction.

Other valuable tools of ROS are Gazebo simulator, which simulates physical interaction of bodies and sensors' data in a given simulated environment, and Rviz, a visualization tool that is able to graphically show the values of sensors' data by subscribing to their topics.

**SIMULATION AND SETUP**

Testing for the competition is performed on a realistic simulated version of the Marsyard: features' geometry derives from 3D scans of the real terrain, onto which textures obtained by aerial photography are applied.

Simulation also includes a LeoRover mobile robot, which packs 2(3) optical sensors (ground level 5Mpx RGB and an upper RGD+Depth Zed2 stereo camera), as well as others that are not the primary focus of this work.

Provided code[1], based on ROS, utilizes sensors' data coming from the simulation and consists in 4 Nodes:

- noise_simulator[2]: registers to the RGB camera topic and publishes it, with addition of noise.
- image_preprocessor[3]: registers to noisy RGB camera topic and publishes it subsampled, smoothed (to a lower level of scale) and denoised; does similarly with Depth Camera topic.
- image_filter[4]: registers to subsampled RGB and Depth topics, computes a mask for the background (terrain and sky) and publishes cleaned foreground for RGB camera.
- object_detector[5]: registers to foreground topic and identifies obstacles and objects of interest.

The repository also contains ROS-free code (as a complement to this report), which instead uses only RGB data from a prerecorded video of the simulation; classes extracted and modified from the ROS-able code are included in a single script[6].

As for now, all the algorithms will be run locally on our machines, thanks to a live connection with the robot. I developed this project having in mind that this code will be reused once we develop our own robot (which will probably pack a Up Board, an Nvidia Jetson and a Beaglebone AI to exploit multi machine capabilities of ROS); hence I tried to test various ideas and keep all the best ones in terms of performances.

[1] repository at https://github.com/alma-x/cv_preprocess
[2] noise_simulation.py
[3] preprocessor.py
[4] image_filter.py
[5] object_detector.py
[6] cv_project_cerri.py

**TASK AND ENVIRONMENT ANALYSIS**

Marsyard terrain was built as a likely imitation of Martian surface for what it concerns colors of terrain, rocks, mud cones and castings. We know for experience that all the flat soil is mostly traversable and does not pose a particular risk nor interest (to be precise, control algorithm has all the capabilities to deal with it); terrain appears with a mostly red to orange tone, with some lighter or darker intensity patches.

The risk is posed by features like rocks, which tend to have a brown, grey (light to dark) color, mud cones and castings, which have a visibly lighter color, with the latter being closer to white.

Objects of interest instead present a color accurately designed to be spotted in this environment: probes are yellow and green, whilst navigation visual markers have a defined shape; moreover, they present visual AR tags, easily identifiable.

Instead, sky, and other objects outside the challenge terrain/area of interest, while having peculiar colors on our planet, have a reddish tone on Mars; in this case the depth data comes to hand.

Also, even if not controlled, environment lighting will stay pretty much the same (short duration of the task, limited areas in shadow)

The idea was hence to use the color of features, as well as their shape when needed, to try to separate foreground and background, since all of the aforementioned features' color resides in a specific range.

The analysis of RGB histograms clearly shows that, on average, R-component is predominant, hence most of the work will be done in that range.

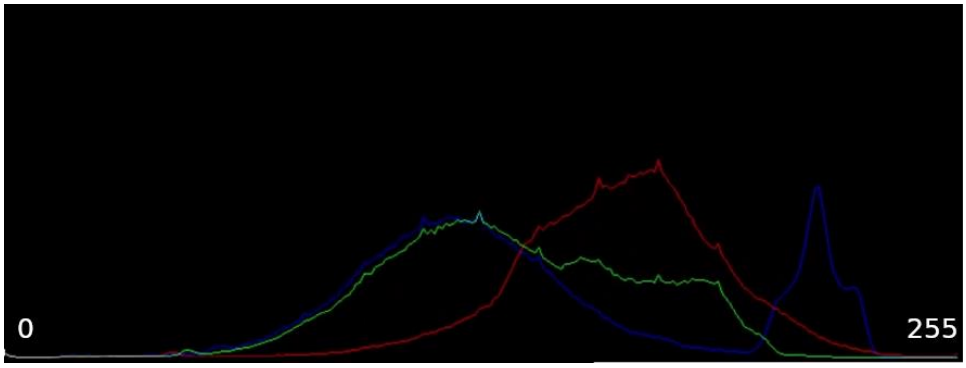


*IMAGE 2: NAVCAM FRAME's RGB HISTOGRAM SHOWING R-CHANNEL DOMINANCE*

To identify which pixels shall be masked at any time, use of Hue, Saturation and Value (HSV) color space presents clear advantages: first of all, tonality is identified using only Hue channel, which makes selection of range simpler[7] (1-dimensional instead of 3-dimensional, since for RGB one must weight the relative intensities of all channels and decide a percentage threshold for R-channel intensity) and secondly, H,S channels tend to be less device and conditions dependent, a feature that will help once I will test this algorithm on a different (real) setup.

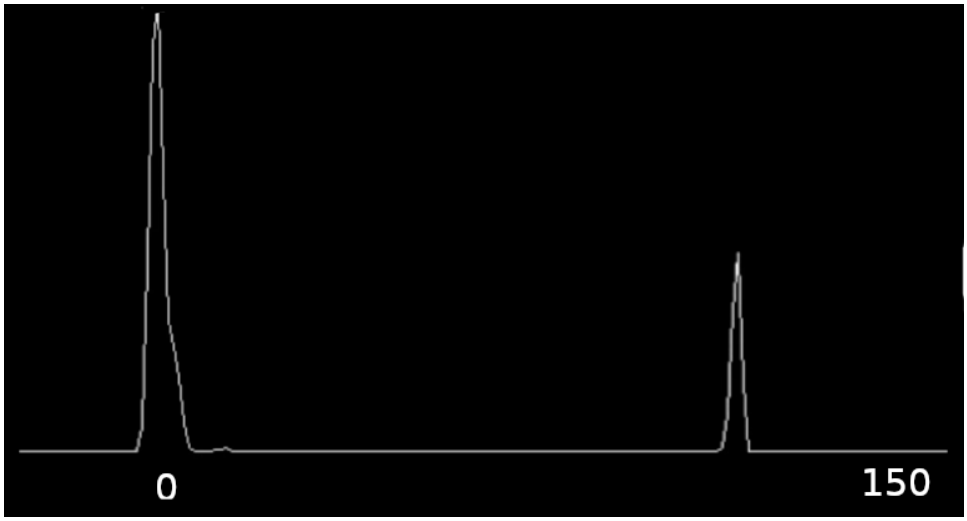


*IMAGE 3: NAVCAM FRAME's HSV HISTOGRAM TO SHOW CLEAR HUE PEAKS FOR GROUND (LEFT) AND SKY (RIGHT)*

[7] I reality, HSV black and white can occur for each value of H; yet, thresholding that values can be done easily by considering only the "central values" in S,V range. In any case, this would require tuning in a real setup and was one of the reason I decided to develop a samples-based, histogram analysis filter

**NOISE**

Since RGB sensor records images from a simulation, I decided it would have been better to manually add noise and try to recreate a more realistic situation.
In reality, Gazebo simulator already implemented a zero mean gaussian noise with standard deviation of 0.007; I decided to plague the signal with heavier noise to test the capabilities of noise reduction algorithms.

Class ImageNoiseSimulator[8] is assigned to this purpose: once it has initialized itself with a sample camera frame to learn image resolution, depth type and number of channels[9], main routine[10] awaits for a signal to come[11] and, upon receiving, uses noiseSimulation() method[12] to superimpose a noisy image to the raw one and publishes it[13].
Types of noise to add are stored in ImageNoiseSimulator.noise_type char variable; default values are a uniformly distributed and a gaussian distributed (representing summation of all other effects), zero mean signals, representing respectively quantization error and summation of all other sources. Value range/variance have been hardcoded to a very pessimistic value.

The effect of noise on histograms for Hue value clearly shows a broadening of the peculiar "pillars"

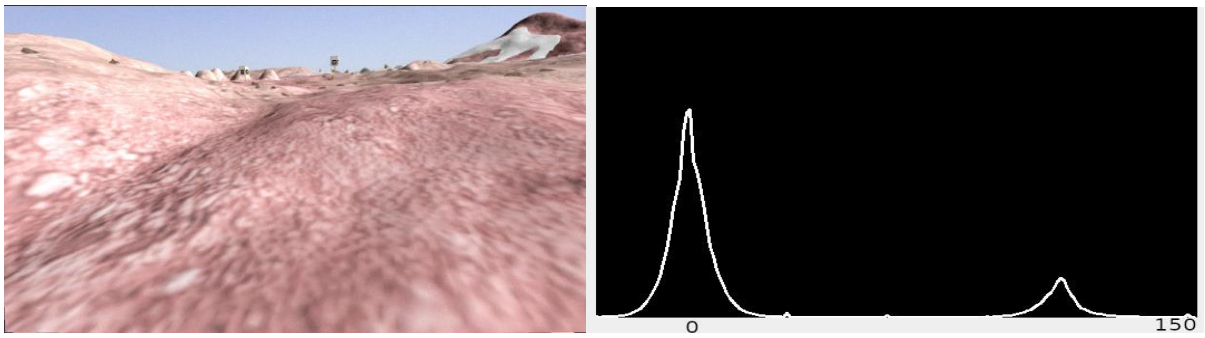


*IMAGE 4: EFFECT OF NOISE OVER IMAGE AND H CHANNEL*

Further improvements

The addition of impulsive noise is ready to be used (for example to simulate disparity matching errors over Depth channel), but was left unused since it's not the focus of this work.

[8] noise_simulation.py line 18, cv_proj_cerri line 37
[9] noise_simulation.py line 58, cv_proj_cerri line 55
[10] noise_simulation.py line 68, cv_proj_cerri line 64
[11] ros only; not-ros simply has input passed as arg of main routine
[12] noise_simulation.py line 92, cv_proj_cerri line 76
[13] ros, but with not-ros data is not publiched, but directly passed to downstream class as input

**IMAGE PREPROCESSING**

Noisy camera stream needs to be denoised; to simplify the computation, since wanted features to analyze posses a bigger scale than the smallest one of the image, subsampling and gaussian smoothing ("motion" to a bigger scale level of a gaussian pyramid) is performed first.

Class ImagePreprocessor[14] is assigned to this purpose: once it initializes itself to learn the input resolution, depth type and number of channels[15], and the required ratio for subsampling, main routine[16] awaits for a noisy input[17] to downsample it using downsampleCamera() method[18]. Output is then feeded to denoising noiseRemotion() method[19]: I tested various steps inside of it, feeding a convolution with a denoising kernel[20] to a bilateral filter and viceversa, and also passing the result to a median filter, but the best result, in terms of performance and efficacy and conservation of edges, was obtained using a bilateral filter after a denoising one[21]. Result is then converted and republished[22].

Denoised Hue histogram shows clear results for what it concerns homogeneous tightening of the "pillars", making the selection of color threshold easier.

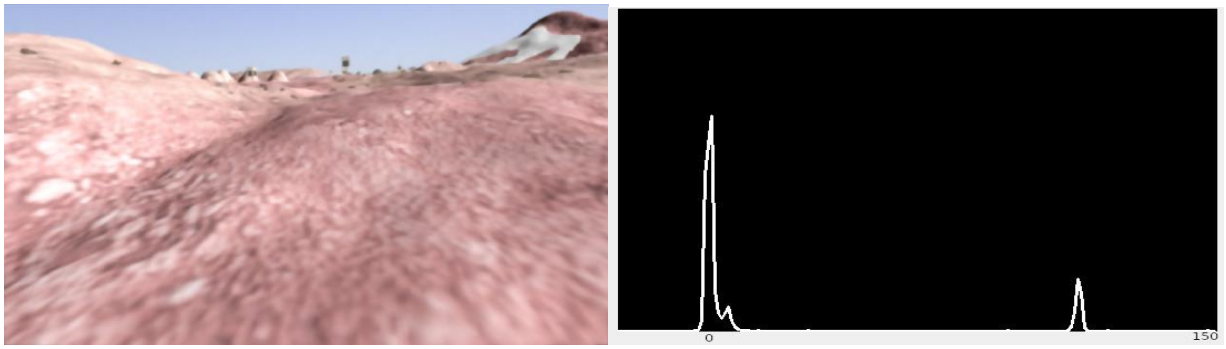


*IMAGE 5: DENOISED MARSYARD IMAGE AND H CHANNEL'S PEAKS*

Further improvements

In the case of presence (i.e. addition in this case) of salt and pepper noise to Depth channel, introduction of a median filter on that channel becomes fundamental.

---

[14] preprocessor.py line 18, cv_proj_cerri line 111
[15] preprocessor.py line 60, cv_proj_cerri line 129
[16] preprocessor.py line 70, cv_proj_cerri line 142
[17] same distinction as before for ros-able code and ros-less
[18] preprocessor.py line 105, cv_proj_cerri line 156 also; ros-able code does it for depth
[19] preprocessor.py line 129, cv_proj_cerri line 176
[20] [[1, 2, 1],[2, 4, 2],[1, 2, 1]]/ 16
[21] lib_filters contains definitions; kernel size=5 for speed, variances of the spatial and intensity gaussian filters are equal, being that 45
[22] ros-able only

**BACKGROUND SEPARATION**

After the preprocess, the camera stream is ready to perform foreground-background separation, the focus of this project work.

Since ground mask has the highest computational cost of all the pipeline, I decided to ease it by applying a further pre-filtering step, dedicated to the remotion of the sky from the image.

Class GroundFilter[23] is assigned to this purpose: after initialization of input resolution (and others, as above) and the default state of the filter[24], main routine[25] awaits for an incoming message[26], applies in sequence pre-filter, filter and filter refinement, then outputs the converted output. To test the different settings at runtime, it also listens for pressed keys (a complete list[27] of possible action is also available at launch).

Prefiltering

Initially, I experimented with the possibility to simply cut off the part of the image containing the sky, basing on the hypothesis that the sky is always above the ground in the image (unless robot capsized) and that the terrain in mostly flat, making sky and ground patches approximable with rectangles; by counting the number of pixels inside a specific range, centered around sky and ground peaks in the Hue histogram[28] (hence dividing Hue channel in 2 complementary sequence of bins) I could more or less decide the relative height of sky and ground rectangle, then keep only the lower one[29]. This presented problems regarding objects standing out in the sky (like the visual AR tags) or when the rover was tilted (i.e. working on a slope). Instead of using the IMU data and since, in the end, it is a color analysis, I decided to do it directly.

The preMask() method[30] takes care of this step: the first available metho uses a predefined color range and masks it from the image; the second, uses a more robust approach by masking the pixels which have a non-numeric distance value (from Depth camera), like when bigger than max sensor‘s range. Whilst more robust, it suffers from the fact that also at distances lower that min range sensor shows this

---

[23] image_filter.py line 114, cv_proj_cerri line 208
[24] image_filter.py line 234, cv_proj_cerri line 245
[25] image_filter.py line 368, cv_proj_cerri line 262
[26] ros-able only
[27] s: displays sample filter
r: displays range filter
z: enables all samples in sample folder for respective filter
x: disables all samples
l: add new sample(s) to filter from ROI(s) selection;
multiple allowed, esc to exit
k: same as 'k' & new samples are saved in ./media/samples folder
m: increase morphological operations amount
n: decrease morphological operations amount

[28] image_filter.py line 465: cameraAnalysis() method
[29] image_filter.py line 524, splitCamera() method
[30] image_filter.py line 543, cv_proj_cerri line 288

value; also, some ground points that are too fare are removed. Yet this method resulted really effective and fast, since the distance of interest is always contained in the useful distance range (as for when an AR tag is identified, that part is not masked).
Since Depth data is not present in ROS-less code, only ”color” method is available.

Filtering[31]

The resulting image is then processed for ground identification. For this purpose I developed 3 different solutions and I compared them with 2 (3) existing ones in terms of results and computational speed. Starting from the latter:

Otsu’s filter[32]: simply uses cv2.threshold() with cv2.THRESH_OTSU parameter. Performant, but suffers from non-clearly bimodal histograms (eventuality that could happen in homogeneous terrain); it also affected AR tags and castings.

(Spatial adaptive f.[33]: uses cv2.adaptiveThreshold(). Performance was really bad, hence it’s only named for matter of inclusiveness)

Histogram balancing f.[34]: uses as threshold the barycenter of the histogram. A bit slower, working better in mostly all conditions (including uni-modal histogram) except for a specific spot in one of the craters. Like Otsu’s, it also affected AR tags and castings.

Distance f.[35]: thresholds pixels using the local distance values from the images average intensity. Both Mahalanobis and Euclidean are available in the designed function[36], which uses Numpy’s Einstein’s convention for matrix multiplication[37] to improve speed. Despite that, it was too slow to make it usefull.

Range f.[38]: a naïve approach, since it uses hardcoded ranges of Hue channel to threshold using cv2.inRange() method. This performs extremely well in the simulation, but it should be treated for comparison only. It could be possible to use this in the real setup by considering a more device and conditions independent color space, e.g. LAB space.

I tested an algorithm of image clustering[39] with the intention to extract the dominant color(s) from the image to use it as reference value for distance and range filtering, but all available methods[40] were

[31] Red values have their mean at Hue=0 and extends above and below it; a translation of the whole channel was mandatory to avoid duplicate many functions (remapping Hue’=Hue+shift amount)
[32] image_filter.py line 581
[33] image_filter.py line 612
[34] image_filter.py line 633
[35] image_filter.py line 752
[36] image_filter.py line 791, myDistance() method
[37] numpy.einsum()
[38] image_filter.py line 690, cv_proj_cerri line 313
[39] image_filter.py line 420, cameraClustering() method
[40] cv2, sklearn, scipy

definitely too slow. Yet, this could be a useful method to run offline and configure the filter.

Sample f.[41]: so far, filters lacked performance or robustness. While I could forgo a reasonable amount of the first, flexibility to conditions (hence the possibility to fully automatize this step) was my focus.
In this filter mask is constructed incrementally by extrapolating histogram of various representative samples of the terrain (or any other unwanted feature), which are stored in a GroundFilter.sample array.

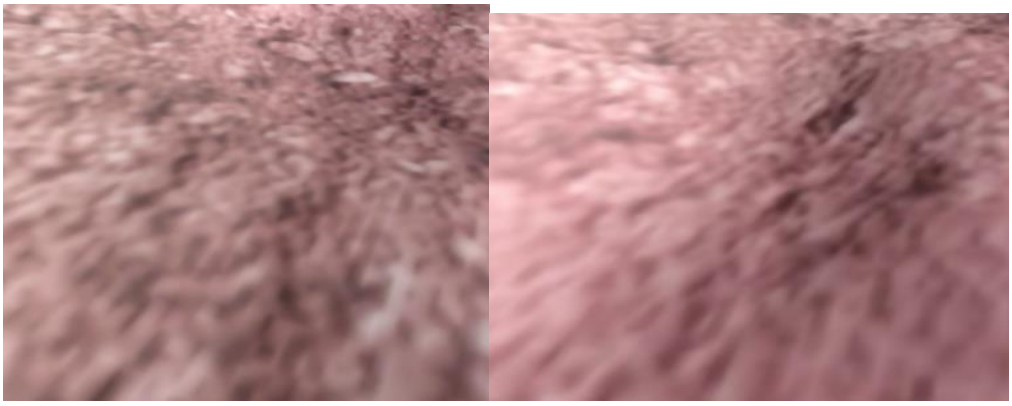

*IMAGE 6: GROUND SAMPLES*

Loading of samples can be done offline[42] or online[43], either saving the sample in the respective folder or simply loading its histogram in the array; it is also possible to remove all[44] or the last one[45] of the histograms loaded at runtime.
Once the selected samples' histograms are loaded, for each of them, the algorithm performs a statistical comparison with HS channels histograms of the current camera frame by computing the back projection mask[46], showing for each pixel of the image the probability it can belong to that particular sample. Each of these masks is then refined using convolution with an elliptical kernel, normalized in [0,255] range and thresholded[47]. Total mask is obtained by Boolean sum[48] of the previous ones.

Mask Refinement
Each filter's last step is to refine the obtained mask[49] to remove the smaller structures remaining after

[41] image_filter.py line 813, cv_proj_cerri line 352
[42] image_filter.py line 957, cv_proj_cerri line 443; loadAllSamples() method is called during initialization

[43] image_filter.py line 982, cv_proj_cerri line 464; key action can trigger the call of addNewSample() method
[44] image_filter.py line 1071, cv_proj_cerri line 504
[45] image_filter.py line 283 removeSampleCallback() method; available only for ros-able code. Analogous method which waits for a specific message, also available for adding a new sample image_filter.py line 270
[46] image_filter.py line 881, cv_proj_cerri line 383

[47] image_filter.py line 889, cv_proj_cerri line 387
[48] image_filter.py line 891, cv_proj_cerri line 389
[49] call for GroundFIlter.maskRefinement() can be found at the end of each filter, before show_result condition

the filtering step, which will cause wrong result in the object detection step.
Implementation uses maskRefinement() static method[50], which as a set of tested ”refinement level”, with increasing strength; levels giving the best results implements closing(1) and closing followed by erosion(2) using, by default, elliptical kernel of size 5.
Level can be changed at runtime, and the operation can even be turned off to test its effects.

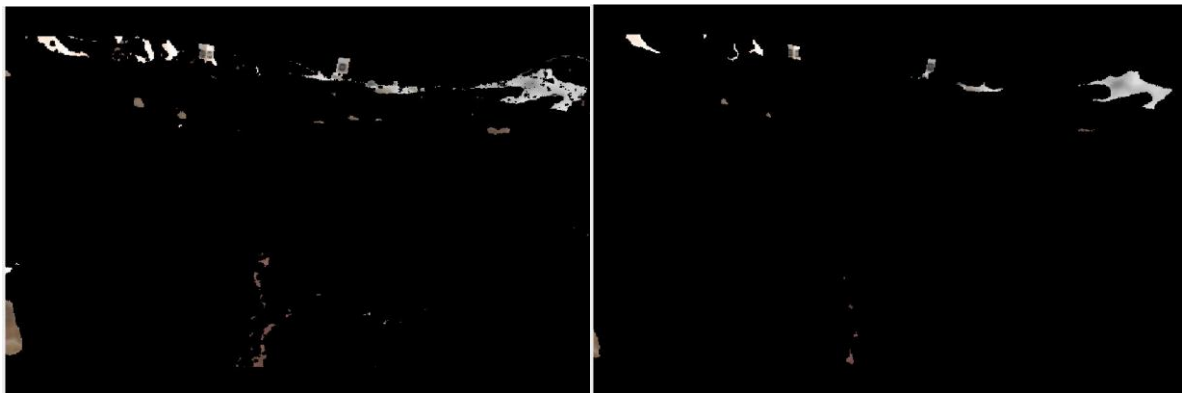

*IMAGE 7: MASKS AT DIFFERENT LEVELS OF REFINEMENT (LEFT: LOWEST, RIGHT: 2/6)*

Results

Sample filter showed a good efficacy without the need of too much tuning: the only parameter that is, in fact, hardcoded is the threshold level for acceptance of the mask.
The best results are obtained by taking larger sample images (and not smaller and more specific ones as I thought at the beginning of this work), without any real decrease in performance compared to smaller samples, since histogram computation is done offline/once, and not every cycle.
It also has a clear advantage over Otsu’s since it does not require bimodal histogram.

Considering absolute performance, average cycle repetition time for the algorithm settled to 6ms (compared to <1ms of the reference InRange filter), which is more than enough to sustain the 30 fps which are usually set as default level in the Gazebo/Rviz environment.

[50] image_filter.py line 991, cv_proj_cerri line 402

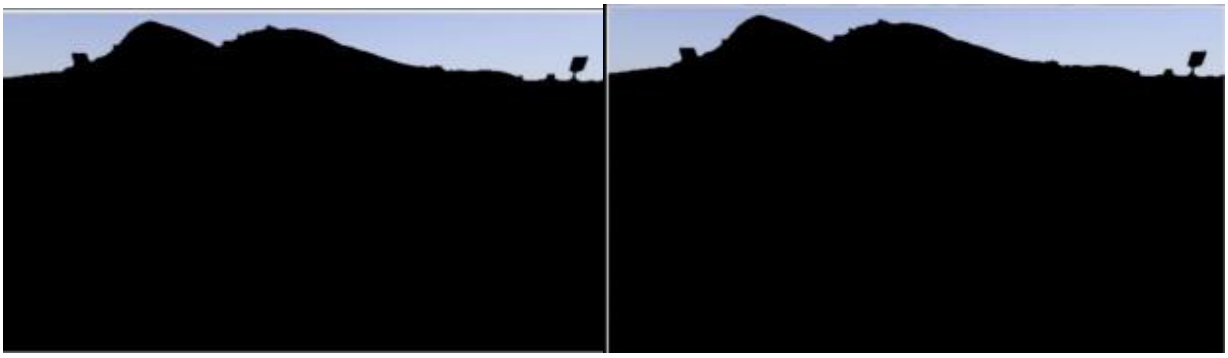

*IMAGE 8: POOR PERFORMANCE OF OTSU'S AND HISTOGRAM BALANCING FILTERS TO TELL GROUND AND OBSTACLES (ROCKS,…) APART. SKY LEFT FOR CLARITY*

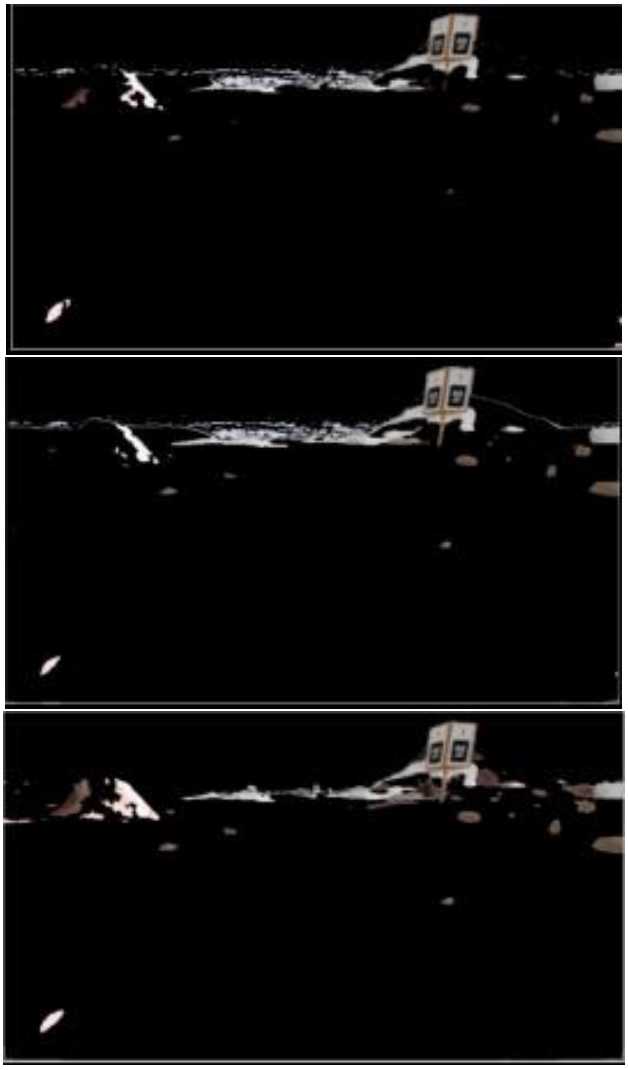

*IMAGE 9: PERFORMANCES OF SAMPLE (ABOVE), InRANGE (CENTER) AND DISTANCE (Mahalanobis, BELOW) FILTERS*

Further Improvements

Acquisition of samples in a single instance of time makes them prone to noise: by keeping the rover steady, it could be possible to acquire many samples of the same area and average them (it must be noted that this effect is partly rejected by taking bigger samples, which will contain multiple instances of the same color).

A major improvement will be the automatic selection of the samples; an already implemented[51] (but excluded) step of the algorithm was the test of correlation for the histograms of the samples: comparison element can be based over the histogram of dominant colors (obtained with clustering) or an average of the ground, extracted by the sky/terrain separation routine described in the prefiltering paragraph. Samples selection method could be implemented using this same routine.

[51] image_filter.py line 1036

**OBJECTS RECOGNITION**

Once the filtered image is obtained, it is fed to the object detection step: class ObjectDetector[52] is assigned to this purpose; after the initialization of the image resolution, the cleaning and the detection methods[53], main routine[54] awaits for an incoming frame[55], converts the image to grey scale[56] and then performs in sequence image binarization[57], edges detection[58] and contours identification[59]. Then it proceeds to draw[60] them and show the result.
Key listening[61] lets the user decide if prefiltering and input image cleaning must be performed, which type of detector to use and the visualization of the biggest identified contours, any or rectangular specifically.
As for now, this step is but a mere visual help for the robot operator/supervisor to identify the interesting objects laying. Steps to further improve this section are identified later on.

Preprocessing

Input image still retains some spurious elements. After the conversion to greyscale, and the binarization, using as "pass" threshold any value greater than zero (hence the pixels not masked by the color filter), a median filter is applied[62]: by comparison, I tested a sequence of both opening and closing and the result was far much worse; since most of the pixels are zero in the image, median filter does not add too much overhead.

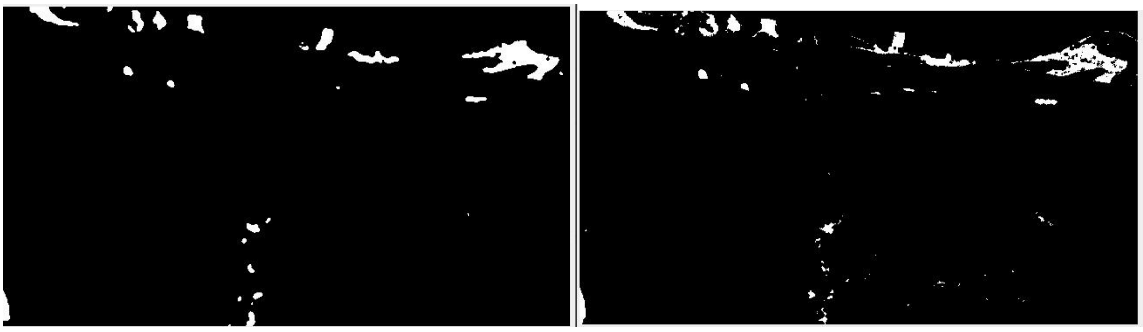

*IMAGE 10: COMPARISON WITH (LEFT) AND WITHOUT (RIGHT) PREPROCESSING*

[52] object_detector.py line 75, cv_proj_cerri line 566
[53] object_detector.py line 144, cv_proj_cerri line 595
[54] object_detector.py line 207, cv_proj_cerri line 607
[55] ros-able only, using message callback
[56] object_detector.py line 207, cv_proj_cerri line 607
[57] object_detector.py line 216, cv_proj_cerri line 620
[58] object_detector.py line 227, cv_proj_cerri line 634
[59] object_detector.py line 259, cv_proj_cerri line 668
[60] object_detector.py line 294, cv_proj_cerri line 683 onwards
[61] e: toggles enclosing rectangles
y: toggles canny's edges
h: toggles high filter for edges detection
i: inner edges, morph based over dilated-original image
P: toggles prefiltering (median)
[62] object_detector.py line 218, cv_proj_cerri line 622

Edges Detection

Due to the characteristics of the binary image, edges can be simply found by subtracting the original image to a version of it which has been subjected to dilation,[63] using a simple square kernel of size 3.

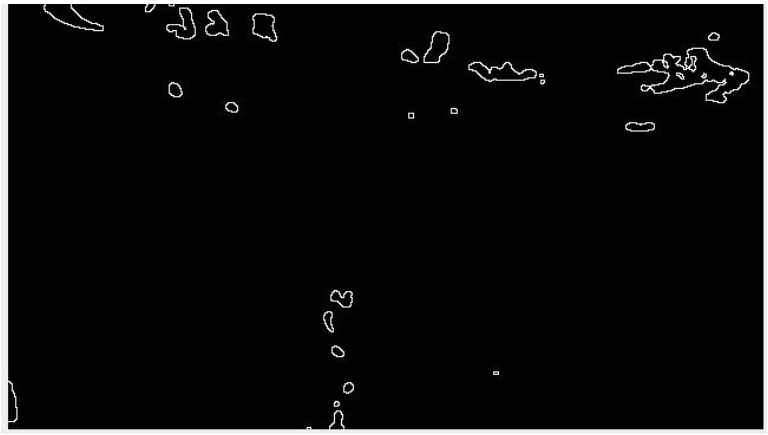

*IMAGE 11: EDGES' IMAGE*

I also tested Canny's edge detection algorithm[64], with different values of low and high thresholds, obtaining constant results (grey image presents sharp edges due to the majority of black pixels), and an high pass filter[65]; both also include the internal edges, which are not of real use (they could be useful for the visual markers, which are affixed to boxes).

Contours

Contours are found using cv2.findCountours()[66] method. Since the image is not perfect, I wanted to threshold the dimension of the detected ones: I sort contours by area[67] and keep only the biggest 20[68]. To identify the boxes, a test over shape[69] is performed: testShape()[70] method approximates contours using the perimeter and let only the ones having a specified number of edges (by default 4) pass; after being sorted by area[71], only the biggest 10 are then selected[72].

[63] object_detector.py line 227, cv_proj_cerri line 634
[64] object_detector.py line 242, cv_proj_cerri line 652
[65] object_detector.py line 249, cv_proj_cerri line 659
[66] object_detector.py line 259, cv_proj_cerri line 668
[67] object_detector.py line 277, cv_proj_cerri line 671
[68] object_detector.py line 278, cv_proj_cerri line 672
[69] object_detector.py line 279, cv_proj_cerri line 673
[70] object_detector.py line 325, cv_proj_cerri line 699
[71] object_detector.py line 280, cv_proj_cerri line 674
[72] object_detector.py line 281, cv_proj_cerri line 675

Visualization

All contours are filled[73], using cv2.drawContours(), with a specific color: purple for any of them, yellow for rectangular ones. The ones passing the test over dimension are enclosed in a (not minimal) rectangle[74], oriented the way the image reference frame is, using green for the rectangular ones, cyan for the others.

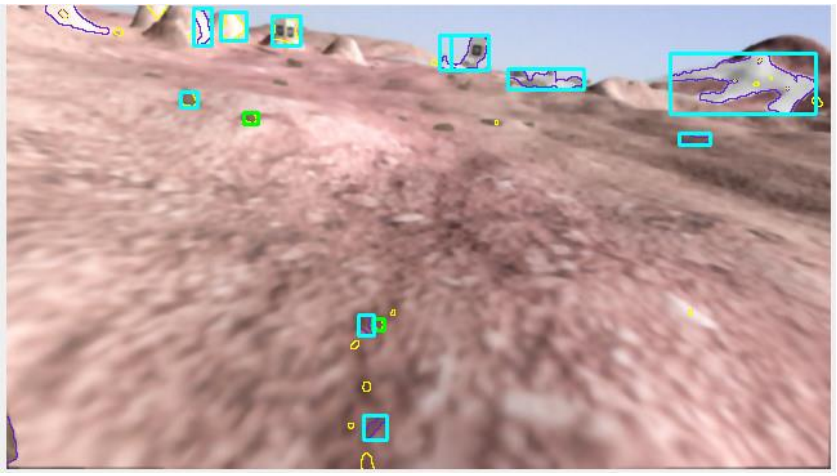

*IMAGE 12: DETECTION OVER INPUT STREAM*

Results

The algorithm is able to find and tell apart all the boxes supporting the visual markers at any distance and most of the bigger obstacles at medium distances (4m range). As said above, it gives more pertinent results with respect to Canny's algorithm.

[73] object_detector.py line 294, cv_proj_cerri line 683

[74] object_detector.py line 296, cv_proj_cerri line 685

Further Improvements

Results provided by this algorithm can be considered useful per se, since it gives a precious visual help to the operator assigned to the identification of interesting objects in that specific task (as for now, it is not already automated). Easily, position of the visual center of the objects could be estimated using the barycenter (or more easily, the center of the enclosing rectangle) of the "blob-like" features after binarization and retrieving the relative value in the depth map.

A more precise identification method for rectangular contours could be implemented by comparing its area to the one of the enclosing rectangle and thresholding the results; moreover, since depth data is available, using the enclosing rectangle's length (scale of the direction of maximum elongation) and width (referred to the minimum elongation), a comparison between values reconstructed from these and the real ones (given by the organizers) could help tell from far away (outside a range where the resolutions lets the rover identificate the visual AR tag with a proper algorithm) it the object is in fact a visual marker (just after a thresholding step).

As for what it concerns some of the obstacles constituted by levigated rocks, which tend to present a smooth elliptical shape (which, in the context of the competition, could also show the past present of water, which is a thing all teams should consider) ellipticity value could be considered (ratio between blobs's area and one of an ellipse built with same length and width; also in this case thresholding would be required).

To implement all the routines above, separation of blobs using a labelling algorithm must be performed first. In the first version of this work (not included here), for the contours too close to be discerned by a simple labelling algorithm, I tested separation using watershed algorithm, obtaining fairly good results

**THANKS FOR YOUR READING, COMMENTS AND SUGGESTIONS ARE WELCOME**