Obstacle Avoidance Algorithm

Obstacle Detection

High Level Overview

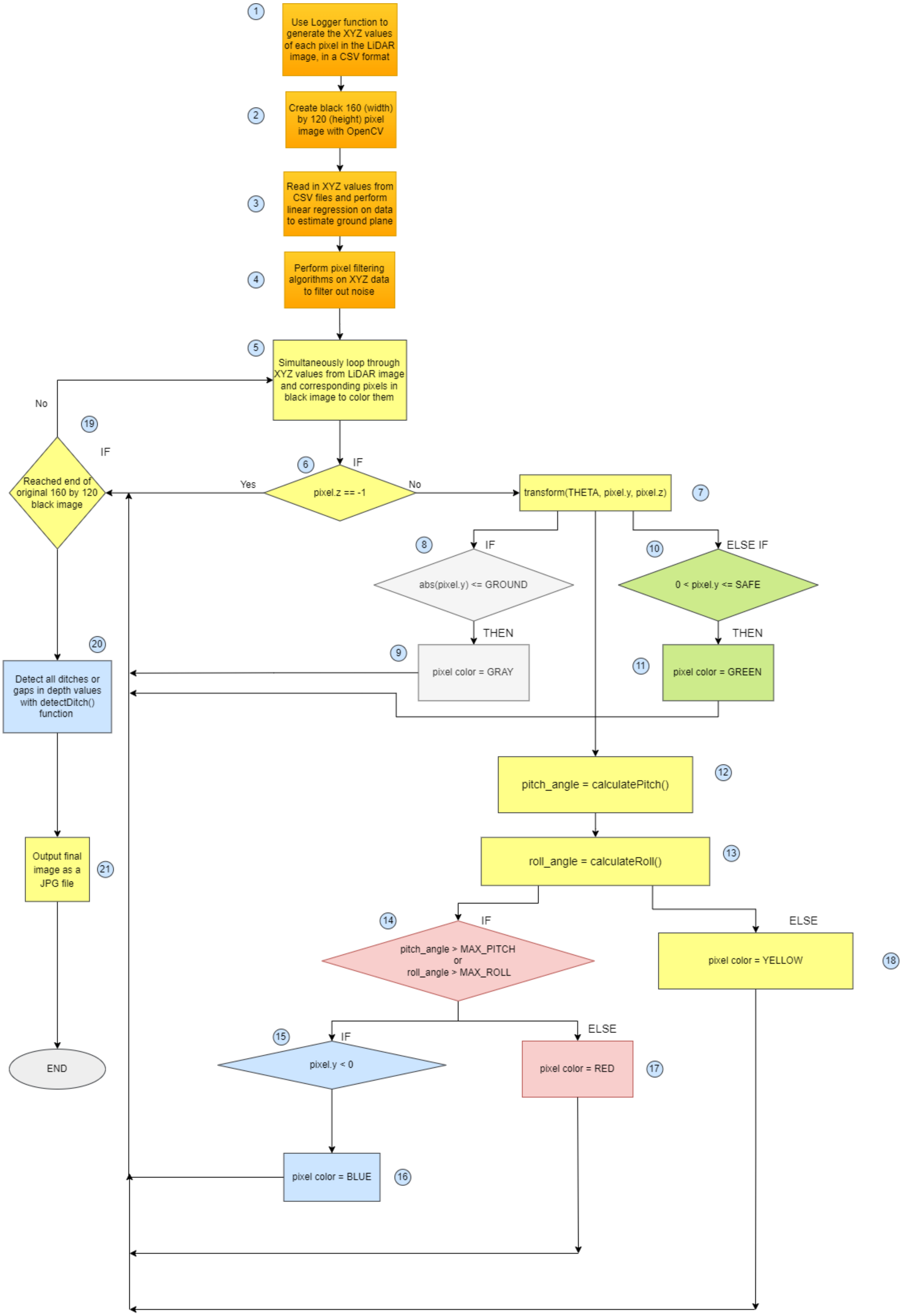
My algorithm is designed under the assumption that the LiDAR is mounted at the centre, highest point on the front face of the Rover. A slope-based obstacle detection algorithm is used to detect and classify obstacles based on the slope of their surface. This algorithm transforms the XYZ coordinates of each detected pixel in a 160 by 120 LiDAR image generated by the ToF sensor, from the ground's reference frame to the LiDAR's reference frame, before calculating the pixel's slope with reference to a nearby pixel in the same column of the LiDAR image.

The algorithm starts by creating a black 160 by 120 pixel image. Then the XYZ values are read in from the CSV files and linear regression is performed on this data to estimate the ground plane. After being read in, any noise (random error) in the XYZ data is filtered out with pixel filtering algorithms such as the median filter, gaussian filter, and bilateral filter. Then, the XYZ values of each pixel on the 160 by 120 pixel LiDAR image are iterated through simultaneously with the corresponding pixels in the black image to color these black pixels based on a few conditions.

If the pixel has a valid depth value detected by the LiDAR, the pixel is transformed from the ground's reference frame to the LiDAR's reference frame. The pixel's slope with reference to a nearby pixel (in the same column), is used to calculate its pitch and roll Euler angles. Based on these angles, the pixel will then be assigned a specific color (RED = Large Pitch or Large Roll, YELLOW = Warning Zone, GREEN = Safe Zone, and BLUE = Ditch) where red represents a large slope that should not be driven over, yellow represents an area or slope that should be driven over with caution, and green represents an area or obstacle that can be safely traversed without worry. If a pixel below the ground has a large pitch or roll angle, it will be colored navy blue to represent a ditch.

If the pixel's absolute value height from the ground is below 100 mm, it will be colored gray to represent the ground or resemble an occupancy grid indicating obstacle-free zones the operator could drive the Rover towards. The pixels in the LiDAR image that have no depth value, will remain black in the 160 by 120 pixel black image. The detectDitch() function will then color all pixels that could represent ditches or gaps in depth value between two objects in the LiDAR image blue. Finally, the final image will be outputted as a JPG file.

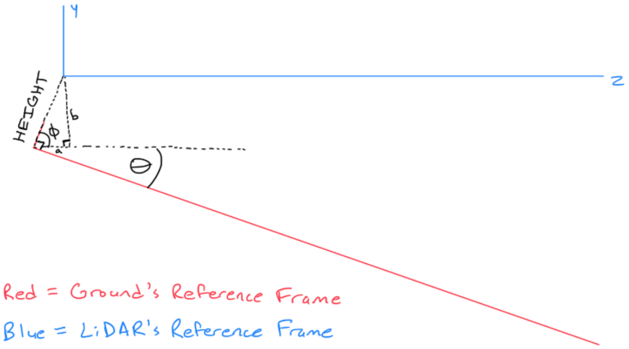
Here is a flowchart representation of the algorithm (the flowchart is in the ***Flowcharts*** folder as well):



1
Use Logger function to generate the XYZ values of each pixel in the LiDAR image, in a CSV format
2
Create black 160 (width) by 120 (height) pixel image with OpenCV
3
Read in XYZ values from CSV files and perform linear regression on data to estimate ground plane
4
Perform pixel filtering algorithms on XYZ data to filter out noise
5
Simultaneously loop through XYZ values from LiDAR image and corresponding pixels in black image to color them
6
IF
pixel.z == -1
Yes
No
7
transform(THETA, pixel.y, pixel.z)
8
IF
abs(pixel.y) <= GROUND
THEN
9
pixel color = GRAY
10
ELSE IF
0 < pixel.y <= SAFE
THEN
11
pixel color = GREEN
12
pitch_angle = calculatePitch()
13
roll_angle = calculateRoll()
14
IF
pitch_angle > MAX_PITCH
or
roll_angle > MAX_ROLL
ELSE
18
pixel color = YELLOW
15
IF
pixel.y < 0
ELSE
17
pixel color = RED
16
pixel color = BLUE
19
IF
No
Reached end of original 160 by 120 black image
20
Detect all ditches or gaps in depth values with detectDitch() function
21
Output final image as a JPG file
END

Here is a step-by-step explanation of each block of the algorithm:

1. Use the Logger function in the Sentis ToF API to generate the XYZ values of each pixel in the LiDAR image, in a CSV format.
2. Use OpenCV to create a black 160 (width) by 120 (height) pixel image.
3. Read in the XYZ values from the CSV files and perform linear regression on this data to estimate the ground plane.
4. Perform pixel filtering on XYZ data using filters such as median filter, gaussian filter, and bilateral filter.
5. Simultaneously iterate through the XYZ values from the LiDAR image and the corresponding pixels in the 160 by 120 pixel black image to color these corresponding pixels.
6. Check if the pixel from the LiDAR image has a valid depth value (z value of -1 means invalid depth value).
7. Since from the LiDAR's point-of-view the ground is tilted upwards by a positive angle even though the ground is completely flat in real life, this means that the LiDAR's reference frame is tilted upwards (meaning every point in the LiDAR's point-of-view is also tilted upwards) from the ground's reference frame by this angle. If the pixel has a valid depth value (z value != -1), we plug in THETA (the angle by which the ground's reference frame will be rotated about the x-axis to become parallel with the LiDAR's reference frame), the pixel's y value, and the pixel's z value into the transform() function which transforms the pixel from the ground's reference frame to the LiDAR's reference frame. See drawing below for explanation. The angle THETA was determined using the Linear Regression algorithm described later in this document.



Red = Ground's Reference Frame

Blue = LiDAR's Reference Frame

HEIGHT = Vertical height from LiDAR lens to ground

θ = Rotation angle between ground and LiDAR reference frames

$\phi = 90^\circ - \theta$

$a = \text{HEIGHT} \cos(\phi)$ $b = \text{HEIGHT} \sin(\phi)$

To tranform ground reference frame to LiDAR reference frame:

1) pixel.z += a

2) pixel.y += b

3) Rotate pixel by positive θ (positive is counter-clockwise)

8. After the transformation, if the pixel's absolute value height from the ground is below or equal to 100 mm (GROUND), the pixel can be considered a part of the ground.
9. Since the pixel can be considered ground, it will be colored gray. Continue to the next iteration of the algorithm.
10. If the pixel's height from the ground is positive and less than or equal to 150 mm (SAFE), the pixel can be considered "safely traversable without worry".

11. Since the pixel can be considered “safely traversable without worry”, it will be colored green. Continue to the next iteration of the algorithm.
12. Call the calculatePitch() function to calculate the pitch angle of the current pixel with reference to another pixel (with a valid depth value) that’s 3 pixels below it and in the same column of the LiDAR image.
13. Calculate the roll angle of the current pixel using the calculateRoll() function which (by searching columns of the LiDAR image from top-to-bottom) finds two reference pixels. One reference pixel is the first pixel that is located a half rover width to the left of the current pixel (in 3D space) and has a similar depth value to the current pixel, and the second reference pixel is the first pixel that is located a half rover width to the right of the current pixel (in 3D space) and has a similar depth value to the current pixel. The angle between these two reference pixels is calculated to determine the roll angle. If no reference pixel a half rover width to the left of the current pixel has a similar depth value to the current pixel, the algorithm searches inwards towards the current pixel (searches the next column of the LiDAR image until an appropriate left reference pixel is found or the column of the current pixel has been reached). Same thing applies for the right reference pixel (if an appropriate right reference pixel isn’t found, the algorithm searches inwards towards the column of the current pixel).
14. If the pitch angle is greater than the Max Pitch threshold or the roll angle is greater than the Max Roll threshold, the pixel will be considered “dangerous” indicating that the Rover should not attempt to drive over it.
15. If the pixel exceeds the Max Pitch or Max Roll thresholds, check if it’s below the ground.
16. If the pixel exceeds the Max Pitch or Max Roll thresholds and it’s below the ground, color it blue as it represents a ditch. Continue to the next iteration of the algorithm.
17. If the pixel exceeds the Max Pitch or Max Roll thresholds and it’s above the ground, color it red as it represents a large slope that cannot be traversed. Continue to the next iteration of the algorithm.
18. Else, in the case that the pixel doesn’t exceed the Max Pitch or Max Roll thresholds, it will be considered “safe but proceed with caution” and will be colored yellow. Continue to the next iteration of the algorithm.
19. Check if the algorithm has reached the end of the original 160 by 120 black image and continue iterating through the rest of the pixels in the 160 by 120 LiDAR image if the end has not been reached.
20. If the end of the image has been reached, call the detectDitch() function to color all pixels that represent ditches or gaps in depth value between objects blue. The detectDitch() algorithm will be explained in more detail later in this document.
21. Output the final image as a JPG file.

Estimation of Ground Plane Using Linear Regression with Method of Least Squares

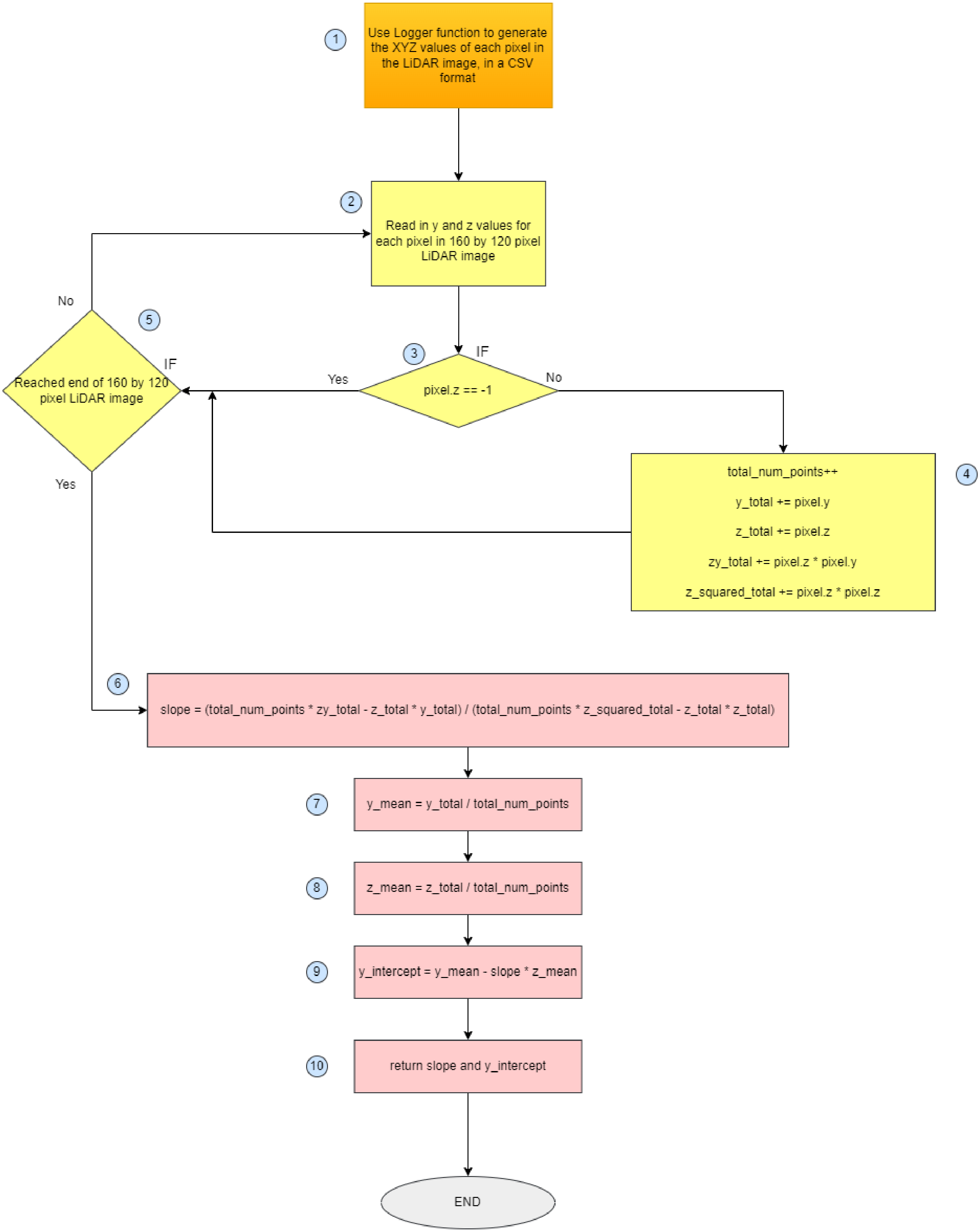
This section explains the algorithm used to determine the angle THETA that represents the angle by which the ground's reference frame will be rotated to become parallel with the LiDAR's reference frame. For now, let's assume the LiDAR is upright (has a 0° angle of inclination). First, we need to estimate the ground plane using linear regression with the method of least squares to determine the slope of a completely flat ground plane as it appears in the LiDAR's point-of-view.

To determine the slope of a perfectly flat ground plane as it appears in the LiDAR's point-of-view, I placed the LiDAR upright on the ground on a completely flat surface where only this surface and no other obstacles or objects were present in the LiDAR's field-of-view. I then used the Logger function in the Sentis ToF API to generate the XYZ values of each pixel in this field-of-view.

Next, I created the linearRegression() function to read in these XYZ values from CSV files and perform linear regression on the y and z values of the pixels that have a valid depth value (the pixels with valid depth values are the pixels detected by the LiDAR that represent the ground plane), with the method of least squares.

The linearRegression() function outputs the slope and y-intercept of a line on this ground plane. The slope of this line represents the slope of a perfectly flat ground plane as it appears in the LiDAR's point-of-view. By taking the inverse tan of this slope and converting to degrees, we can determine the angle OMEGA by which the ground's reference frame must be rotated to become parallel with the LiDAR's reference frame when the LiDAR is upright. If not upright, we add the LiDAR's angle of inclination to OMEGA, to get THETA.

Here is a flowchart of the linearRegression() algorithm (the flowchart is in the ***Flowcharts*** folder as well):


1
Use Logger function to generate the XYZ values of each pixel in the LiDAR image, in a CSV format
2
Read in y and z values for each pixel in 160 by 120 pixel LiDAR image
3
IF
pixel.z == -1
Yes
No
4
total_num_points++
y_total += pixel.y
z_total += pixel.z
zy_total += pixel.z * pixel.y
z_squared_total += pixel.z * pixel.z
5
IF
Reached end of 160 by 120 pixel LiDAR image
No
Yes
6
slope = (total_num_points * zy_total - z_total * y_total) / (total_num_points * z_squared_total - z_total * z_total)
7
y_mean = y_total / total_num_points
8
z_mean = z_total / total_num_points
9
y_intercept = y_mean - slope * z_mean
10
return slope and y_intercept
END

Here is a step-by-step explanation of each block of this algorithm:

1. Use the Logger function in the Sentis ToF API to generate the XYZ values of each pixel in the 160 by 120 pixel LiDAR image, in a CSV format.
2. Read in the y and z values for each pixel in the 160 by 120 pixel LiDAR image.
3. Check if the pixel has a valid depth value (z value of -1 means invalid depth value).
4. If the pixel has a valid depth value (z value != -1), add 1 to total_num_points (the total number of points on the ground plane that have been detected by the LiDAR), add the pixel's y value to y_total, add the pixel's z value to z_total, add the product of the pixel's z and y value to zy_total, square the pixel's z value before adding it to z_squared_total.
5. Check if the algorithm has reached the end of the 160 by 120 pixel LiDAR image and continue iterating through the rest of the pixels if the end has not been reached.
6. If the end of the LiDAR image has been reached, calculate the slope of a line on the ground plane.
7. Calculate the mean average of all the y values.
8. Calculate the mean average of all the z values.
9. Calculate the y-intercept of the line on the ground plane.
10. Return the slope and y-intercept of the line.

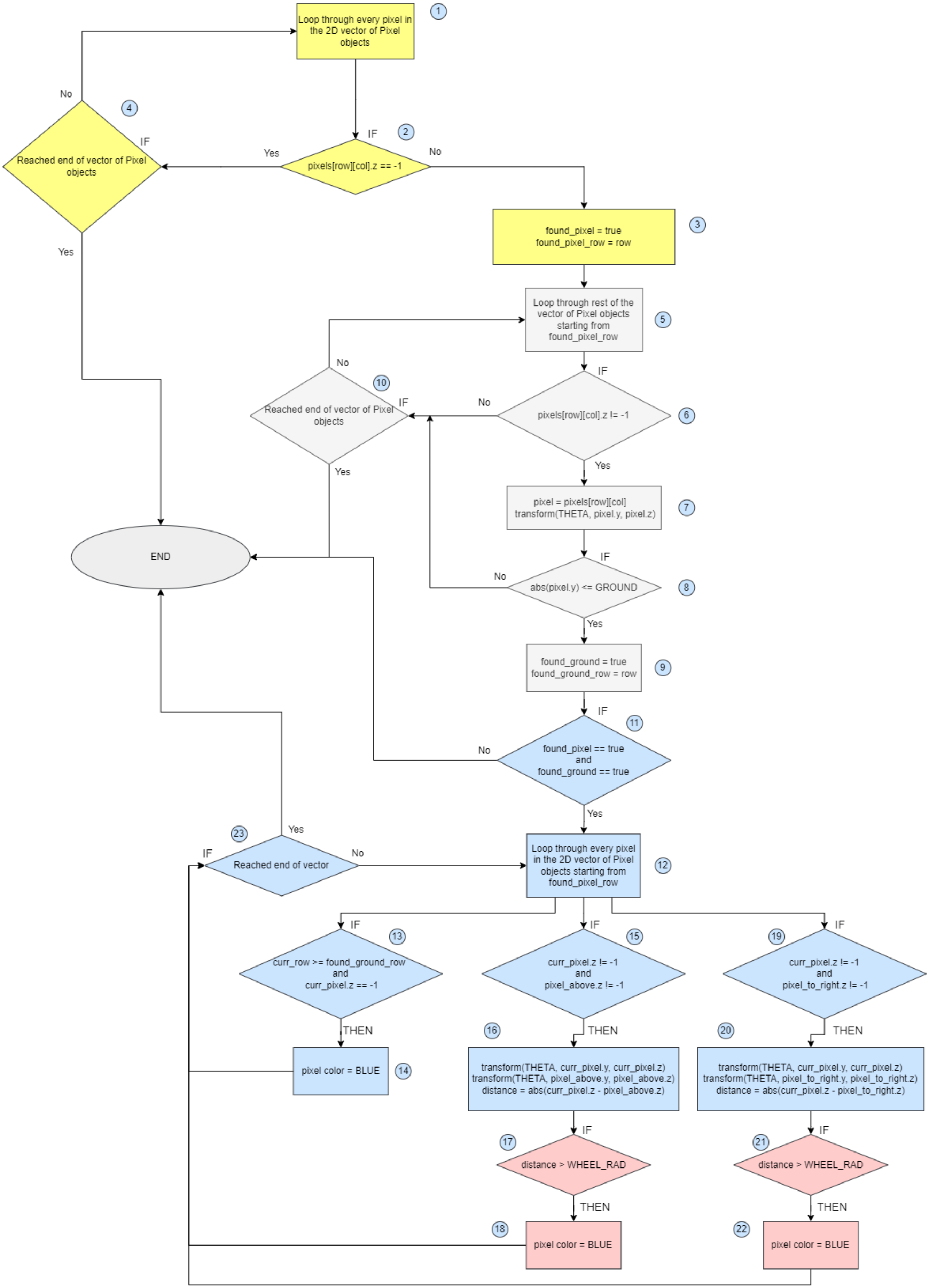
Algorithm to Detect Ditches and Depth Value Gaps

This section explains the detectDitch() function which colors all pixels that represent ditches or gaps in depth value between objects blue. The detectDitch() function takes in an image to perform the ditch detection on and a 2D vector of Pixel objects that represents all the pixels in the 160 by 120 pixel LiDAR image. The algorithm then loops through every pixel in the 2D vector of Pixel objects until a pixel with a valid depth value is found. Once a pixel with a valid depth value is found, the function stores the row of this pixel. Next, the function loops through the rest of the vector of Pixel objects searching for the first pixel on the ground that it finds and stores the row of this pixel.

Starting from the row of the first pixel with a valid depth value that was found, the algorithm loops through the pixels in the rest of the vector. Since the function had also saved the row of the first pixel on the ground that was found, every pixel in the vector that's below this row and has an invalid depth will be colored blue (every pixel on the ground that has an invalid depth can be considered a ditch).

Otherwise, the algorithm compares the depth values of the pixel the loop is currently on to the pixel directly above it and the pixel directly to its right to determine if there is a significant gap in depth value (a depth value gap greater than the radius of the Rover's wheels). If there is a significant gap, the algorithm colors the current pixel blue to indicate to the operator that there may be a crevasse at this point or that it could represent the edge of an obstacle that's in-line with another obstacle in the LiDAR's field-of-view.

Here is a flowchart representation of the algorithm (the flowchart is in the ***Flowcharts*** folder as well):


1
Loop through every pixel in the 2D vector of Pixel objects
IF
2
pixels[row][col].z == -1
Yes
No
4
IF
Reached end of vector of Pixel objects
No
Yes
3
found_pixel = true
found_pixel_row = row
5
Loop through rest of the vector of Pixel objects starting from found_pixel_row
IF
6
pixels[row][col].z != -1
No
Yes
10
IF
Reached end of vector of Pixel objects
No
Yes
7
pixel = pixels[row][col]
transform(THETA, pixel.y, pixel.z)
IF
8
abs(pixel.y) <= GROUND
No
Yes
END
9
found_ground = true
found_ground_row = row
IF
11
found_pixel == true
and
found_ground == true
No
Yes
23
IF
Reached end of vector
Yes
No
12
Loop through every pixel in the 2D vector of Pixel objects starting from found_pixel_row
IF
13
curr_row >= found_ground_row
and
curr_pixel.z == -1
THEN
14
pixel color = BLUE
IF
15
curr_pixel.z != -1
and
pixel_above.z != -1
THEN
16
transform(THETA, curr_pixel.y, curr_pixel.z)
transform(THETA, pixel_above.y, pixel_above.z)
distance = abs(curr_pixel.z - pixel_above.z)
IF
17
distance > WHEEL_RAD
THEN
18
pixel color = BLUE
IF
19
curr_pixel.z != -1
and
pixel_to_right.z != -1
THEN
20
transform(THETA, curr_pixel.y, curr_pixel.z)
transform(THETA, pixel_to_right.y, pixel_to_right.z)
distance = abs(curr_pixel.z - pixel_to_right.z)
IF
21
distance > WHEEL_RAD
THEN
22
pixel color = BLUE

Here is a step-by-step explanation of each block of this algorithm:

1. Loop through every pixel in the 2D vector of Pixel objects, searching for the first pixel with a valid depth value.
2. Check if the pixel has a valid depth value (z value of -1 means invalid depth value).
3. If the pixel has a valid depth value, set the found_pixel bool variable to true to indicate that a pixel with a valid depth has been found and set the found_pixel_row variable to the current row to remember the row of this pixel.
4. If the pixel did have an invalid depth value, check if the end of the vector of Pixel objects has been reached. If it has been reached, the algorithm ends. If it hasn't been reached, continue to the next iteration.
5. Once a pixel with a valid depth value has been found, loop through the rest of the vector of Pixel objects starting from found_pixel_row, the row of the first pixel with a valid depth value that was found.
6. Check if the pixel has a valid depth value (z value of -1 means invalid depth value).
7. If the pixel has a valid depth value, transform it from the ground's reference frame to the LiDAR's reference frame with the transform() function.
8. Check if the pixel is on the ground (has an absolute value height less than or equal to GROUND).
9. If the pixel is on the ground, set the found_ground bool variable to true to indicate that a pixel on the ground with a valid depth has been found and set the found_ground_row variable to the current row to remember the row of this pixel.
10. If the pixel did have an invalid depth value, check if the end of the vector of Pixel objects has been reached. If it has been reached, the algorithm ends. If it hasn't been reached, continue to the next iteration.
11. Check if the found_pixel and found_ground bool variables are both true. If they aren't both true, the algorithm ends.
12. If both found_pixel and found_ground are true start from found_pixel_row, the row of the first pixel with a valid depth value that was found, and loop through the rest of the vector of Pixel objects.
13. Check if the current row is below found_ground_row and if the current pixel has an invalid depth value.
14. If the current pixel has an invalid depth value and the row it's in is below found_ground_row, it means the pixel is on the ground has no detected depth value so it will be considered a ditch and will be colored blue.
15. Check if the current pixel and the pixel directly above it both have valid depth values.
16. If the current pixel and the pixel directly above it both have valid depth values, transform both of these pixels from the ground's reference frame to the LiDAR's reference frame. Then calculate the absolute value z distance between both pixels.
17. Check if the calculated distance is greater than the radius of the Rover's wheels.
18. If the calculated distance is greater than the radius of the Rover's wheels, there is a significant depth gap between both pixels and therefore the current pixel will be colored blue to indicate this gap.
19. Check if the current pixel and the pixel directly to its right both have valid depth values.
20. If the current pixel and the pixel directly to its right both have valid depth values, transform both pixels from the ground's reference frame to the LiDAR's reference frame. Then calculate the absolute value z distance between both pixels.

21. Check if the calculated distance is greater than the radius of the Rover’s wheels.
22. If the calculated distance is greater than the radius of the Rover’s wheels, there is a significant depth gap between both pixels and therefore the current pixel will be colored blue to indicate this gap.
23. Check if the end of the vector of Pixel objects has been reached. If it has been reached, the algorithm ends. If it hasn't been reached, continue to the next iteration.

Obstacle Avoidance/Path Planning

Possible Obstacle Avoidance Algorithms

- Bug 1 Algorithm
- Bug 2 Algorithm
- Artificial Potential Field Method
- Vector Field Histogram
- A* Algorithm
- Bubble Band Technique

Descriptions of Avoidance Techniques + Pros & Cons

Bug 1 Algorithm

The Bug 1 algorithm is the simplest algorithm among those discussed in this document and provides high reliability by reaching the goal almost every time. However, its efficiency is a concern as the robot moves along the shortest path from its position X to the goal location until it encounters an obstacle. Upon encountering the obstacle, the robot revolves around it and calculates the distance from the destination. After a full revolution, it determines the closest departure point to the goal and adjusts its direction of motion based on the distance between the departure point and the point of impact. This method can be described in three steps: moving towards the goal, circling around obstacles while remembering proximity to the goal, and returning to the closest point before continuing the journey. Despite increasing computational effort by circling around each obstacle, the ease of implementation makes the Bug 1 algorithm useful when task completion is the only priority, regardless of time.

Pros:

- **This algorithm either finds a path in finite time if it exists, or terminates with failure if no such path exists.**

Cons:

- **Bug 1 requires significant memory and computational resources, leading to lower efficiency.**
- **It utilizes an exhaustive search method.**
- **It is very time-consuming.**

Bug 2 Algorithm

The Bug 2 algorithm is a revision of the Bug 1 algorithm, prioritizing direction towards the goal rather than seeking the minimum distance point. The slope of the line connecting the starting point and the target point is calculated. If the robot encounters an obstacle, it moves along the edge until a point with the same slope is found and then resumes moving towards the goal. This algorithm is more efficient than the Bug 1 algorithm in most scenarios, as it does not require the robot to go around the obstacle completely. It is also easier to program.

Pros:

- **This is a greedy search algorithm, which selects the first option that appears to be the best.**
- **It has low computational requirements and is simple to implement.**

Cons:

- **In very complicated obstacle avoidance scenarios with numerous obstacles, it sometimes fails.**

Artificial Potential Field

The Artificial Potential Field (APF) method involves the assumption that the robot and obstacles have the same electric potential, while the goal has an opposite potential. This causes the robot to be repelled by obstacles and attracted to the goal. The velocity of the robot decreases as it gets closer to the goal and stops when the distance is zero. However, this method can fail if the robot reaches a point of local minima, where the resultant of all forces is zero, but is not at the destination point.

Pros:

- **It is an algorithm with great accuracy.**
- **It is a time and space efficient search algorithm.**

Cons:

- **It is quite difficult to implement.**

Vector Field Histogram

The Vector Field Histogram (VFH) method involves creating a polar histogram based on information obtained from range sensors, which solves the problem of sensor noise. The histogram plots the probability of obstacle presence against the angle associated with the sonar sensor. This information is gathered by constructing a local map of the robot's surroundings. The histogram helps identify passages large enough for the robot to move through, and the path selection is based on a cost function that takes into account the alignment with the goal and the change in wheel orientation. The goal is to have the minimum cost function.

Pros:

- **It addresses sensor noise by creating a polar histogram that indicates the likelihood of an obstacle in a specific angular direction.**

Cons:

- **It does not consider the dynamics of the robot.**
- **It is also quite difficult to implement.**

A* Algorithm

The A* path planning algorithm is a widely used and efficient algorithm for path planning and obstacle avoidance. It uses a heuristic approach to evaluate the distance between the current node and the goal node, as well as the cost incurred in reaching the current node. The algorithm then selects the node with the lowest total cost as the next node to explore.

In obstacle avoidance scenarios, the A* algorithm incorporates the presence of obstacles by representing them as obstacles or untraversable regions on the map. The algorithm then generates a path that avoids these obstacles and reaches the goal.

Pros:

- **Can efficiently handle complex environments with multiple obstacles, as it evaluates the cost of each possible path to avoid obstacles and reach the goal.**
- **The algorithm can be easily modified to accommodate additional constraints or objectives, such as energy efficiency or time constraints.**

Cons:

- **Has high computational requirements in larger or more complex environments.**

Bubble Band Technique

The Bubble Band method involves creating a bubble around the robot that encompasses the maximum available free space and can be traveled in any direction. The shape and size of the bubble are determined by the robot's geometry and range sensor information. By forming a series of these bubbles, a robot can navigate and avoid collisions. This method uses time-of-flight sensors placed at a fixed angle to assess the presence of obstacles. The concept of the bubble is used to determine if a collision is likely, and by combining this with the profile of the obstacle, the necessary maneuvers to avoid collisions can be determined.

Pros:

- **It demands less computation.**
- **It avoids static obstacles well.**
- **It is a faster algorithm.**

Cons:

- **It cannot guarantee reaching the destination point.**

Recommendation

With regards to the Lunar Rover Mission, I recommend using the Bubble Band path planning technique or something similar because our goal is to avoid obstacles to the best of our abilities and not necessarily to reach a specific destination autonomously. For this goal, Bubble Band is good because it demands less computation and avoids static obstacles well since it looks for the obstacle-free space in any direction and guides the Rover to the closest obstacle-free space before repeating this process recursively. For the Lunar Rover Mission, a strategy could be implemented such that the obstacle detection algorithm would notify the operator of obstacles as well as obstacle-free spaces/bubbles (colored gray as an output of obstacle detection algorithm) in the Rover’s field-of-view. With this information, the operator could decide which obstacle-free space the Rover should drive to.