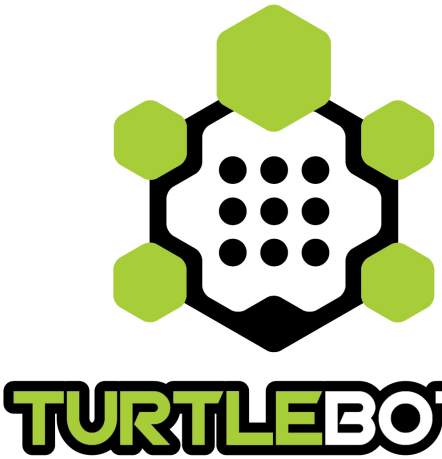
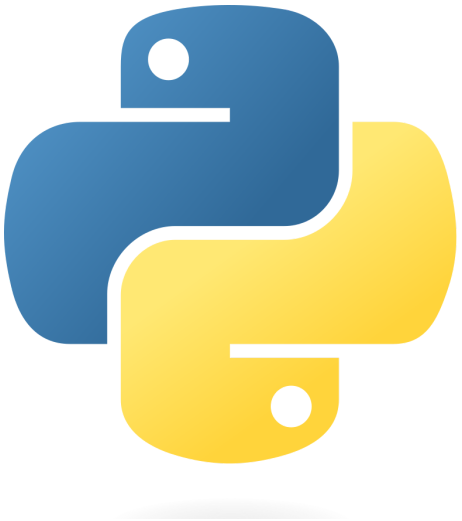
Making and editing a cartography





Work done by Adrien MABIRE

Teacher: Rédha HAMOUCHE

May 2023

1. Context and goals

This project's goal is to allow me to catch up for missing academic credits (2 ECTS) due to a failed exam in robotics [1].

The proposed project is to create a map of an environment using a TurtleBot3 running with ROS and then, offline, be able to edit the created map and reinject it. The main part of the project is the editing. This project represents the continuity of one of the robotic subjects on which I worked last year where I started on the end to create a map using the same set-up.

2. Design

All the parts have been coded using Python. The choice of Python was motivated for four reasons:

a) I was looking for a serious reason/project to start learning Python;
b) To speed up the development process;
c) ROS is available either in C++ or Python ;
d) Making a GUI is easier in Python compared to other programming languages.

2.1. Requirements

The design requirements are

1. to create a more robust and non-binary map compared to my previous year's work;
2. to be able to share this map ;
3. to be able to graphically edit the shared map and then feed it back to a robot.

The following sections explain how these three points have been solved, criticize some design choices and point out possible improvements.

2.2. Mapping the environment

In order to map the environment two pieces of information are needed: the readings from the Lidar and the position/orientation of the robot. In the TurtleBot3 they are respectively provided by the topics /scan [2] and /odom [3].

---

[1] https://www4.ceda.polimi.it/manifesti/manifesti/controller/ManifestoPublic.do?EVN_DETTAGLIO_RIGA_MANIFESTO=evento&aa=2022&k_cf=225&k_corso_la=473&k_indir=XEN&codDescr=052366&lang=EN&semestre=1&idGruppo=4579&idRiga=286615

[2] http://docs.ros.org/en/melodic/api/sensor_msgs/html/msg/LaserScan.html

[3] http://docs.ros.org/en/noetic/api/nav_msgs/html/msg/Odometry.html

The two topics are published neither at the same rate (respectively 5Hz and 30Hz) nor precisely at the same moment. Thus, to minimise the noise on the map, the measurements provided by both topics must only be used if they are close enough in time, as shown in Figure 1. Only the information contained within the messages that were received within a certain amount of time is used.

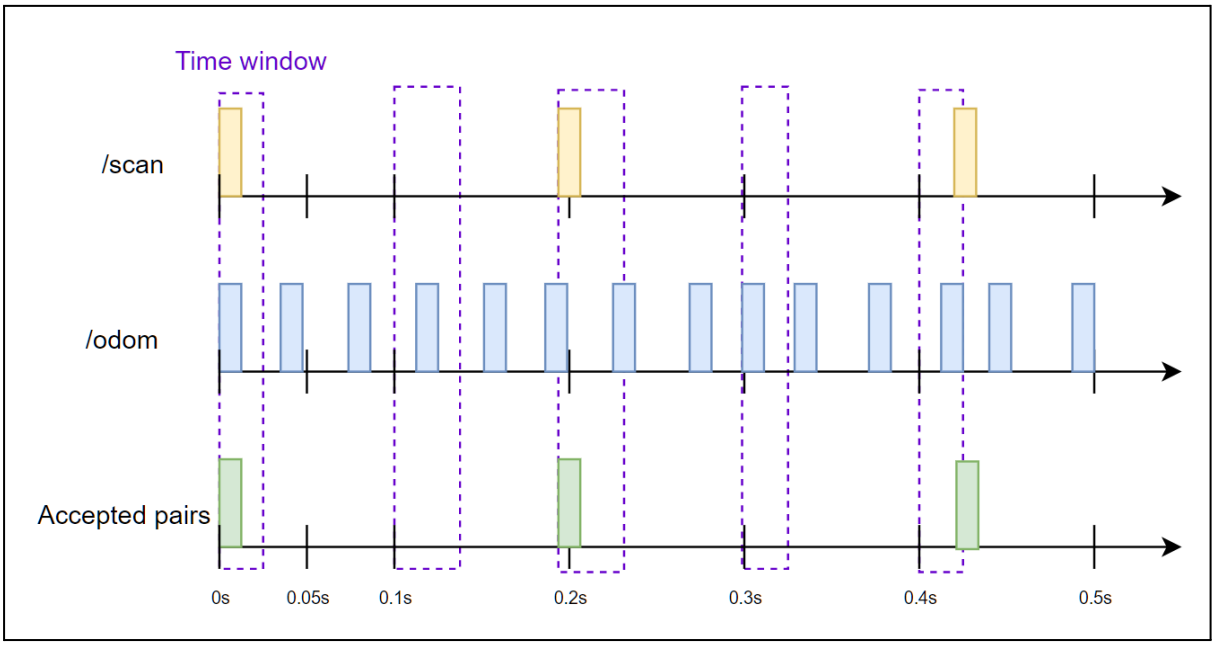


Figure 1: Time filtering of the messages

Thus the beginning of the ROS node is the following:

```
def mapping():
    NodeFreq = 10 #/scan=5Hz, /odom=30Hz
    TimeSyncWindow = 0.25*1/NodeFreq

    Scan_sub = message_filters.Subscriber("scan",data_class=LaserScan)
    Odom_sub = message_filters.Subscriber("odom",data_class=Odometry)
    Synchronizer = message_filters.ApproximateTimeSynchronizer([Scan_sub,Odom_sub],1,TimeSyncWindow)
    Synchronizer.registerCallback(SyncCallback_Map)

    rospy.init_node("Mapper",anonymous=False)
    rate = rospy.Rate(NodeFreq)
#...
#while-loop which only displays the map and put the node to sleep
```

The frequency of the node has been selected to respect Shannon's sampling theorem and the time window is defined as 25% of the node's wake-up time in order not to miss any messages. Indeed, a message can have a significant time shift (positive or negative) relative to the node's frequency.

After which, the callback function computes the measured points' position and updates the map accordingly:

```
DimX = np.array([-3,3])
DimY = np.array([-3,3])
EpsilonX = 0.015 #meters
EpsilonY = 0.015 #meters
Threshold = 10 #Used for the display and sharing
CoordX = np.linspace(DimX[0]-EpsilonX/2,DimX[1]+EpsilonX/2,((DimX[1]-DimX[0])/EpsilonX)+1)
CoordY = np.linspace(DimY[0]-EpsilonY/2,DimY[1]+EpsilonY/2,((DimY[1]-DimY[0])/EpsilonY)+1)
CoordMapX,CoordMapY = np.meshgrid(CoordX,CoordY) #Used for the display
Map = np.zeros((CoordX.size,CoordY.size)) #Initialise the map
Angles = np.radians(np.arange(360))
global ScanRanges
global Pos_X, Pos_Y, Pos_Theta
Pos_X = 0
Pos_Y = 0
Pos_Theta = 0
```

```
def SyncCallback_Map(in_Scan,in_Odom):
    global CoordX, CoordY, Pos_Theta
    global Angles
    global ScanRanges
    global Pos_X, Pos_Y, Pos_Theta

    ScanRanges = np.array(in_Scan.ranges) #Get the scan measurements
    RangeMin = in_Scan.range_min #Get the limits of the measurements
    RangeMax = in_Scan.range_max

    #Get the pos measurements
    Pos_X = in_Odom.pose.pose.position.x
    Pos_Y = in_Odom.pose.pose.position.y
    Quad = in_Odom.pose.pose.orientation
    #Finf the euleur angle from the quaternion
    QuadList = [Quad.x, Quad.y, Quad.z, Quad.w]
    Pos_Theta=euler_from_quaternion(QuadList,)[2] #0:Roll, 1:Pitch, 2:Yaw

    #Keep only the valid measurements i.e. those that are in the confidence range according to the
manufacturer
    ValidScansId = np.asarray(np.where((ScanRanges>RangeMin)&(ScanRanges<RangeMax)))[0]

    #Compute the real position of the measured points
    Phi = Pos_Theta + Angles[ValidScansId]
    Point_X = Pos_X + ScanRanges[ValidScansId] * np.cos(Phi)
    Point_Y = Pos_Y + ScanRanges[ValidScansId] * np.sin(Phi)

    #Find in which cell of the map do they fall without duplicates
    cell_id_x = np.argmin(np.abs(CoordX - Point_X.reshape(-1, 1)), axis=1)
    cell_id_y = np.argmin(np.abs(CoordY - Point_Y.reshape(-1, 1)), axis=1)
    #Increment the cells value
    Map[cell_id_y,cell_id_x] += 1
```

There is one main criticism that can be made about the method of creating the map, it is its lack of noise robustness. Indeed the position of each point can vary a bit through time (for a given position and orientation of the robot) thus resulting in a low-precision map. In order to compensate for this lack of robustness one could convolute a map with only the last computed points (M_d) with a mask that compensates for the noise such as a 2D-Gaussian (M_s). The resulting map (M^*) could then be included with the actual map. The last step could be an addition, a multiplication or something more complicated such as using Bayes' law to reduce the noise with time.

The noise is not taken into account in the node. However, for sharing the map, a saturation technic is used to reduce the uncertainties as explained in the following section.

2.3. Sharing the map

The most important criterion when dealing with a map is the ability to tell if one place is reachable or not. In other words, is the robot trying to pass through a wall? The simplest way to quantify this is to attribute a value between 0 and 1 where 0 means that there is no obstacle and 1 that there is a wall.

As explained in the previous section, the constructed map has no upper bound for the values stored in its cells thus making it impossible to take into account the noise. In order to solve this problem a very simple solution has been chosen. The core idea is to define a threshold representing the minimum number of measurements in one cell to be able to tell with high confidence that the cell is a wall. The algorithm is detailed below in pseudo-code:

```
#Assuming that there is no drift in the measurements
#If a cell has accumulated 20 or more measurements we are sure that there is a wall.
Threshold = 20
#The map is normalised. 0=no wall ; 1>= wall
MapScaled = Map / Threshold
#A value above 1 or above the threshold does not contain additional information so it is cut.
MapScaled[MapScaled>=1] = 1 #Alternatively: MapScaled[Map>=Threshold] = 1;
```

By doing so the only remaining noise is relative to the map's cells which from the beginning were already noisy.

One criticism that can be made of this technique is that if the robot spends a long enough time period in an area, the noisy measurements will accumulate around the "real shape" and end up reaching the threshold. Thus making the final shape bigger than it actually is.

The second most important criterion when dealing with a map is the ability to correctly interpret it. Indeed, in memory, the map is just a 1D array of values with no additional information thus this missing information must be provided. First, come the dimensions of the map in the physical world given by its extreme values: xmin, ymin, xmax and ymax. Then comes the resolution of the map along each dimension: stepx, stepy (called epsilon in the codes above). In order to have a better understanding of the measurement the value of the threshold (also called level) can also be included resulting in the following fields to be shared.

```
"xmin": #left side of the map in meters
"ymin": #bottom side of the map in meters
"xmax": #right side of the map in meters
"ymax": #top side of the map in meters
"stepx": #x-axis resolution in meters
"stepy": #y-axis resolution in meters
"level": #saturation value
"data": #array with all the map's cells values
```

It exists multiple file formats that can store this information. An intuitive approach would be to choose a picture file format such as JPEG, PNG or GIF [4]. However, JPEG compresses the data thus part of the information is lost and all of them would have to store the non-map data in its metadata (e.g. the tEXt chunk for PNG). It is not in itself a problem but I wish to have a more transparent storage format. JSON and XML are standard data exchange formats making them more appropriate for the task. The JSON format was selected for its simplicity over XML.

The result looks like this:

```
{
    "xmin": -3,
    "xmax": 3,
    "ymin": -3,
    "ymax": 3,
    "stepx": 0.015,
    "stepy": 0.015,
    "level": 10,
    "data": [
        0.1408478733304343,
        0.14084787333043428,
        0.14084787333043422,
        0.14084787333043422,
        ...
        0.14613647769271879,
        0.12196409961942059,
        0.29489240868472

    ]
}
```

An obvious criticism that can be made is that the size of the file will increase in $O(n^2)$ when increasing either the size of the map or its precision. Possible solutions are to only store the non-zero values (e.g. stocking the associated coordinates or the cell unique ID or using a B-tree data structure), using beforehand a shape detection algorithm to extract the geometric data (then stored in an XML or an SVG file) or instead of storing strings as it is the case with the JSON to first map float: [0;1] onto uint8_t: [0;255] which takes 4 times less size and to store in another file format.

To conclude the two previous sections, here is the flowchart of the ROS node (the purple blocks are for debugging purposes):

[4] Bitmaps are not included despise being the most basic raster format because they only contains booleans which contradicts the first requirement.

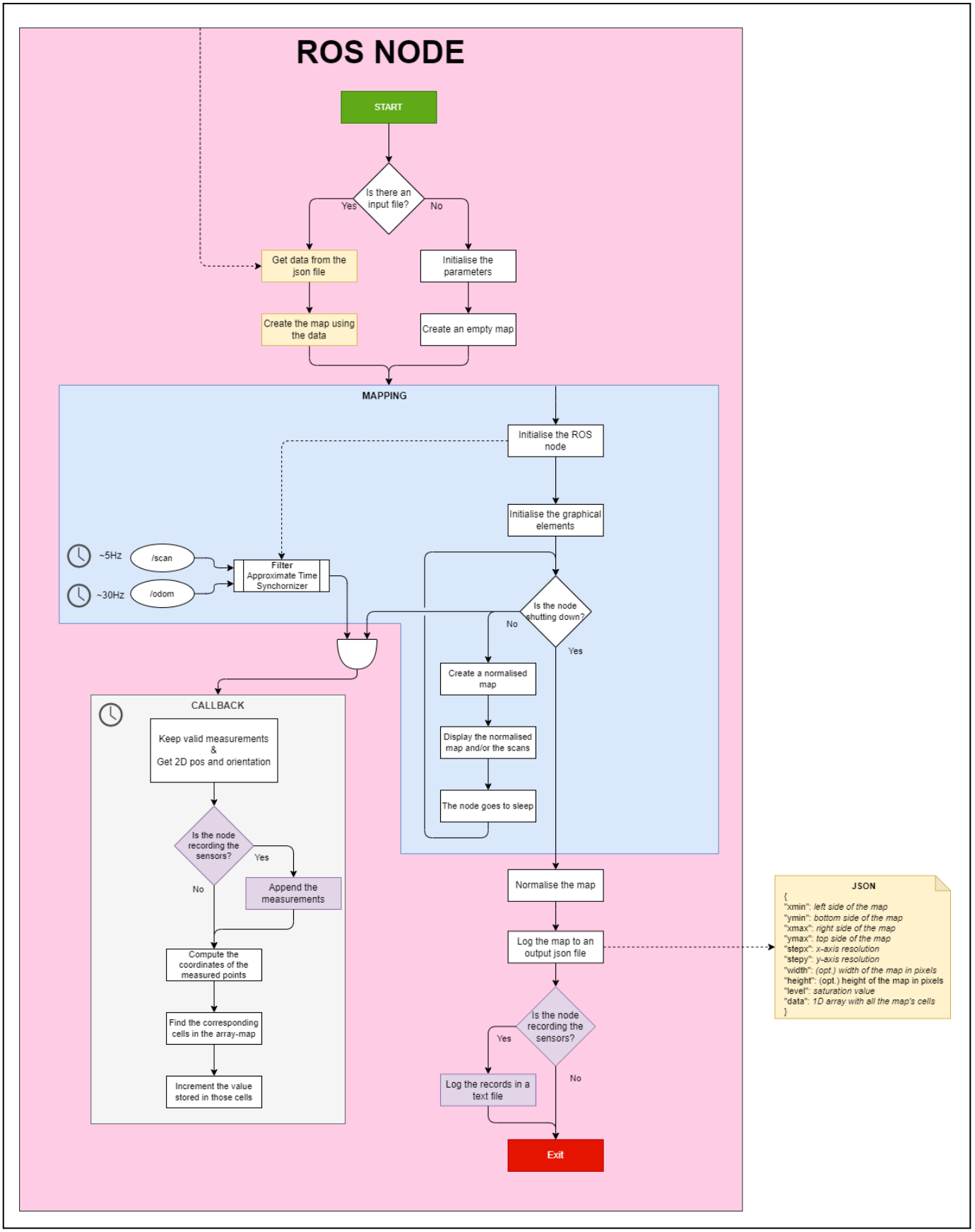


Figure 2: ROS node's flowchart

2.4. Edit the map

The goal of this part is to create a graphical user interface (GUI) that reads the generated map (JSON file) in the previous section and allows the user to easily edit the map and then share the result with the same method explained previously. Thus the sub-requirements for this part are

3.1. to read the JSON and recreate the map;
3.2. to display the map;
3.4. being able to add and erase walls (binary choice);
3.5. being able to edit the map precisely;
3.6. to save the revised map as a JSON following the same rules as in the ROS node.

The third sub-requirement is justified because it does not make a lot of sense to add or erase noise given the noise reduction applied in the previous section.
The fourth sub-requirement is justified by the fact that a map can be composed of elements of various scales so it does make sense to be able to edit at those various scales.

There are a lot of tools to create a GUI, however since I never created a GUI before I chose to do it using Python as a follow-up to the ROS node. Python has a dedicated GUI creator module called Tkinter [5].

For this GUI, three graphical elements are needed: a canvas to draw, one or more labels to display information and a button to save the result. To each of those graphical elements, one or more callback functions are associated.

The editing is done only using the mouse. The right button allows adding walls while the left button allows adding emptiness (i.e. removing walls). The mouse wheel adjusts the diameter of the drawn circle from 0 to 100 canvas pixels by increments of 1. When pressed, the “Save map” button (see Figure 3) saves the map into a JSON file. Figure 3 shows side by side the GUI at its launch and while or after editing.

[5] https://docs.python.org/3/library/tk.html

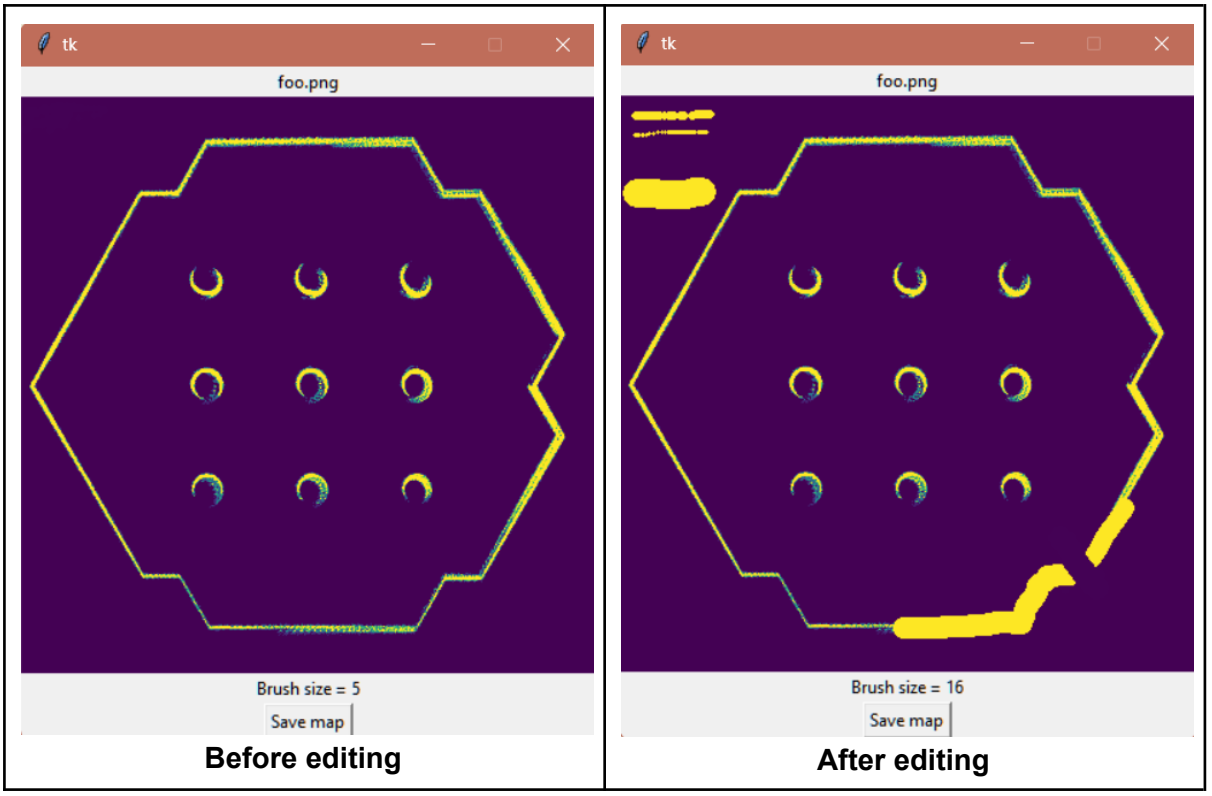


Figure 3: Editing a map

The Tkinter module does not provide a method to pre-draw the map making it impossible or very slow to convert an array into a canvas drawing. However, it is possible to add a background image. Once the array is created from the JSON file, it is converted into a temporary PNG file that is added to the canvas as a background image. Unfortunately, the Tkinter module also does not provide a method to save the canvas as an array of an image. To do so the method is to take a screenshot of the canvas. This adds extra steps to the saving process. Screenshots can be made using the ImageGrab module from the PIL library [6]. ImageGrab does not capture an area defined by distances proportional to the height and width of the screen as it is usually done in drawing applications but rather captures using the coordinates of screen pixels. This slight variation makes it unable to take into account the screen zoom level that many people use to compensate for the screen size. This scale factor can be accessed using the `GetScaleFactorForDevice` function from the ctypes [7] library. Using this scale factor and the coordinates of the GUI and its components given by Tkinter it is possible to precisely take a screenshot of the canvas. The screenshot is then converted to a 2D array.

```
#get the screen scale factor: 100% 125% 150% 200% on Windows
#there is an invisible border around each side so +/-1 on each side
scaleFactor = ctypes.windll.shcore.GetScaleFactorForDevice(0) / 100
x = (widget.window.winfo_rootx() + widget.canvas.winfo_x() +1) * scaleFactor
y = (widget.window.winfo_rooty() + widget.canvas.winfo_y() +1) * scaleFactor
width = (widget.canvas.winfo_width() -1) * scaleFactor
height = (widget.canvas.winfo_height() -1) * scaleFactor
box = (x,y,x+width,y+height)
```

[6] https://pillow.readthedocs.io/en/stable/reference/ImageGrab.html
[7] https://docs.python.org/3/library/ctypes.html

```
    #Take the screenshot
    img_edited = ImageGrab.grab(bbox = box)
    #Keep only the Luminance: RGB to Gray (L = R * 299/1000 + G * 587/1000 + B * 114/1000)
    img_edited = img_edited.convert(mode='L')
    #Normalise the result
    new_map = np.array(img_edited)/255.0
```

However, because while editing the map colours are used when turning the screenshot into a greyscale image the “wall colour” does not correspond to white(see the formula in the code snippet above). The highest value thus depends on the colour map used to display the map. The same is true for the “empty colour”. Another issue is that depending on the screen resolution, the resulting array may not have the right dimensions as the original one. To solve these two problems the next few lines of code are added to compensate and correct the distortions:

```
    # because of the scaling factor, the captured image does not have the same dimension as the
original
    # thus a resampling is done
    x_zoom = source_map.width/new_map.shape[0]
    y_zoom = source_map.height/new_map.shape[1]
    new_map = scipy.ndimage.zoom(new_map,(x_zoom,y_zoom))
    #normalisation
    new_map = new_map - np.amin(new_map)
    new_map = new_map/np.amax(new_map)
    # When the screenshot is done, the colours are dimmed, and a threshold is applied.
    # In the original picture where there was a 1 is now a 0.65 (measured by hand)
    # Assuming everything has been dimmed by the same factor
    # all values are scaled by 1/0.65 and clamped if they are above 1.
    new_map = new_map*(1/0.65)
    new_map[new_map>1] = 1
```

Both of these corrections have the effect of stretching the noise in the spatial space (interpolation done by the resampling) and in the value space (because of the shift and the scaling).

The resulting map is then turned into a JSON file using the parameters from the input map.

The figure Figure 4 shows the flow chart of the program.

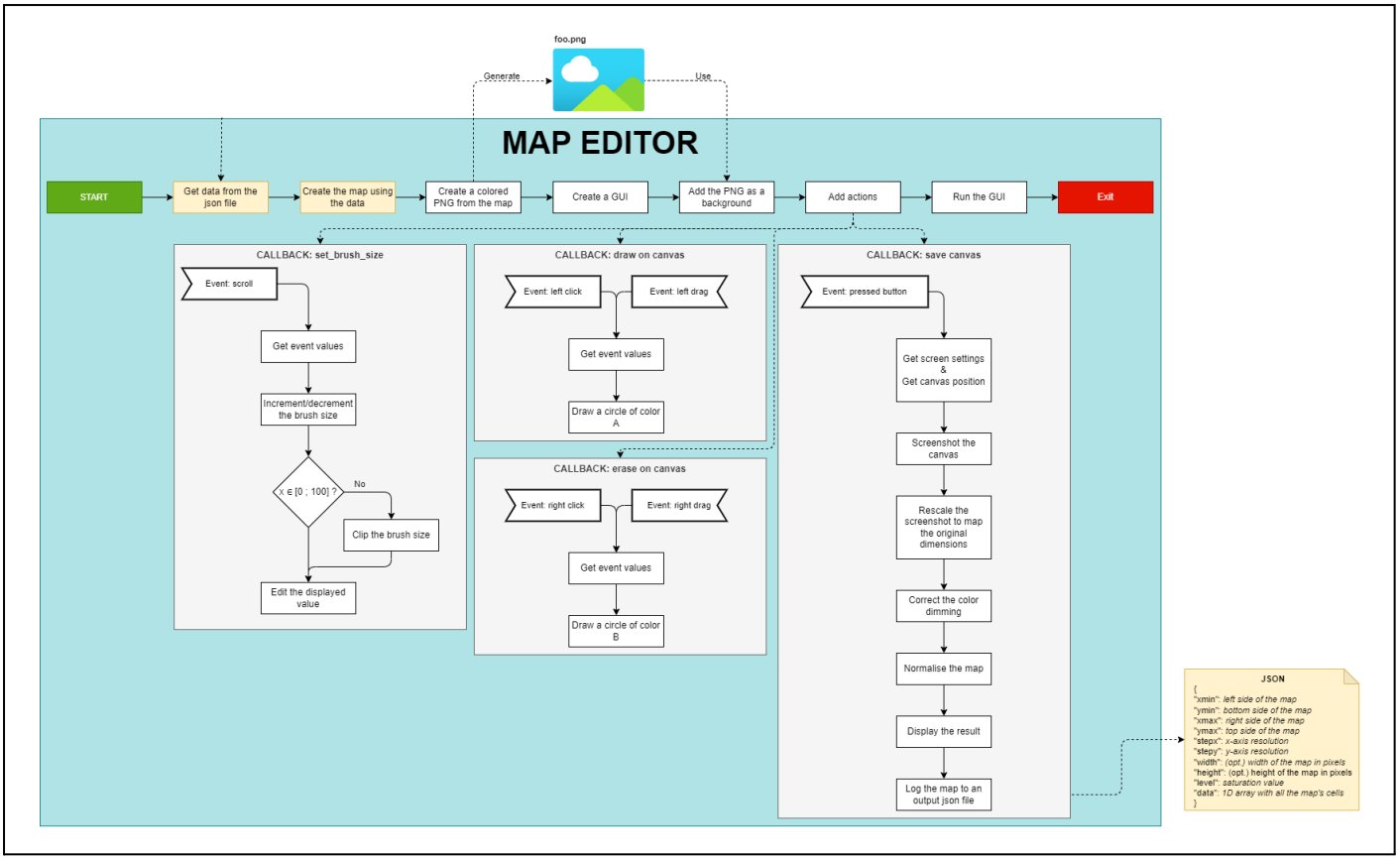


Figure 3: Map editor's flowchart

3. Conclusion

To conclude, the three requirements have been achieved: I have create a more noise-resistant map, this map can be easily shared between users or machines and I have created a graphical user interface to edit the maps. To resume the various criticisms in a few points: the map is not highly noise-resistant on the long term, the size of the map sharing file can be optimised on various levels and the editing distords the map. In order to solve these various problems many technical solutions have been proposed with different complexity levels and efficiency.

From a personal perspective, this project took me around 60 hours across 4 weeks from learning the basics of Python to the writing of this sentence. It fullfills two wishes of mine: learning Python and improving what I considered an unfinished academic project. It also gave me the opportunity to look into various related topics from data structures to file encoding that would definitively inspire me to redo an improved version of the project.