ARES Mars Rover

SOFTWARE AND AUTONOMOUS SYSTEMS DESIGN ANALYSIS 2019

Akhil Vaid (2017UCO1521) | COE-1 | May 16

Introduction

Hi, my name is Akhil Vaid, Computer Science and Engineering Sophomore at NSUT, New Delhi. My main interests and expertise lie in the field of **Computer Vision** and **Image analysis**.

I have worked with OpenCV since last year and have experience and knowledge of the various techniques used for extracting information from images. If given the chance, I would like to work on the **Obstacle Detection and Motion planning thread** primarily, although I am not closed out on the other threads too.

Analysis of Obstacle Detection Thread

1. Stereo Cameras

There are some options that one can have in terms of getting vertical disparity accuracy. Some of the methods in term of accuracy are:

LiDAR > Infrared > Cameras

If we use cameras then we have to carry out the process of Stereo Reconstruction.

Stereo Cameras involve taking two images shifted in one plane such that we mimic the biological working of our eyes. Using these 2 images we can create a Vertical Disparity map for positions of obstacles in front of the rover.

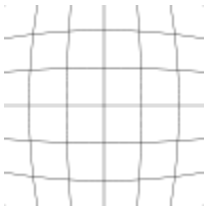
But before we are able to process our captured images, Distortion should be removed

2. Distortion and Calibration

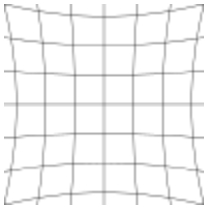
Distortion is of following types:

1. **Barrel**

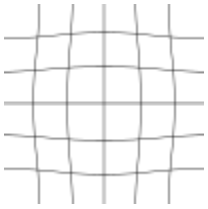
In barrel distortion, image magnification decreases with distance from the optical axis. The apparent effect is that of an image which has been mapped around a sphere or barrel. Fisheye Lenses use this distortion to map Panorama pictures into finite plane.



2. **Pincushion**
   In pincushion distortion, image magnification increases with the distance from the optical axis. The visible effect is that lines that do not go through the center of the image are bowed inwards, towards the center of the image, like a pincushion.



3. **Mustache**
   A mixture of both types, sometimes referred to as *mustache distortion* (*moustache distortion*) or *complex distortion*, is less common but not rare. It starts out as barrel distortion close to the image center and gradually turns into pincushion distortion towards the image periphery, making horizontal lines in the top half of the frame look like a handlebar mustache.



Distortion can be classified as radial or tangential.

Radial distortion can be represented as follows:

$$X_{distorted} = x(1+k_1r^2+k_2r^4+k_3r^6)$$

$$y_{distorted} = y(1+k_1r^2+k_2r^4+k_3r^6)$$

Tangential Distortion is represented as:

$x_{distorted} = x + [2p_1xy + p_2(r_2 + 2x^2)]$

$y_{distorted} = y + [p_1(r_2 + 2y^2) + 2p_2xy]$

To eliminate distortion, we need Distortion coefficients matrix:

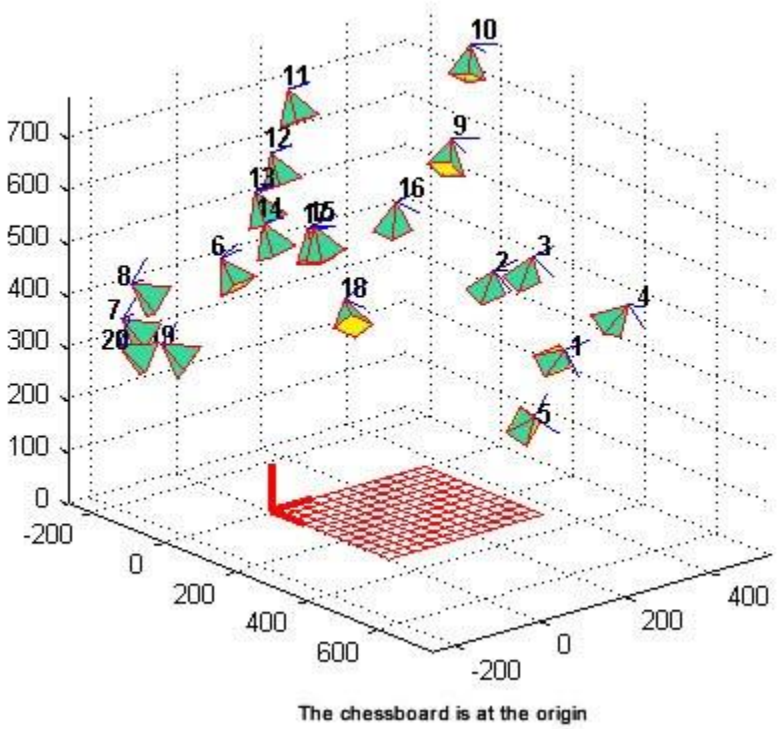
$$Distortion\ coefficients = (k_1\ k_2\ p_1\ p_2\ k_3)$$

OpenCV provides certain functions for Camera calibration.

Some of these functions are:

1. **cv2.findChessboardCorners()**
2. **cv2.calibrateCamera()**
3. **cv2.getOptimalNewCameraMatrix()**
4. **cv2.undistort()**

Usually a chessboard pattern is used for Camera Calibration.



After analyzing all the pictures, we then run the cv2.calibrateCamera algorithm. This is the algorithm that outputs the camera parameters. This algorithm returns the camera

matrix (K) distortion coefficients (dist) and the rotation and translation vectors (rvecs and tvecs).

In summary steps are:

1. Determine the distortion matrix
2. Determine the camera matrix
3. Take input from Camera, Video and Image file list
4. Read configuration from XML/YAML file
5. Save the results into XML/YAML file
6. Calculate re-projection error

Further explanation and code can be found from the official page of OpenCV-python linked below. Calibration is an important step and contains a vast and meticulous approach to achieving it**. The ARES report doesn't mention calibration and so I wanted to suggest it.** It helps achieve accuracy and is a required step before image processing.

https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/py_calibration.html

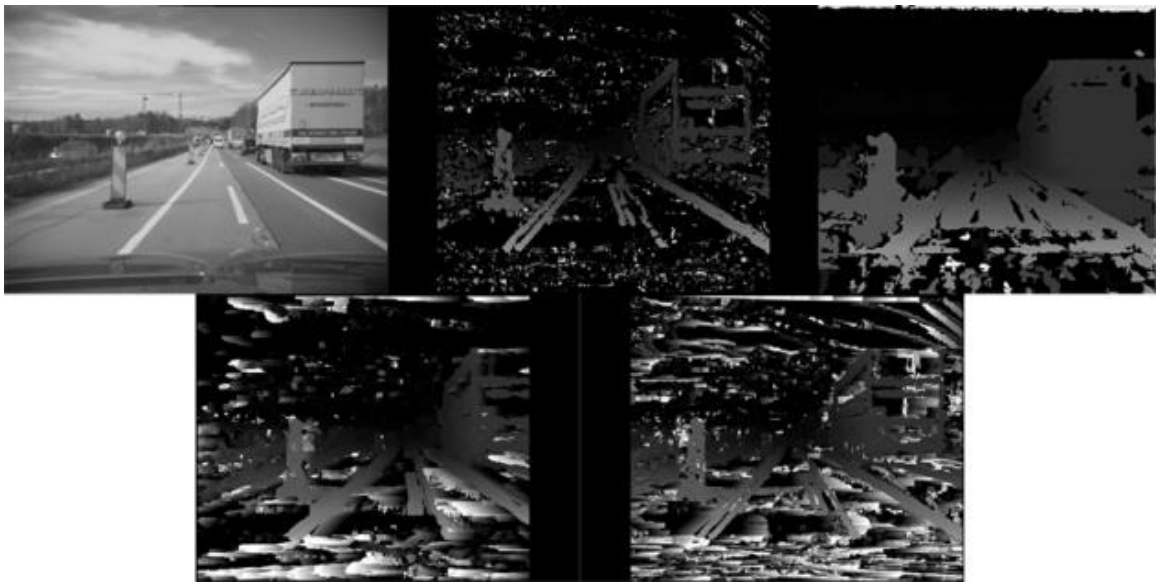
3. SGBM vs BM

SGBM = Semi Global Block Matching
BM = Block Matching

Block matching focuses on **high texture images** (think a picture of a tree) and semi-global block matching will focus on sub pixel level matching and pictures with more **smooth textures** (think a picture of a hallway).

The various Stereo Vision Algorithms that can be used are

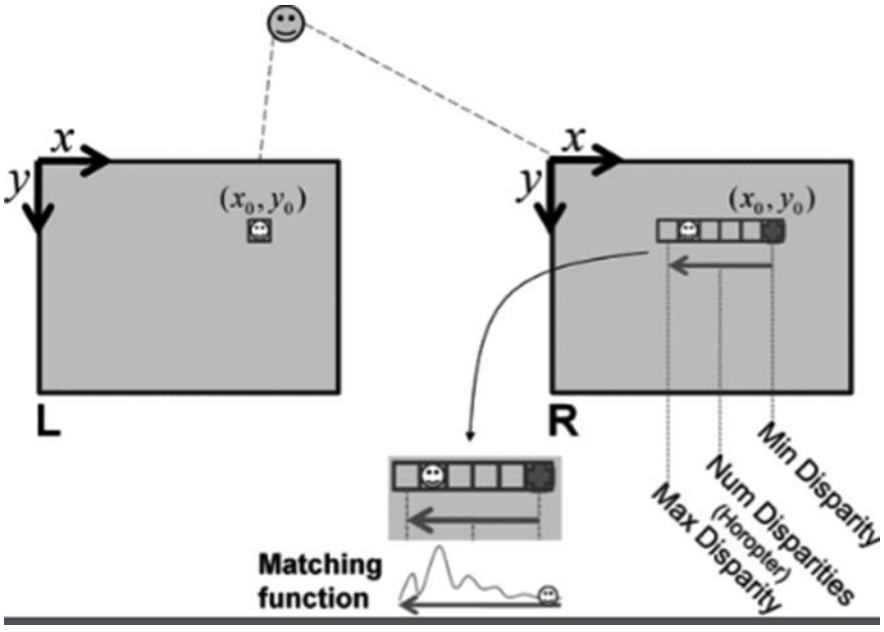
1. SGBM
2. BM
3. Sum of Absolute Differences (SAD)
4. Normalized Cross Correlation (NCC)



The figure above looking from left to right shows comparison of all algorithms (BM, SGBM, SAD, NCC) disparity map with the original image.

OpenCV algorithms deal with Occlusion (when the pixels in one image aren't found in the other image). Occluded areas appear black.

SGBM appears to produce cleanest map out of all and BM seems to have hard time dealing with road. The correct way for the road to be shown is smoothing from grey to black.



Noise Level in images can be reduced by increasing window size and then we see that BM is better albeit doesn't take care of road which can be added later.

Comparison between SGBM and BM

Semi-Global Block Matching	**Block Matching**
Used for **Smooth** textures	Used for **High** texture images e.g. The outside environment
Provides clean map with less noise	Can provide **clean** map under conditions of **big window** size
Takes a toll on resources. **Average** time and FPS are higher than that of BM	It's the **fastest** of all Stereovision algorithms implemented by OpenCV.
If we want to use this then we can **skip frames** in real-time system to counteract the accuracy required.	This algorithm can work on all frames received in a real-time system but has **less accuracy**.

Finally, the question still remains whether to use SGBM or BM for disparity map.

Personally, I feel that BM should work better in case of a high texture environment that should be expected, but since the ARES Mars Rover will most likely be tested in Hallways around smooth textures and Obstacles, SGBM seems to be right. Also, SGBM is highly optimized implementation provided by OpenCV.

I would really like to work with the team at ARES to check the implications of either algorithm being used in real-time.

Calculating Vertical Disparity Map

Quoting from the Paper: "FPGA implementation of the V-disparity based obstacles detection approach"

The V-disparity algorithm supposes that a disparity map *I* has been computed from a stereo image pair.
Once *I* has been computed, the" V-disparity" image *v I* is built by accumulating the pixels of same disparity in *I* along the *v* & axis.

The V-disparity image has the same number of rows than the disparity map but its columns number is equal to the maxima disparity.

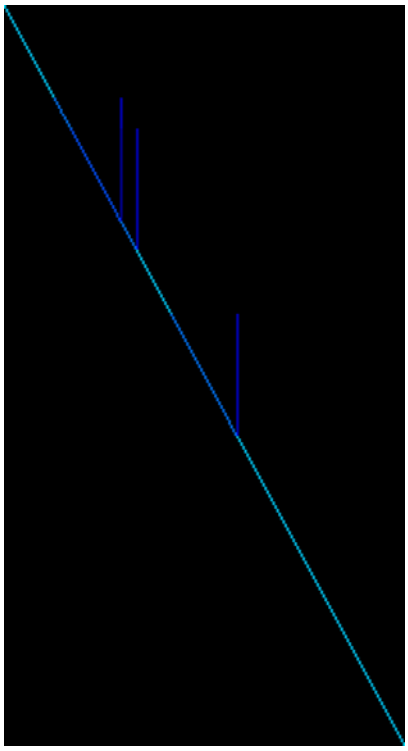


Figure shows: V-disparity Map

For obstacles detection, only the V-disparity image is needed.

But the additional construction of a second image, called the U-disparity image, allows knowing obstacle dimensions after its detection.

The U-disparity image can be built in same way the V-disparity is built. In fact, the number of columns of the U-disparity image is equal to this of the disparity map while its rows number is equal to the maxima disparity. In the U-disparity image, the road profile is not visible while obstacles are represented by horizontal lines.

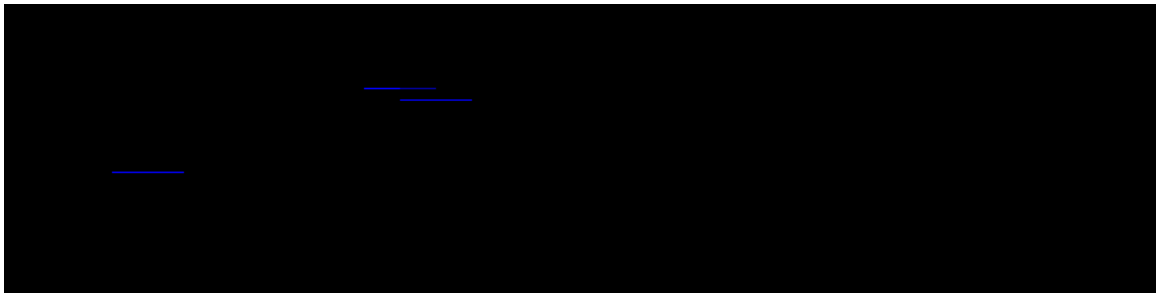


Figure shows: U- disparity map

Combining both U-V disparity Map we can get a Fake Disparity Map that models the obstacles in real-time

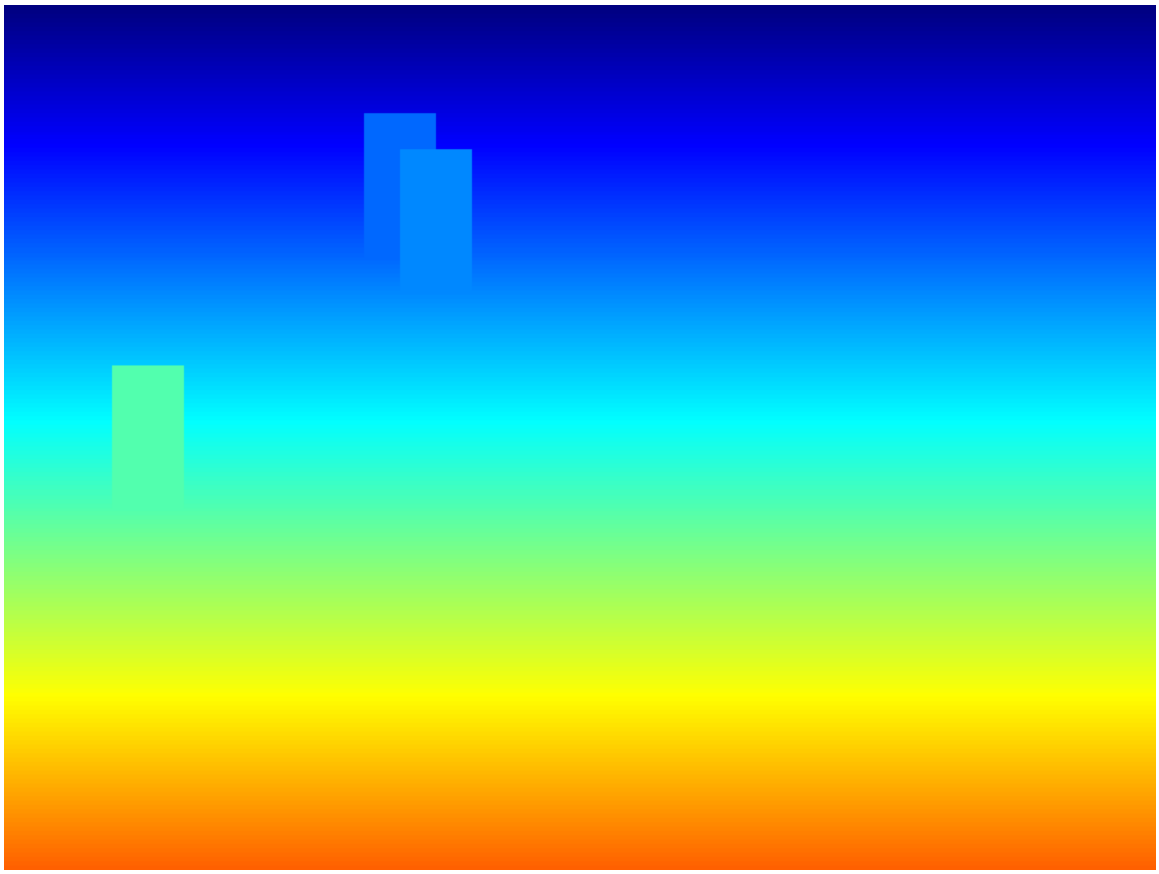


Figure shows: Fake Disparity Map combining U-V maps

MY PROPOSITION

My proposition for **ARES Rover** Obstacle Detection thread is to include a **U-disparity Map** that will help us calculate dimensions for the obstacles in front instead of just how far they are. This can provide useful information for motor controls to navigate around obstacles.

The paper "**Fast Obstacle Detection Using U-Disparity Maps with Stereo Vision**" **by Florin Oniga, Ervin Sarkozi, Sergiu Nedevschi** provides an algorithm by which conventional V-disparity map computational complexity can be bested.

To quote:

*"The solution presented next can identify the obstacles on the road surface, with very low computational complexity. It is **independent** of the road characteristics such as: planar or non-planar, curved or straight shaped surface, making it a feasible solution for general road scenarios."*

HOUGH TRANSFORM

There are primarily two forms of Hough Transform that are used for edge detection.

These are:

1. Randomized Hough Transform (RHT)
2. Probabilistic Hough Transform (PHT)

The **RHT** is better suited for the analysis of **High quality low noise edge images**, while for the analysis of **noisy low quality images the PHT** should be selected.

In our case, we will be receiving High quality Low Noise images from the SBGM matching with Vertical disparity and so we can use the **Randomized Hough Transform** that is implemented in OpenCV as:

```
img = cv2.imread('dave.jpg')

gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

edges = cv2.Canny(gray,50,150,apertureSize = 3)

lines = cv2.HoughLines(edges,1,np.pi/180,200)
```

Where we use the Canny edge detection Algorithm as it is one of the fastest, accurate detection method.

Lines returns array of (ρ, θ) values in Python needed to correspond to Hough Transform.

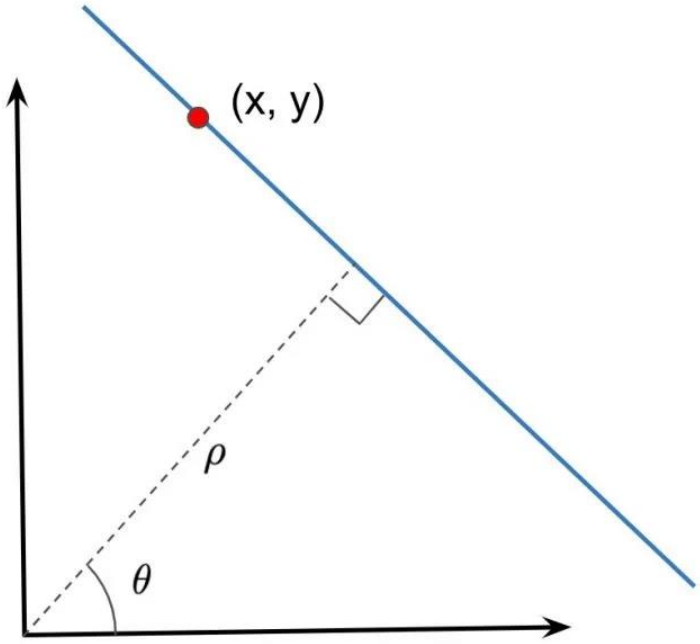


Figure depicts Hough Transform Parameters used in RHT

Hough Transform is used to map the ground plane and Navigable region is detected from V-disparity Map.

My proposition is to use the Randomised Hough transform if not being used before.

Projection to 3D Point Cloud

The next step after getting our disparity maps completed is to get the **transformation Matrix that will be required to project the depth into 3D space.**

Open CV's documentation has an example of a **transformation matrix**.

This matrix is then stored in form of CSV file which **OpenGL** uses to compute the **3D point cloud**.

To compute the 3D matrix all we need is a matrix containing the 3D coordinates and a matrix containing color values for point.

This part is **more implementation based** and doesn't require much deliberation upon.

The only important fact is that since our System will have **incoming frames every second**, we must calculate the values every time for a real-time motion planning to occur. We should send the frames to our GPU for computation.

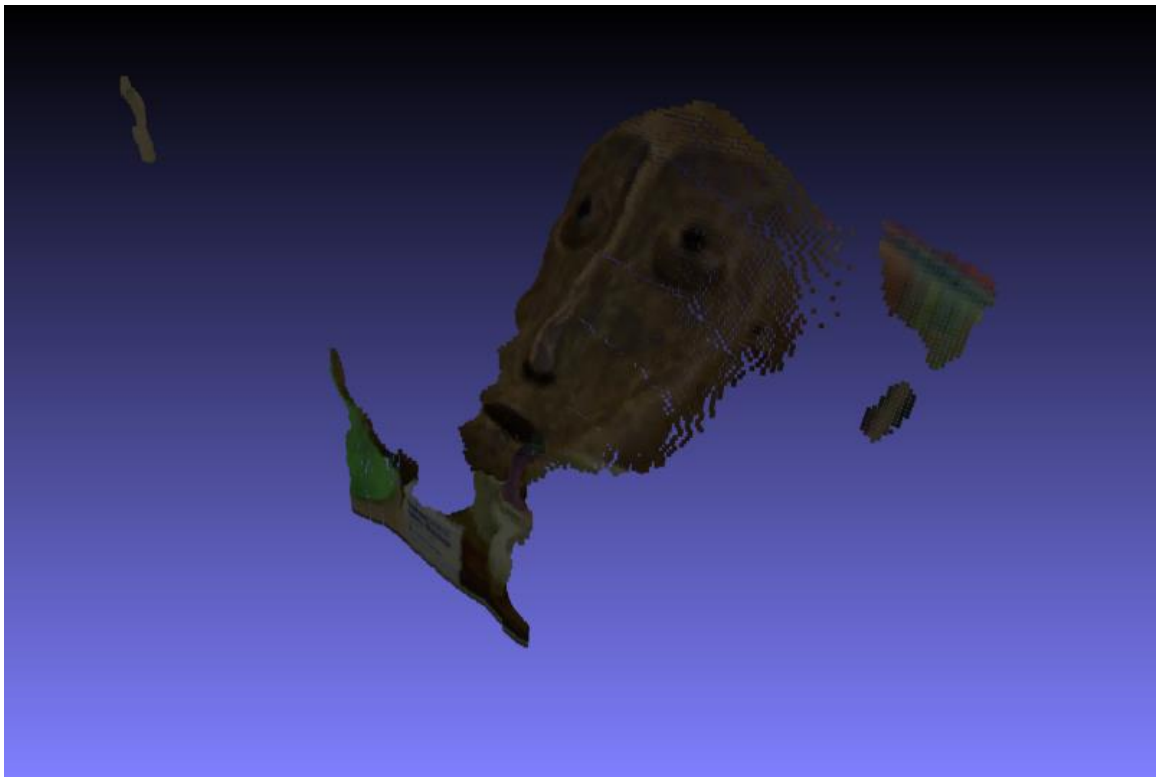


Figure shows a nice disparity map being projected onto a 3D point cloud. Result is clean and noiseless.

MOTION PLANNING THREAD

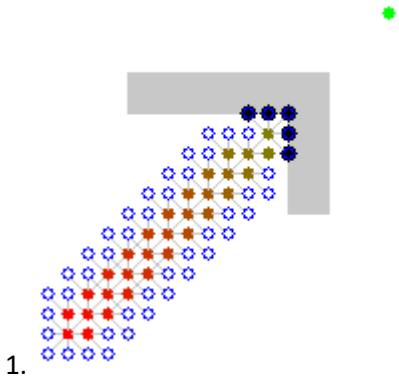
The motion planning Thread involves getting the global state of rover from the Localisation Thread. What this means is that we get the following information:

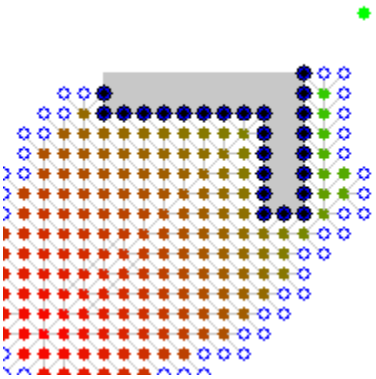
1. Rover Speed:
    - The speed of rover in X, Y, Z directions
2. Rover Direction:
    - The current direction that the cameras are facing
3. Rover Coordinates:
    - The global coordinates of our rover

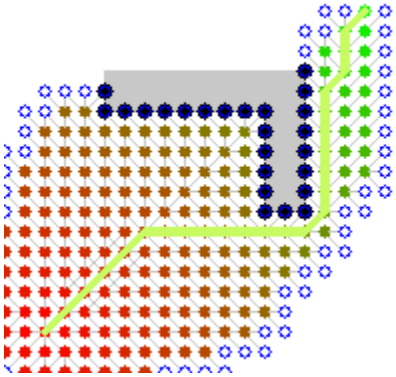
From our Obstacle Detection Thread, we will get the Obstacle Points.

Combining all this information and with the help of the OMPL (Open Motion Planning Library) we use the A* Algorithm for pathfinding.

The A* algorithm is an algorithm which is widely used for graph traversal and pathfinding. It is popular due to its performance and accuracy.

1. 

2. 



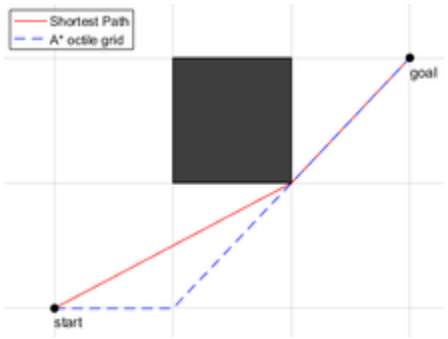
Figures show the Implementation of A algorithm to find route from Start to finish Point (Green)*

MY PROPOSITION

The A* algorithm is based upon the fact that we can move in either the X, Y axis directions at a **single step** i.e. a turn of 45° or 90 ° is used whereas in-fact, our rover can move at any angle or **a combination of a X-vector and Y-vector**.

A variant of the A* algorithm that uses said knowledge of moving at any angle is the

Any-angle path planning algorithms



Any-angle path planning algorithms are necessary in order to quickly find an **optimal** path. For a world represented by a grid of blocked and unblocked cells, the brute-force way to find an any-angle path is to search the **corresponding visibility graph.**

This is problematic since the number of edges in a graph with V vertices is $O(V^2)$. Searching the discrete grid graph can be done quickly since the number of edges grows linearly with the number of vertices, **but the paths are not optimal since the angle of the turns are constrained to 45° or 90°, which will add turns and increase the overall length of the path.**

Smoothing a grid-constrained path after does not fix this problem since the algorithm that found that path did not look at all possible paths.

*Any-angle path planning algorithms find **shorter paths than the grid-constrained algorithms** while taking roughly **same amount of time** to compute.*

So far, four main any-angle path planning algorithms that are based on the heuristic search algorithm A* have been developed, all of which propagate information along grid edges:

1. Block A*
2. ANYA
3. Theta *
4. Field D* (FD*) and 3D Field D*

Further discussion can be done as **a team** on the various implications of choosing the following algorithms for the motion planning thread.

OTHER THREADS

Other than the Obstacle Detection and Motion Planning thread, Other threads and modules come together to work and successfully operate the rover. They are:

Sensor Fusion for Drift Correction Thread

This Thread is basically working to **monitor the movement and motion of the rover**. The (DOF-IMU which is the 9 Degrees of Freedom sensor is used in the rover. Information collected by the following devices is passed through 2 Kalman Filters for extraction of information for the Localization Thread. IMU stands for Inertial Measurement Unit.

Kalman Filter: " **Kalman filtering**, also known as **linear quadratic estimation** (**LQE**), is an algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe." (source: Wikipedia)

Kalman Filter 1:

This calculates the drift correction in the rotation parameters for our rover. It's inputs are:

1. Accelerometer
2. Gyroscope
3. Magnetometer

Its outputs are:

1. Pitch correction
2. Yaw correction
3. Roll correction

These outputs are fed into the local_map_optimization module of the Localisation Thread.

Kalman Filter 2:

This filter calculates the bias correction in the IMU & compensation for a Low GPS update rate. Its inputs are:

1. Accelerometer
2. Gyroscope
3. Magnetometer
4. Latitude
5. Longitude

Latitude and Longitude are obtained from the GPS Module (acquires Global Rover coordinates)

Its outputs are:

1. Velocity
2. X_{global}, Z_{global}

Which are fed into the global_map_representation module of Localization Thread.

Localization Thread

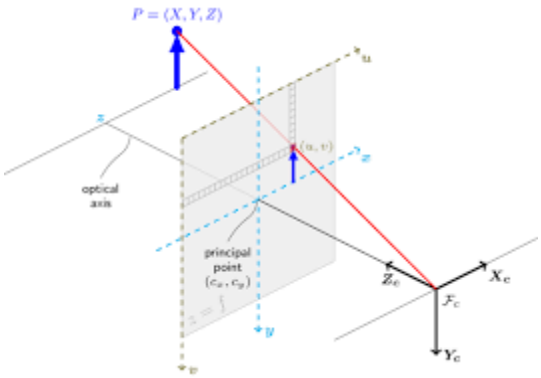
In the Localization Thread we have 3 modules:

1. **Stereo_visual_odometry**

   This module takes inputs from Camera Setup, uses OpenCV functions:

   1. detect:features()
   2. triangulate:points()
   3. SolvePnpRANSAC()

Triangulate points works to reconstruct a bunch of points using triangulation. Its output is a vector of 3D projection points.



solvePnPRansac is very similar to **solvePnP** except that it uses Random Sample Consensus (RANSAC) for estimation.

"Using **RANSAC** is useful when we suspect that a few data points are extremely noisy. For example, consider the problem of fitting a line to 2D points. This problem can be solved using linear least squares where the distance of all points from the fitted line is minimized. Now consider one bad data point that is wildly off. This one data point can dominate the least squares solution and our estimate of the line would be very wrong. **In RANSAC, the parameters are estimated by randomly selecting the minimum number of points required**. In a line fitting problem, we randomly select two points from all data and find the line passing through them. Other data points that are close enough to the line are called inliers. Several estimates of the line are obtained by randomly selecting two points, and the line with the maximum number of inliers is chosen as the correct estimate."

2. Local_map_optimization

Outputs from visual_odometry module are passed onto Local_map_optimization module. Actually, this module will convert the local coordinates to global coordinates and find the heading direction of our Rover.

General Graph Optimization (g2o) namespace is used here and pose estimation of rover is done. g2o is **an open-source** C++ framework for optimizing graph-based nonlinear error functions. g2o has been designed to be easily extensible to a wide range of problems and a new problem typically can be specified in a few lines of code.

Also a 3D point cloud is generated.

The functions used are:

1. Sparse Optimization
2. SE3 Quat

   This provides **optimized** local and global coordinates which are given to **OpenGL** for plotting a 3D point cloud.

Finally, the global map is presented to the global_state () module which carries the rover state and other dynamic constraints if any. This module is called by the Obstacle detection and Motion planning Thread discussed before.

ROS NODE FOR MOTION CONTROL

After the A* algorithm is implemented and a shortest path is calculated, the commands for movement are generated and passed onto this thread.

The drive motors as ros node for getting control signals from the controller.

Motion is carried out and rover moves ahead.

BALL DETECTION

Though it isn't mentioned explicitly in the ARES report, it is implied that the rover is expected to pick up a green coloured ball.

The Frames received by the Camera can be processed using OpenCV **contour detection** for Circle.

<u>Implementation of Contour detection in OpenCV</u>

```
void cv::findContours ( InputOutputArray      image,
                        OutputArrayOfArrays   contours,
                        OutputArray           hierarchy,
                        int                   mode,
                        int                   method,
```

Point *offset* = Point()

)

Finds contours in a binary image

OpenCV also provides Colour detection function which will be used to verify if the ball is the right colour or it is something else.

For colour detection we basically use a mask of our desired colour and we take the **BITWISE AND** on the input image and our mask. So that only the green coloured region is highlighted and stored.

Conclusion

In this report I have summarized the main points on which I would like to work and improve on in the **Obstacle Detection and Motion Planning Thread**. There is always room for improvement and so, the proposed changes can be implemented in other optimized manners.

I have also discussed what I understood from the other threads and how they all come together to work as one. The various outputs from one Thread to the inputs to another Thread are explained.

I would really appreciate the opportunity to work with the **ARES team** and my seniors at NSUT as I am interested in carrying out research and the **ARES Mars Rover** is a **practical** and **exciting** project at the forefront of the field **of Computer Vision and Applications**.