Project Title: **Generation of a Combined Bird's Eye View of a Region around a Robotic Rover From Omnidirectional Stereo Images**

1 Project Problem Description

Research Question: The goal of this project was to answer the question: are we able to get meaningful bird's eye view images from 360°omnidirectional/monocular stereo images [1] taken on a small robotic rover travelling along a path through simulated Mars/Moon terrain?

Project Updates: In the original proposal in addition to the above task, I intended to stitch together sequential orthophoto images of regions around the rover at a particular time-point to build an aerial map of the areas the rover traveled along it's path. This was not implemented ultimately due to the runtime of the code (which was most slowed down by the python griddata interpolate function in the scipy library). Since it took so long to generate a single orthophoto of a region around the rover it would have taken a long time to debug this combined orthophoto map generation feature. I simply didn't have enough time to get to this part of the project completed under the time constraints.

The primary functionality of generating a flat orthographic map of a region around the rover at a particular time point and correcting that for variable sun exposure to improve the readability of that image was achieved and implemented. In addition, the code generates a pixel correspondence between the location of the pixels in that orthophoto image to the location of the pixel with respect to the rover body frame (in meters), which could easily be transformed to the global site frame. With that information it would be a simple extension and brute force job to generate the sequential orthophotos and stitch them together to build an aerial map of the test site.

Background: For a rover, obstacle prediction and avoidance, as well as path planning all depend on robust estimates of the 3D positions and dimensions of other entities in the surrounding environment. One way around this issue is to use rich LiDAR point clouds for accurate measurements of obstacle locations, however due to the high cost of such units, poor measurements at long distances, and the need for sensor redundancy, 3D object detection using monocular images for 3D object detection remains a valuable topic to investigate. An orthographic image/"orthophoto", also known as a bird's eye view (BEV) enables us to escape the image domain by mapping image-based features into an orthographic 3D space. In this orthographic 3D space, where *scale* is consistent and *distances between objects* are meaningful, we can reason holistically about the spatial configuration of the scene for such obstacle prediction and avoidance. This project is meaningful and interesting because post-processing stereo omnidirectional images to orthographic/BEV (used interchangeably) images can greatly improve the understandability of the data to a human user, produce a true-to-scale map which can be useful for reasoning about the configuration of obstacles in the environment, and give better visualization of the path conditions the rover traversed.

2 Project Methodology

2.1 High Level Overview

The overarching goal is to generate a BEV image surrounding the rover comprised of BEV transformed stereo images. This will involve obtaining the individual BEV images from the omnidirectional stereo images (Section 2.2). Then those will need to be stitched together to form a single surround-rover orthophoto (Section 2.3). Additionally, corrections were made for the varying exposure levels (arising from the sun position) among the omni-directional camera images taken at the same time step (Section 2.4). The code so far serves as a strong basis to work off of in order to fufill the last part of the proposal, i.e. stitching together sequential surround-rover orthophotos to generate an aerial map, Section 2.5 explains the steps to be taken with the current code base to achieve this objective. See Table 1 for a summary of all the functions used in this project, refer to the submitted code for more details.

Running the code inside Docker will output the following into the output folder for a given fram:

- 5 original omni-camera images name "img[n].png" n = 0, 1, 2, 3, 4
- the corresponding orthophotos of those 5 images "rgb[n].png" n = 0, 1, 2, 3, 4
- the stitched surround-rover orthophoto called "ALLPXrgb.png"
- a text output of the position and orientation of the orthophoto in "orthophoto_pose.txt"

Note that due to limitations in Docker and time it took to get it set up and running properly, the submission as it is outputs all of these quantities for one time point only (specifically for "fileid = "2018_09_04_18_18_15_891197"", "frame = "000978""). The same code was just run on other time-points to generate the orthophotos at different time-points in 7 (how to specify what time point to generate an orthophoto is described below).

2.2 Generating Individual Orthophotos

The orthographic photo/BEV transformation is a way to generate a top view perspective of an image. To obtain the BEV image I use the depth information of the terrain shown as the surface in Figure 1. This terrain depth information was obtained by simply reading in the pre-computed point cloud data provided in the CPET dataset[1] (this could have also been found by the using stereo camera pairs to compute the dense disparity maps (as per Assignment 4) which can be used to compute the depth of each pixel and thus the 3D position of that pixel point). As depicted in Figure 1, at every pixel of the orthophoto, the height/elevation was found through interpolation (nearest neighbours, bilinear/bicubic, linear, etc.) of the digital surface model, i.e. the point cloud data (for this project), where the height/elevation information is known only at the specified points in the point cloud. Each of these interpolated points (which compose the orthophoto grid) on the 3D-surface of the terrain are then transformed to the camera frame (using knowledge about the camera pose and the fact that the point cloud is in the reference frame of the respective omni-camera) in order to assign the RBG intensity value of a given pixel in the orthophoto as the corresponding pixel in original RBG stereo image. The quality of the interpolated height estimates are crucial and the biggest source of error, because the rover camera view angle is from the side, not from above. With this side view angle, differences in the height estimate translate directly to the incorrect mapping to the RGB stereo image, i.e. an incorrect pixel assignment.

It is clear that this process is heavily dependent on the quality of interpolated results. Unfortunately, though it would have been interesting to do so, there was not enough time to test different interpolation methods and compare the resulting orthophotos.

The following gives more details on how this translates to the code implementation. In "main()" the code starts by loading in the omni-camera intrinstics and extrinsics as well as the transforms of the omni-cam to the omni-sensor (i.e. "allOmni", "allT_OrefP[n]"). Lines 794 and 795 are lines that should be changed to match the frame and timestamp of the point along the path you want to obtain the orthophoto for, the information for the timepoints used in this report are in comments on lines 779-788 and summarized in Figure 6. The code is set up for submission is such that it generates a single surround orthophoto and outputs all the intermediate images as described in the Section 2.1 (default time-point set to be fileid = "2018_09_04_18_18_15_891197" and frame = "000978").

An orthophoto of that single omni-camera image and the corresponding transformed grid in the omni-sensor frame is generated for every omni-camera n=0,2,4,5,8 in lines 804-840. Note only these "n" cameras are used because the point clouds in the dataset are measured in the frame of those cameras only. The "getOrthophoto" function is called for every n. In there the following occurs. The images are loaded in and corrected for lens distortions and the camera's intrinsics with OpenCV's undistort function and for sun effects as per Section 2.4. The 3D point cloud (has the 3D location of pixels in the frame of the omni-camera in question) is projected onto the image plane (transformed to pixel coordinates) of the omni-camera using "get3d33d", which does this through back-projection, (see formulas below):

$$x = f_x \times \frac{x}{z} + c_x$$
$$y = f_y \times \frac{y}{z} + c_y$$
$$z = \frac{z}{z}$$

Notice that in the image plane of the omni-camera, the z component is eliminated, and the y and x component remain (see the "0" frame of the omni-camera in Figure 1) for a visualization of this). Although not shown in the report this transforms the trapezoid point cloud which has gaps due to rock occlusions (see again Figure 1), into a rectangle of uniform points (which makes sense because that's what the camera's image sensors "see").

All points that aren't in the in the range of the 2D image plane of the omni-directional camera are eliminated using a binary mask. Using bilinear interpolation as per "bilinear_interp_vectorized", each back-projected point in the point cloud is interpolated on the image from the camera for each R, G, B channel. This gives every point in the point cloud a corresponding pixel colour (denoted "[r,g,b]_interp" in the code, this is done on line 520-522). Using the binary mask obtained above, the points in the 3D cloud that aren't in the omni-image in question are zeroed out and removed. An appropriate grid (in meters measured with respect to the omni-camera frame) is generated from the point cloud points that are "in range" and the pixel values that belong in each of those are interpolated from the (points in point cloud, "[r,g,b]_interp") (point, value) pairs obtained above using "griddata" from scipy.interpolate in lines 584-586. The single view orthophoto and the grid specifying the location of each pixel in the omni camera frame are output as a result.

This grid which is in the specified in the omni-camera frame is then transformed to the omni-sensor frame with the "tranformGrid" function in lines 812, 821, 827, 833, and 839 for each of the n cameras respectively. See Figure 3 for an example of the 5 orthophoto generated for a single time-point.

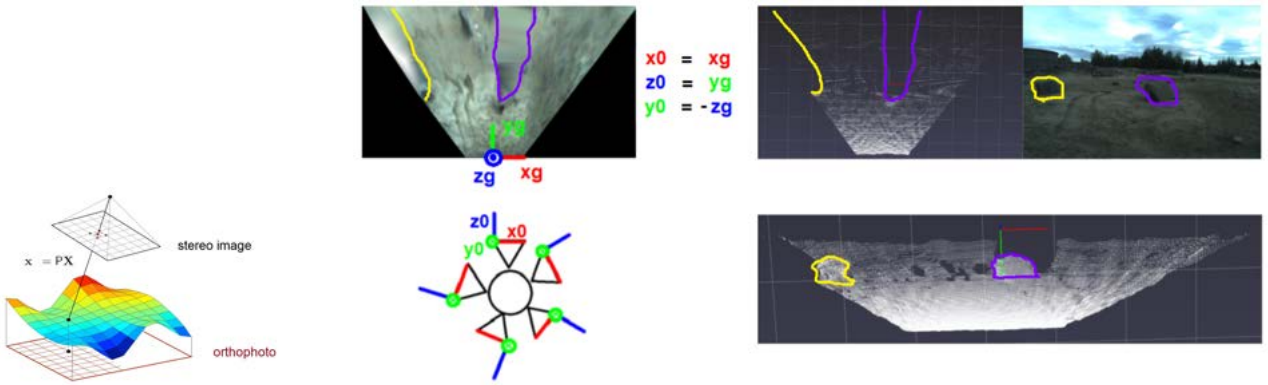

Figure 1: (left) The process to transform an image to a birds eye view [2]. (right) A depiction of the relation of the point cloud to the orthophoto/how it is used to generate the orthophoto. This figure outlines explicitly how occlusions in the original omni-camera image (topmost right corner) are reflected in the 3D point cloud and orthophoto. Occlusions are a rock on the right and left outlined in purple and yellow respectively.

2.3 Stitching Individual Orthophotos Into One

There are many advanced methods to stitch together the individual orthophotos obtained above, ex. matching features (ex. Harris Corner Detector), finding the appropriate transformation to align the features, applying those transformations to the individual stereo images, then stitching the stereo images together. An advanced method could have been used to remove undesirable "seams" in the stitched "surround"-rover images would be OpenCV graphs cuts.

A simpler approach outlined in Figure 2 was implemented and are contained within the function "surroundOrthophoto". It uses a very similar grid interpolation technique as described above. Using the all 5 orthophotos obtained and their corresponding transformed grids (containing the location [meters] of each pixel in the orthophoto with respect to the common frame of the omni-sensor). Using a similar binary mask technique, all 0 pixel values (i.e. black) are removed because they will interfere with the interpolation see the coloured areas in the orthophotos in Figure 2 A). All the pixels and their corresponding grids from the 5 images are concatenated into a master list of all pixels values and a master list of their corresponding grid coordinate pairs specifying their location in the omni-sensor frame (see B) in the same Figure). A master grid is generated to encompass all the pixels in all the 5 orthophotos similar to how this was done above (see C) in the same Figure). Using "griddata" this grid was interpolated on the concatenated pixel coordinate pair data prepared above. As a result the stitched orthophoto was created (see D) in the same Figure).

Note that after removing the 0 pixels, there is not a lot of overlap with the pixels (see B) in the same Figure). In addition, because of sun correction, the seam is minimal and the transitions in the combined orthophoto are smooth.

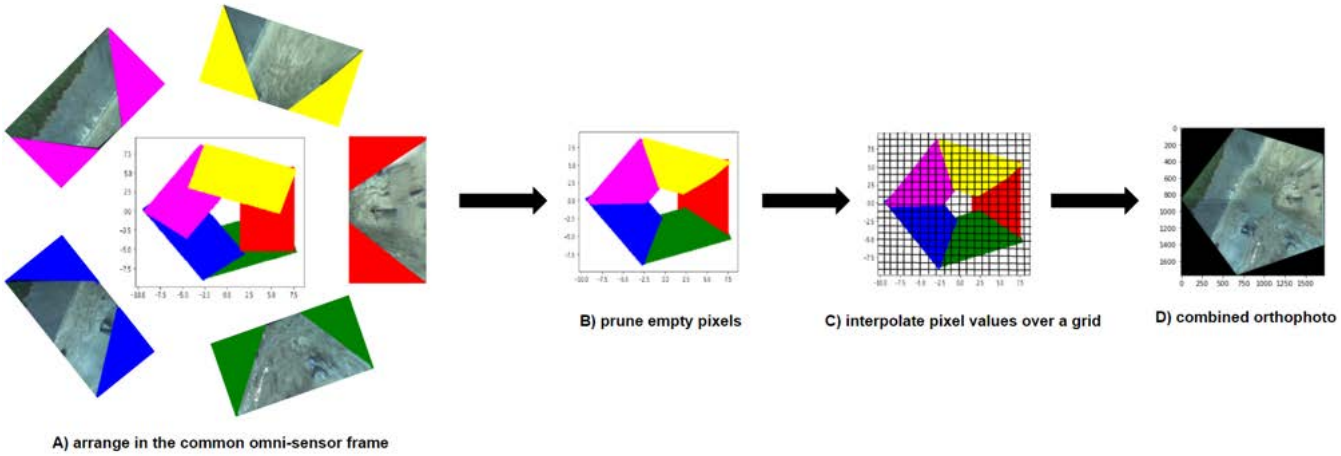

Figure 2: This outlines the how the individual orthophotos are stitched together to form the combined orthophoto.

2.4 Sun Correction

It is important to compensate for the effect the sun's position relative to the rover because the differences in darkness that result from it negatively impact the resultant orthophoto's image quality. The sun's position with respect to the rover impacts the auto exposure of each camera in the omni-directional setup independently and differently. As a result, there are inconsistencies in darkness of the individual orthophotos obtained from these images, which result in hard to understand stitched orthophotos as seen in Figure b) and d) in 4. In the proposal I proposed solving this problem by correcting for these exposure variations using gamma correction [5]. Since the CEPT dataset provides all the individual stereo images as well as the sun pose vector with respect to the rover body frame the idea was to create a function that calculates the quantity of incident light from the sun on the camera. Based on the quantity of incident (direct) sunlight on a given stereo camera I would be able to scale the corresponding gamma adjustments proportionately.

I actually ended up finding a much simpler solution to achieve the same sun exposure correction result, without using the extra sun pose information. Before transforming to the orthophoto view, the original omni-directional camera images are corrected for the effect of the sun position by performing a gamma correction on the lower half of each rover image (precisely all pixels with y position greater than 200) as shown in Figure 4 a). The red box encapsulates all the representative pixels to be considered when finding an optimal gamma. This gamma is found using the power-law relationship i.e. $\gamma = \dfrac{\log V_{out}}{\log V_{in}}$, with the $\log V_{in}$ being the mean of the representative pixels and the $\log V_{out}$ being $0.5 * 255$ (i.e. the middle of the dynamic range for for 8-bit integer images). A gamma correction is then performed with this computed gamma on the omni-image in question (see b) in Figure 4. This evens out the dynamic range and centers the image to be the mean of the "dirt" (lower-half) pixels, making the ground lighter. This is because the sky (being much brighter than dirt and even more brighter where the sun was) took up a lot of the dynamic range of the camera which the auto-exposure tried to capture. Because the ground (lower-half) pixels are generally the same across different omni-camera images, after correcting each image individually, transforming to the orthographic view (as seen in the last row of Figure 3), then snitching them together (see d) in Figure 4), we see that we get a more pleasant and even image (comparing b) and d) in Figure 4).

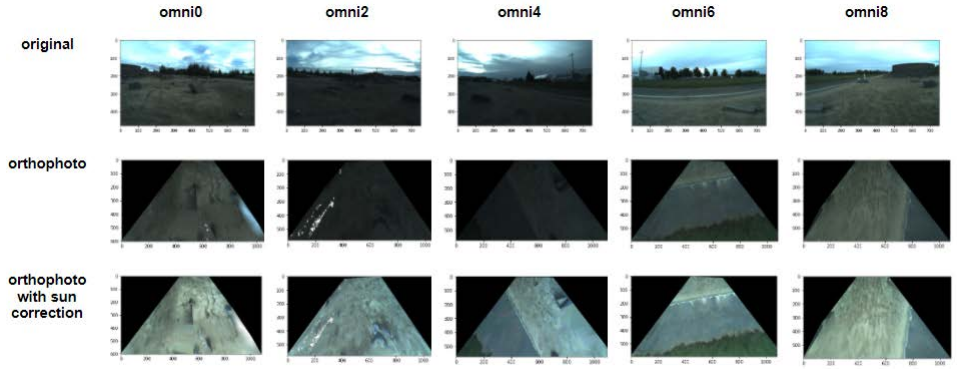

Figure 3: Shows original 5 different omni-directional camera views, the corresponding transformed orthophoto, and the sun corrected corresponding orthophoto.

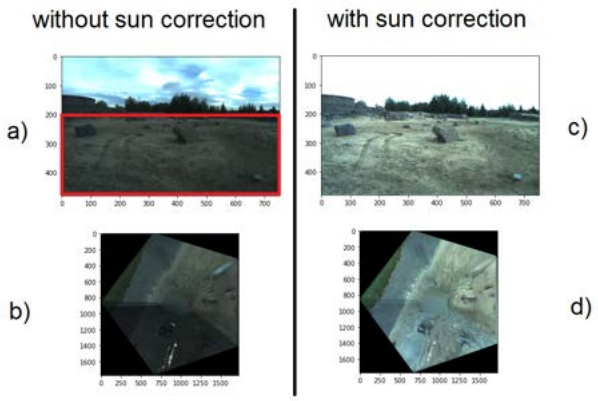

Figure 4: Shows the how sun correction was implemented as well as the positive impact it has on the stitched surround-rover orthophoto results.

```
    def sunCorrection(i):
        # take only the bottom pixels (i.e. the ground pixels that will be relevant for the
        #orthophoto/bird's eye view image)
        img = i[200:, :, :]
        # convert img to grayscale
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        # compute gamma = log(mid*255)/log(mean)
        mid = 0.5
        mean = np.mean(gray)
        gamma = math.log(mid*255)/math.log(mean)
        # do gamma correction
        img_gammacorr = np.power(i, gamma).clip(0,255).astype(np.uint8)
        return img_gammacorr
```

2.5 Creating A Map

Using similar approaches to those described above it is a small extension (that will require a lot of debugging and time) to generate a map. It would have been really interesting to see how a generated full map would look, but unfortunatlly time was tight. Below describes the approach that would have been taken. This was actually implemented with the current outputs of the implementation manually in Figure 7 which was made for evaluation. This section describes how it would be executed in code.

Using the pose information from the dataset ("global-pose-utm.txt") the exact location of the rover the moment each stereo image were acquired is known (in this implemetation this is found and exported to the orthophoto_pose.txt file). To construct a map the center of each surround-orthophoto image is placed at the location it was taken [4]. This accounts for the translation of the rover in the map. To account for the changing orientation of the rover relative to the aerial map it's moving across, the relative orientation heading derived from the IMU data (accelerometer, gyroscope i.e. pose information) will be used to apply an appropriate rotational transformation to the BEV images.

It would have been interested to see the map constructed in this way. I suspect that due to phenomena such as wheel slip, the results using the GPS information will differ from those generated from a feature matching technique.

3 Project Evaluation and Results

To evaluate the quality of the individual orthophotos/BEV images produced the resulting orthophotos are compared to the aerial map of the terrain (i.e. the georeferenced map of the MET at the CSA in UTM coordinates [1]). Rather than compare the pixel values one by one, qualitative visual inspection of the images was performed. Similarly, the efficacy of the sun-correction is evaluated through qualitative visual inspection. This is all done in Section 3.2. To evaluate the accuracy of the orthophotos through quantitative evaluation, the distances between objects in the orthophoto are computed and checked to see if they match what the distance should be in real life, this is done in Section3.1.

3.1 Quantitative Evaluation: Checking if Distances in Orthophotos are True-To-Life

In this section, the relative distances between objects in the generated orthomap are checked. It is difficult to get relative distances between objects just using the georeferenced map of the MET (due to the coarse axis markings). The idea here is to validate that distances in the orthophoto are true to life (a property of true orthophotos) by looking at the distance between the *wheel tracks in the orthophoto*. This is a good choice because the computed wheel separation from the orthophoto can be compared to a quantity that is actually known accurately (from mechanical drawings of the rover wheelbase).

The magenta and cyan "+" markers shown in Figure 5 visually mark the pixel locations of both rover tracks (respectively $(x_1, y_1) = (1250, 665)$ and $(x_2, y_2) = (1250, 720)$), note that these were found ad-hoc visually. This corresponds to a pixel separation of 55 between the two wheel track features, i.e. the wheels themselves. This orthophoto was interpolated on a pixel grid (generated as per the methods described above) which has a corresponding (x,y) grid of pixel locations of the same size which is in meters measured with respect to the omni-sensor frame. The spacing of the grid for this particular image (as this will vary slighty for each image) is (xspacing = 0.010009985377303465 [m] and yspacing = 0.010008389408076468 [m]). This means that each pixel in the image is equivalent to ≈0.01 meters. Thus, the 55 pixel wheel seperation computed in the orthophoto corresponds to ≈0.55 meters. As per Figure 5 the actual wheel separation taken from the center of the wheels is 5.45 meters. This result is in the right ballpark from coarse placement of the markers. Thus, distances in the orthotphoto are correct and thus validates the orthophoto output of this project.

This validation method could be improved upon by more more careful placement of the markers used for measurement (ex. by placing the marker in the exact center of a feature to improve pixel accuracy). Each pixel corresponds to about a 0.01 meters or 1 cm resolution, but that is not the true resolution of the image features (because there were distortions from the sampling/interpolation methods used to generate the image). To get a more accurate estimate of the image features a higher resolution camera, better interpolation methods to generate the orthophoto, as well as post processing to make the features sharper would all help.

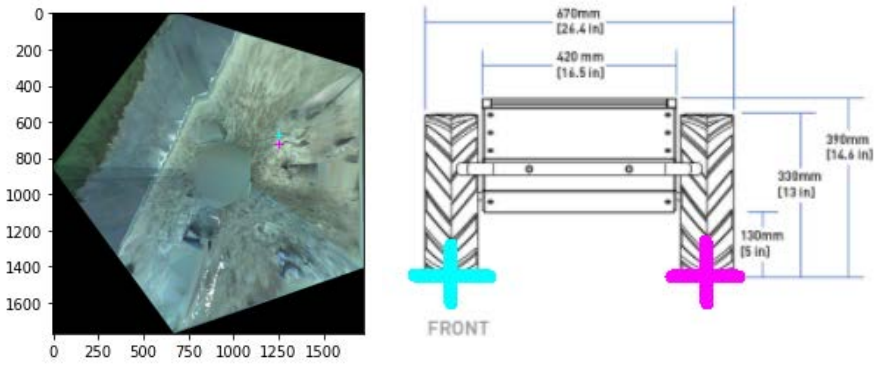

Figure 5: Orthophoto with wheel tracks marked with magenta and cyan markers (left). A technical drawing of the Clearpath Husky Rover outlining wheelbase dimension [3] (right).

3.2 Qualitative Evaluation: Visual Inspection of Orthophotos and Sun Correction

Besides quantitatively comparing predicated vs. actual distances between objects, another feasible method of validating this project is evaluating the quality of the output images by visual inspection. Seeing Figure 3, the effect of sun correction on each individually transformed image is apparent. It is clear that the darker images are brightened and it's easier to discern terrain features. As per Figure 4 b) and c) it is clear that the sun corrected orthophoto (d) is much better than the orthophoto without any sun correction (b). The contrast across the image (b) is much more uniform throughout and the features are easily identified in all parts of the image. Thus, it can be concluded that the sun correction implemented was effective in improving the quality of orthophotos.

An orthophoto, is essentially a bird's eye view of the terrain. The features found in the orthophoto should thus match those in the aerial map of the terrain (i.e. the georeferenced map of the MET at the CSA in UTM coordinates [1]). Eight representative time points with interesting orthographic features (outlined in Figure 6) were selected and the surround-rover orthophotos for those were generated. Using the position and orientation of the rover at these time points, extracted from ""global-pose-utm.txt"" (information listed in Figure 6), the location of the rover (frame) with respect to the site frame is known. This was used to locate (on the aerial map) at what point and in what orientation along the rover's path the orthophoto should be placed/represents. The process of finding the location along the patch, marking it, and reorienting the orthophoto to match the orientation in the aerial map was done manually in an image manipulation software (MS Paint). The result is Figure 7 where the colour of the annotations match the colour assigned to each representative time point in Figure 6. The coloured circles on the aerial map correspond to the center of the respective orthophotos, also marked with the same coloured circle. Features annotated in the appropriate colour in the aerial map correspond to those annotated in the appropriate surround-orthophoto. Note: the scale of the individual surround-rover orthophotos is not equal to that of the aerial map; they are bigger to make it easier to discern the matching features.

As evident in Figure 7, the features match up pretty well. Obstacles such as walls and rocks cause occlusion in the orthophotos (a whole body of research is dedicated to taking this into account and still generating good orthophotos), because there is a lack of pixel information of the terrain that is behind the occlusion. Thus, occlusions (and the fact that the omni-cameras can't physically see the tops of occlusions) cause the obstacles to look a little different than one would intuitively expect in the orthophoto (it's similar but not the same as the information an aerial map would give). However, under further inspection the correspondences between the orthophotos and aerial photo are clear. Distinguishing features such as the outline of the barriers, large rocks, and unique terrain features are clear and annotated in the Figure.

It's interesting to note the effect of a very close occlusion in the orthophotho of the violet in Figure 7 (or 7th) representative time point. A bright purple "x" marks the location of what seems to be a large rock in the aerial map and orthophoto in Figure 7 (it must be large if it's visible in the aerial map). Everything beyond that (outlined in the same bright shade of purple) is occluded and thus shows up as blurry (due to pixel interpolation)

in the orthophoto. How this occurs is more clearly depicted in Figure 1. Again, it's clear that everything behind the purple and yellow rocks in this Figure is unknown and shows up as "holes" in the terrain point cloud. Due to this lack of information, they show up as blurred regions in the resulting orthophoto.

Note: The generated surround-rover orthophotos are centered about the omni-sensor frame. Since the transformation from the omni-sensor to the rover is given, the physical grid coordinates of the orthophoto pixels could have been transformed to the rover frame for accurate positioning (good to note for a future implementation). This level of accuracy was not needed for this type of evaluation, thus the omni-sensor center was approximated to be the rover frame center (a fair assumption considering the scale of comparison with the aerial map).

The manual process of generating Figure 7 by visually visually determining the orientation and position on the map was a very laborious process and not as accurate as codifying the approach using the exact orientation and position value. Given the information the code generates so far there is enough there to easily code this up. In hindsight, given more time it would have been better to read in the aerial map, get the pixel to meters mapping and execute this process through code. It was difficult to start this project because a strong understanding of the data available and the orthographic transformation itself had to be developed first. Only after visualizing a lot of the data (ex. the point cloud data with Python library pptk, displaying at all the omni-images side by side, and drawing diagrams of how all the frames/images related to each other) was progress made. Debugging was easier with this project because it's all very visual and one can simply visualize what going on at each step to find errors.

		Global Pose Estimates from VNIS-Fusion, filtered IMU, orientation and high-resolution terrain elevation model	
Frame	**Time**	**Center Position [easting(m), northing(m), altitude(m)]**	**Orientation represented by Quaternion [qx, qy, qz, qw]**
000000	2018_09_04_18_14_33_618401	[6.25511655e+05 5.04174655e+06 2.42911120e+01]	[-0.02469665 0.02511756 -0.00501619 0.99936681]
000111	2018_09_04_18_14_58_760015	[6.25513909e+05 5.04173894e+06 2.42999476e+01]	[-0.0029515 0.01037439 -0.59627107 0.8027107]
000233	2018_09_04_18_15_26_674852	[6.25516094e+05 5.04172739e+06 2.43192307e+01]	[-0.00396478 -0.00514246 -0.81880061 0.57404128]
000811	2018_09_04_18_17_38_6072	[6.25520787e+05 5.04176239e+06 2.45290972e+01]	[-8.59613256e-05 -5.26090478e-04 4.91855855e-01 8.70676481e-01]
000978	2018_09_04_18_18_15_891197	[6.25513562e+05 5.04177017e+06 2.45847156e+01]	[5.00823826e-04 -1.43678917e-02 9.76045052e-01 2.17092998e-01]
001445	2018_09_04_18_20_01_817314	[6.25499701e+05 5.04174650e+06 2.53140934e+01]	[0.00429468 0.01359513 0.99925285 -0.03592303]
001579	2018_09_04_18_20_32_275748	[6.25493826e+05 5.04175388e+06 2.61396284e+01]	[0.09261088 -0.01397709 0.50268702 0.85937979]
002010	2018_09_04_18_22_10_193947	[6.25494478e+05 5.04173702e+06 2.45143810e+01]	[9.65972581e-03 2.46576170e-04 -4.82313045e-01 8.75945635e-01]

Figure 6: This table includes the details of the 8 representative time points selected for comparison with the aerial map.

References

[1] Olivier Lamarre et al. "The Canadian Planetary Emulation Terrain Energy-Aware Rover Navigation Dataset". In: The International Journal of Robotics Research (2020). DOI: 10.1177/0278364920908922. URL: https://doi.org/10.1177/0278364920908922.

[2] RidgeRun. RidgeRun's Birds Eye View project research. https://developer.ridgerun.com/wiki/index.php?title=Birds_Eye_View/Introduction/Research. 2020.

[3] Clearpath Robotics. HUSKY UNMANNED GROUND VEHICLE. URL: https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot/.

[4] Steven Roebert, Tijin Schmits, and Arnoud Visser. "Creating a bird-eye view map using an omnidirectional camera". In: Proceedings of the 20th Belgian-Netherlands Conference on Artificial Intelligence (BNAIC 2008) (2008). URL: https://www.researchgate.net/profile/Arnoud_Visser/publication/224773001_Creating_a_Bird-Eye_View_Map_using_an_Omnidirectional_Camera/links/004635177b0ce9d6df000000.pdf.

[5] Adrian Rosebrock. OpenCV Gamma Correction. https://www.pyimagesearch.com/2015/10/05/opencv-gamma-correction/. 2015.

Function Name	Description	Input(s)	Output(s)
bilinear_interp_vectorized(I, pt)	Performs bilinear interpolation for a given set of image points. Given the (x, y) location of a point in an input image, use the surrounding 4 pixels to compute the bilinearly-interpolated output pixel intensity. This function is for a *single* image band only - for RGB images, you will need to call the function once for each colour channel.	I, pt	b
compress(grid_x, grid_y)	Given a grid_x and grid_y of length "p", we can compress then into a 2xp numpy array.	grid_x, grid_y	comp
get3d22d(rawpts, allOmni, n)	Transforms 3D point cloud data to 2D points on a plane.	allOmni, n, rawpts	X, Y
getHomog(pos)	Get the Homogenous transform specified by elements in the pose array.	pos	H
getIntrinsicMat(f_x, f_y, c_x, c_y)	Returns the Camera intrinsics matrix.	f_x, f_y, c_x, c_y	K
~~getOMNI2GPS(sensor2omnicam)~~	Returns the transform from the omnidirectional camera to the GPS frame.	sens2omnicam	H
~~getOMNI2Rover(sensor2omnicam)~~	Returns the transform from the omnidirectional camera frame to the rover frame.	sens2omnicam	H
getOmniCamData(n, fileid, input_dir)	Returns the nth omni-directional camera image and the corresponding point cloud.	n, fileid, frame	X, Y
getOrthophoto(n, fileid, frame, input_dir, allOmni)	Gets the Orthophoto/BEV for omni camera 'n' at a specific timeframe.	n, fileid, frame	img, rgb, grid_x, grid_y
loadInfo()	Loads the provided intrinsic and extrinsic camera information provided in the .txt files in the run data.	N/A	allOmni, allT_OrefP
quat2rot(quat)	Converts rotations specified by quaternions to the rotation matrix form.	quat	C
sunCorrection(i)	Correct for the effect of the auto-exposure of the omnidirectional camera so that when stiching the images together there is some consistency.	i	comp
surroundOrthophoto(rgb0, transGrid0, rgb1, transGrid1, rgb2, transGrid2, rgb3, transGrid3, rgb4, transGrid4)	Gets the stiched surrounding orthophoto given all the individual orthophotos at a given time.	rgb[0,1,2,3,4], transGrid[0,1,2,3,4]	ALLPXrgb, grid_x, grid_y
transformGrid(transform, grid_x, grid_y)	Transform the grid for the orthophoto to the another frame (specified by "transform"). We need this because the orthophotos grids are in the frame of their respective stereo cameras. By transforming the grid you with this function, you have the coordinates in meters of each pixel in the orthophoto with respect to the a common frame (which you get to by applying the 'transform' which is given as an input to this function).	transform, grid_x, grid_y	ptcld_rover
~~transformPose(transform, x, y, C)~~	Transform a point orientation and position the orthophoto to the another frame (specified by 'transform'). Similar to transformGrid	transform, x, y, C	transform, grid_x, grid_y

Table 1: A Summary of all the functions used in this project. Functions with a ~~strike-through~~ were written but not used, they are relevant to the map building portion that was not done.

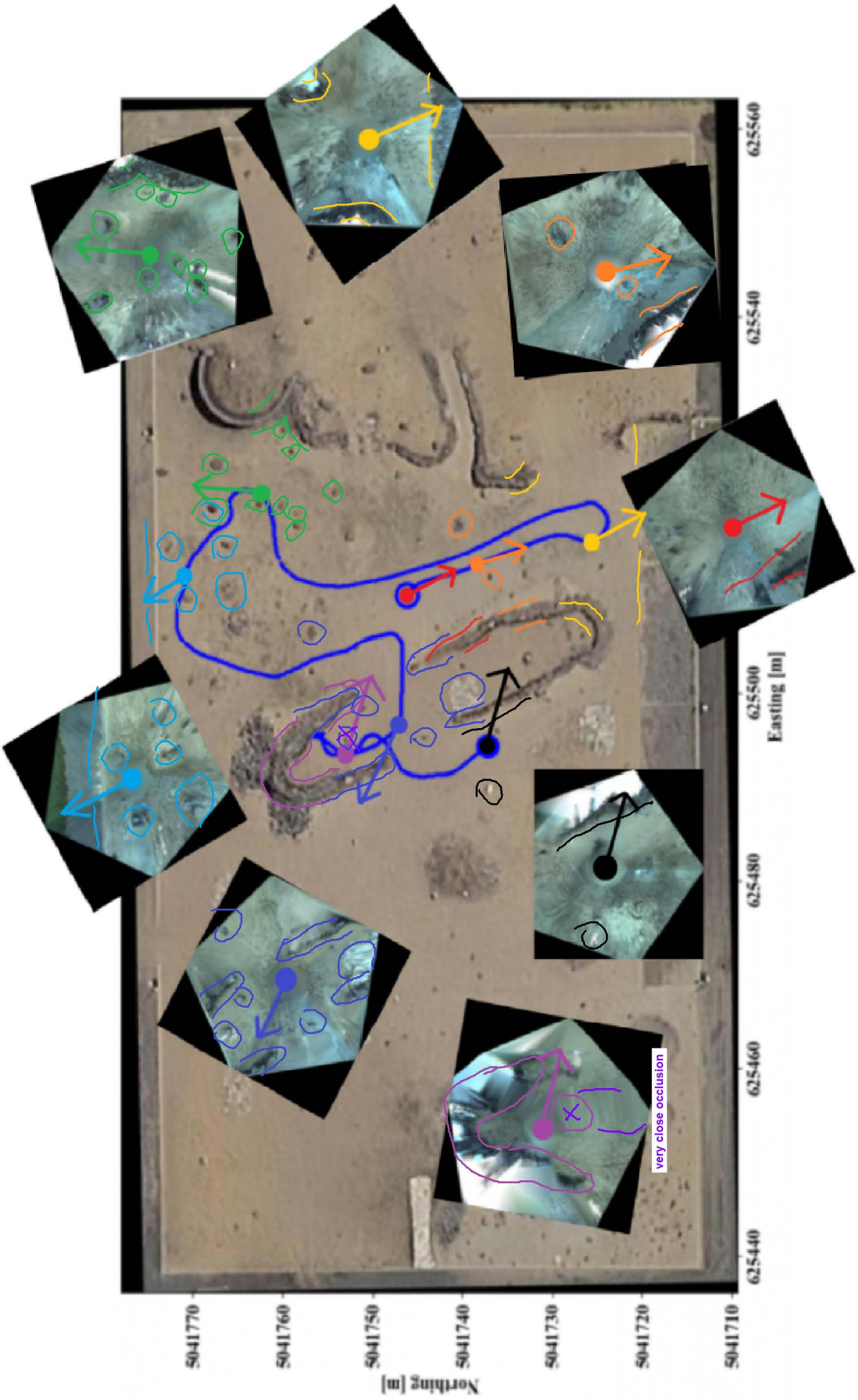

Figure 7: Aerial map comparison with the orthophotos generated at the time-points specified in Figure 6.