Project Report
Replicating Realtime Edge Based Visual Odometry for a Monocular Camera"

Joshua McCready (jmccread@umich.edu)
ECE 642 - Robotic Embedded Systems
University of Michigan - Dearborn

December 18, 2019

1 Introduction

The Picar provides a sensory starved platform with moderate computational power, not quite a contemporary cpu, and not quite an embedded processor. The initiative for the project was to augment the sensing capabilities of the Picar beyond what was asked in Experiment 3. I first explored some example's from Mathworks on structure from motion, and learned more about the visual odometry after a cursory search turned up tutorial slides from Davide Scarmauzza from the University of Zurich.

2 Narrowing on Visual Odometry

After that tutelage, I could effectively search for recent implementations of monocular visual odometry that focused on addressing the constraints of real time operation on mobile robotics with limited computational resources. I narrowed my focus on a series of papers published by Juan José Tarrio and Sol Pedre [2], [3]. Importantly, the authors have an easily accessible Github repository containing the implementation from [2], (link to Real-time Edge Based Visual Odometry for a Monocular Camera) that has both a real-time code base and one dedicated for processing of sample data sets. The sample date set part of the code base will be helpful in validating that I have correctly collected the dependencies.

They developed a novel method of Visual-based Simultaneous Localization and Mapping that operates on edges rather than feature based reconstruction that produce sparse reconstructions or direct methods that operate on most individual pixels and produce dense reconstructions. The Semi-Dense Structured Edge-Based Monocular SLAM that the authors present has the advantage of not being too sparse and not being too dense. Which allows both good reconstruction and the ablity to transmit maps over limited bandwidth channels. I offer a brief comparison of methods in the following table [3].

2.1 REBVO

REBVO's key features are real-time operation with all computations on-board for a monocular visual and IMU odometry fusion system who's depth-map can be presented to a remote operator. The IMU is used to determine real world scale, reduce processing requirements for the visual

Visual Odometry Method	**Representation Sparseness**	**Pro**	**Con**
Classical SLAM and VO using feature extraction	Sparse	Low drift	Poor in unknown environments
Full Image VO	Dense - each pixel	Full maps Low sensitivity to camera blur	Modern CPU + GPU required for real time
Hybrid with features and dense pixel regions	Semi-dense	Reduced computational expense Low sensitivity to camera blur	Complex
Edge based features + IMU	Semi-sparse	Possible to transmit map over low bandwidth Reduced computational expense	Requires IMU

odometry components by calculating orientation and pose. The visual odometry components of the system extract relative translation and rotation between frames.

The novel output of the REBVO algorithm is the edgemap, which is a list of "edge belonging points, [...] called keylines." [3]. The construction of the world comes from the keylines and their relationships. Each keyline has the parameters:

- q: A list of sub-pixel positions in the keyline
- m: Local perpendicular direction of the edge
- ρ: Estimated inverse depth
- σ_ρ: Estimated depth uncertainty
- n_{id}, p_{id}: Reference to next keyline in the both directions of the tangent

3 Prerequisites

3.1 Configuring the system for REBVO

As I learned, configuration is not trivial. Because of the timing of the Demo, I rushed through the configuration needed to use REBVO. In so doing, I installed Qt5 which promptly broke everything ROS. Despite trying to debug I was unable to get any *rqt* commands to work, each failing with new python errors. Consequently, I had to reimage. Once reimaged, I took my time with the package manager for Debian and everything got going just fine.

3.2 Getting an IMU

The fusion of acceleration and angular rate data in the visual odometry algorithm developed by [2] was vital to the algorithm's lightness and is required to run it. It was fairly painless to obtain an IMU for the Raspberry Pi and adapt it for use with ROS. I found the Raspberry Pi Sense HAT was not too pricey (40 USD) and had its own Python package (Python Sense-Hat). Additionally, someone had created a ROS node for sending the measurements that the Pi Sense HAT can do (Sense Hat ROS Node). The REBVO codebase subscribes $sensor\_\ msgs\backslash Imu$ messages so I modified

the available ROS node for the Sense Hat to publish the acceleration and angular rate measurements in the proper units.

The Sense HAT hat was easily installable with the existing hardware, some extra stand-offs, and a special long pin header. I disassembled the motor control board from the pi, installed the extra stand offs, and inserted the extra long header into the Raspberry Pi's GPIO header, then through the Sense Hat board and into the Motor Control board. Once everything was together, there were no conflicts with either the Servo function of the Picar or the DC motor operation.

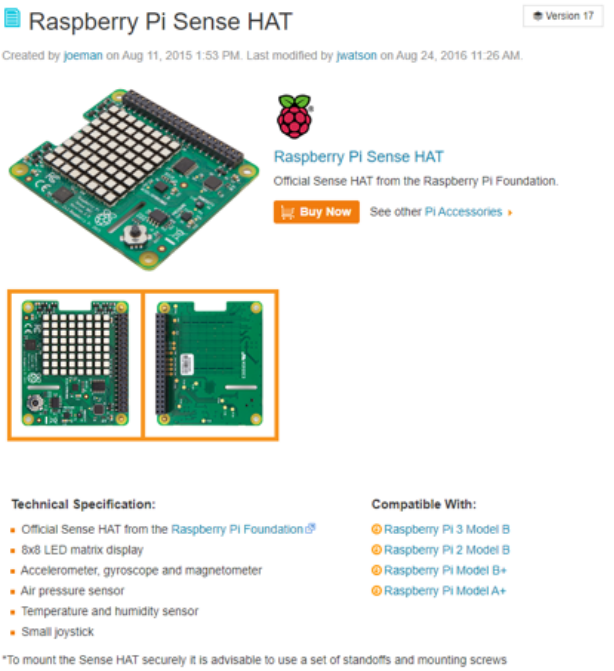

Figure 1: Overview of the Sense HAT for Raspberry Pi.

3.3 Required Parameters for REBVO

The authors of REBVO heavily parameterized the system function with configuration files for the standalone application they wrote and with .yaml file input for the ROS node that they wrote. The parameters needed to customize the algorithm for application with the Picar are listed and discussed in the following subsections. The final settings for these parameters are found in the *rebvo_monocular_Picar.yaml* file

3.3.1 Required Camera Parameters

These camera parameters were required by REBVO, some were simple settings from the usb_cam node. The others focal length, distortion parameters, and principle point were found from the calibration process in Matlab.

- ZfX and ZfY - Camera XY focal length
- PPx PPy - Camera principal point
- KcR2, KcR4, KcR6 - Distortion Parameters (some not used)
- ImageWidth and ImageHeight - Image size
- FPS - Frames per Second

3.4 Required IMU Parameters

The required IMU parameters were more intensive to gather. As can be seen below, REBVO requires detailed knowledge about the statistical behaviour of the IMU's bias (both accelerometer and gyro) and the kinematic relationships between the gyro and and camera's frame.

- transCam2Imu - Translation from camera center to IMU
- rotCam2Imu - Rotation Matrix from camera frame to IMU frame
- ImuMode - Data mode of the IMU (from file or real time data)
- SampleTime - Sample time from IMU ROS node

- InitBias - Whether or not there is an initial gyro bias - there is already filtering on the Picar gyro so not needed.
- InitBiasFrameNum
- BiasHintX - Hint for the gyro bias
- BiasHintY
- BiasHintZ
- g_module - Gravity constant (not on Mars!)
- GiroMeasStdDev - Standard deviation of gyro measurements at standstill
- GiroBiasStdDev - Standard deviation of gyro bias at standstill
- AcelMeasStdDev - Standard deviation of accelerometer measurements at standstill

In order to gather the gyro and accelerometer's standard deviation I recorded some measurements at standstill using ROS bags, the procedure is shown below.

```
%% Calculate some parameters of the IMU
close all

set(groot, 'defaultAxesTickLabelInterpreter','none');
bag = rosbag('2019-12-15-17-27-01.bag');

imu = select(bag,'Topic', '/sensehat/imu');
ts_imu = timeseries(imu);

%% AcelMeasStdDev
a1 = ts_imu.Data(:,11) - mean(ts_imu.Data(:,11));
a2 = ts_imu.Data(:,12) - mean(ts_imu.Data(:,12));
a3 = ts_imu.Data(:,13) - mean(ts_imu.Data(:,13));

a = [a1; a2; a3];
acc_std = std(a);

%% GiroBiasStdDev
g1 = ts_imu.Data(:,8);
g2 = ts_imu.Data(:,9);
g3 = ts_imu.Data(:,10);

g = [g1; g2; g3];
giro_std = std(g);

%% Gyro bias
mu_g1 = mean(g1);
mu_g2 = mean(g2);
mu_g3 = mean(g3);

%% https://www.mathworks.com/help/nav/ug/inertial-sensor-noise-analysis-using-
    allan-variance.html
```

Fortunately, the the python library for the Sense HAT does some filtering so these variances were not critical to the project function (on the order or 10^{-6}). However, the translation and and transformation from camera center to the IMU are of some importance. The camera frame was assumed to be the canonical frame with Z_C pointing out into the world and X_C pointing toward the bottom of the frame. The IMU frame was not assumed, and instead was determined by rotating the Picar until one of the channels output was positive gravity.

3.5 Required IMU Parameters and Orientation

Because the camera articulates, there were many paths to arrive at configuration for the orientation and position of the camera. However I chose the easiest one. Rather than implement a TF tree that would link by link and joint by joint describe the position of each articulations, I simply chose to relate them as links with no articulation. The servo is fixed at 30^o above horizontal looking straight ahead. Hence, the fixed relationship between the Odom frame, the IMU frame, and the Camera frame is shown in Figure 2. Not shown is the translation between F^{IMU} and F^{Camera}, for that I just measured with a ruler the best I could.

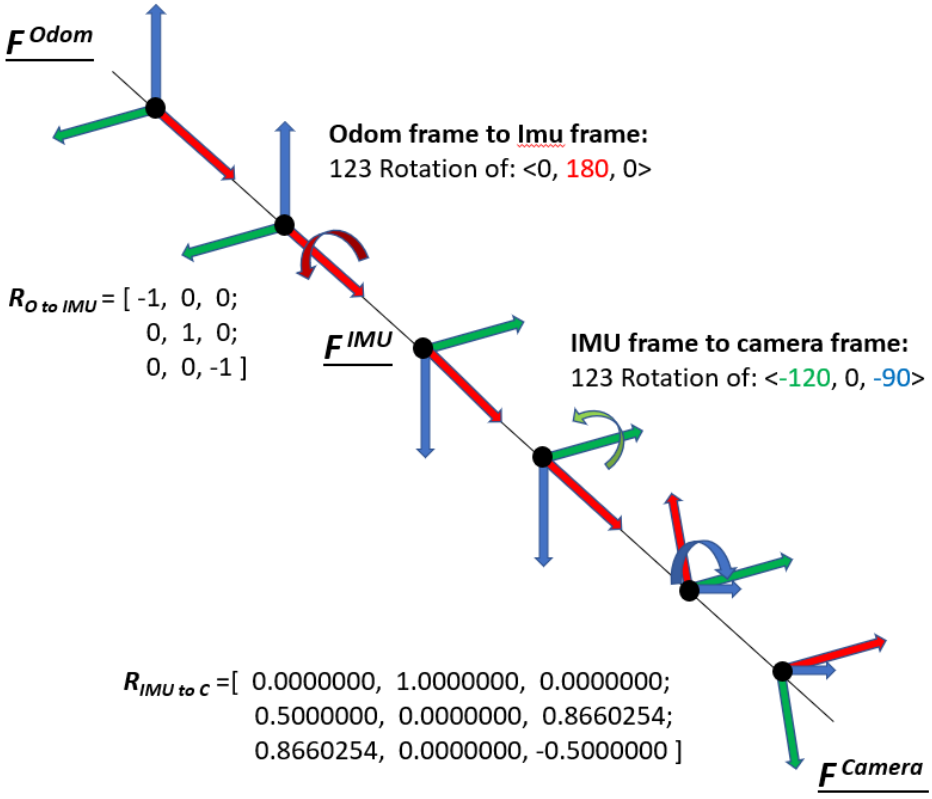

Figure 2: Successive coordinate transformations between Odom frame (assuming it's origin is co-located with that of the IMU), IMU frame, and Camera frames. Additionally, the rotation matrices from the 123 rotations are shown.

I would have preferred to use ROS for the kinematic relationship between joints. I even went so far as to start a URDF skeleton for the Picar (Figure 3), but as time ran short I gave up. Had I done the kinematic relationships in ROSier way, I might have been able to visualize the joint motion or the relationship between frames with Rviz. That visualization would have been invaluable because my spatial reasoning is weak and it would have helped me step through the right rotations. Additionally, a ROSier implementation of the kinematics would have allowed me to use the pan and tilt functionality of the servos to move the camera and have the visual odometer keep track of it by republishing the rosparams.

4 Assessment of Project Proposal Goals

In the project proposal I submitted I stated several goals. I have commented on my progress to achieving them and what artifacts I can offer to demonstrate their completion. There a decent Github readme.md that describes more or less has I can accomplish the following.

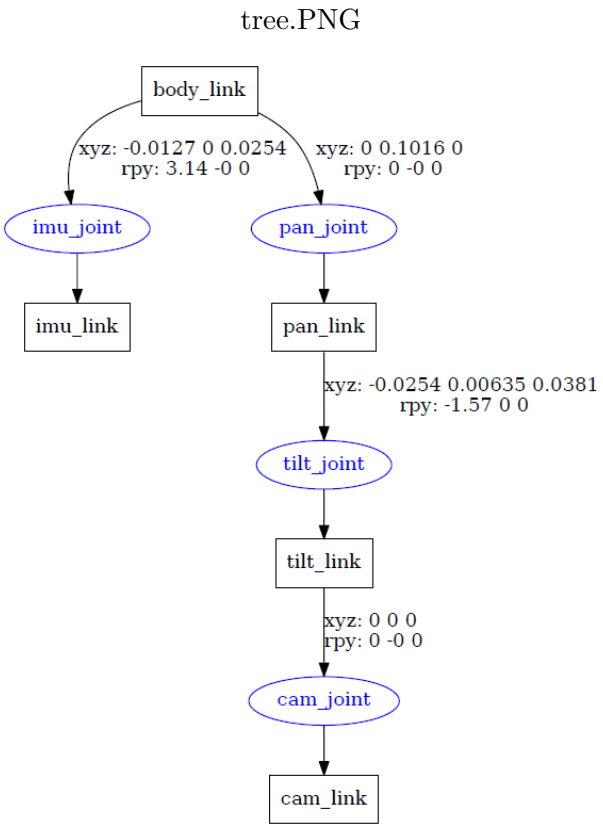

Figure 3

4.1 Replicating Run-time environment on the raspberry pi as required to build

After some initial and protracted struggles with the environment, I was able to build the software from the REBVO repository on the raspberry pi. I was able run the benchmark recorded data set version of the visual odometer system using the freely available EuRoC Micro Aerial Vehicle training dataset.

In my project proposal I said I would generate duplicates of the benchmark EuRoC data set's plots from [2] Section 6. I did not do this, again with time running short I felt it better to move on to configure the system for my application. It would have been valuable to see that the ground truth and the output of the REBVO algorithm matched what the authors put in their paper. Doing so might have allowed me to better understand what variables from REBVO to use in understanding the performance against the ground truth for the Picar. In any case, included in my submission is the file *Demo_Sample_DataSet_wAudio.mp4* that shows this part of the project.

4.2 Configure REBVO for the Picar for Real Time Operation

As discussed in the previous sections, I obtained an IMU and use the camera on the Picar along with some measurements I had previously done of the camera parameters with Matlab. More configurations were necessary for the system to work in real time. Chiefly, running the visualization the algorithm in situ with all the other nodes was infeasible. Luckily, the authors of REBVO considered this and allowed for the visualization to be done on a remote computer over UDP. To facilitate this, I read my raspberry pi's setup into a new image, and put it on another raspberry pi I have. Then I setup the host and client of UDP such that the REBVO visualization ran remotely on my other pi. Adding this complexity greatly reduced the number of dropped frames.

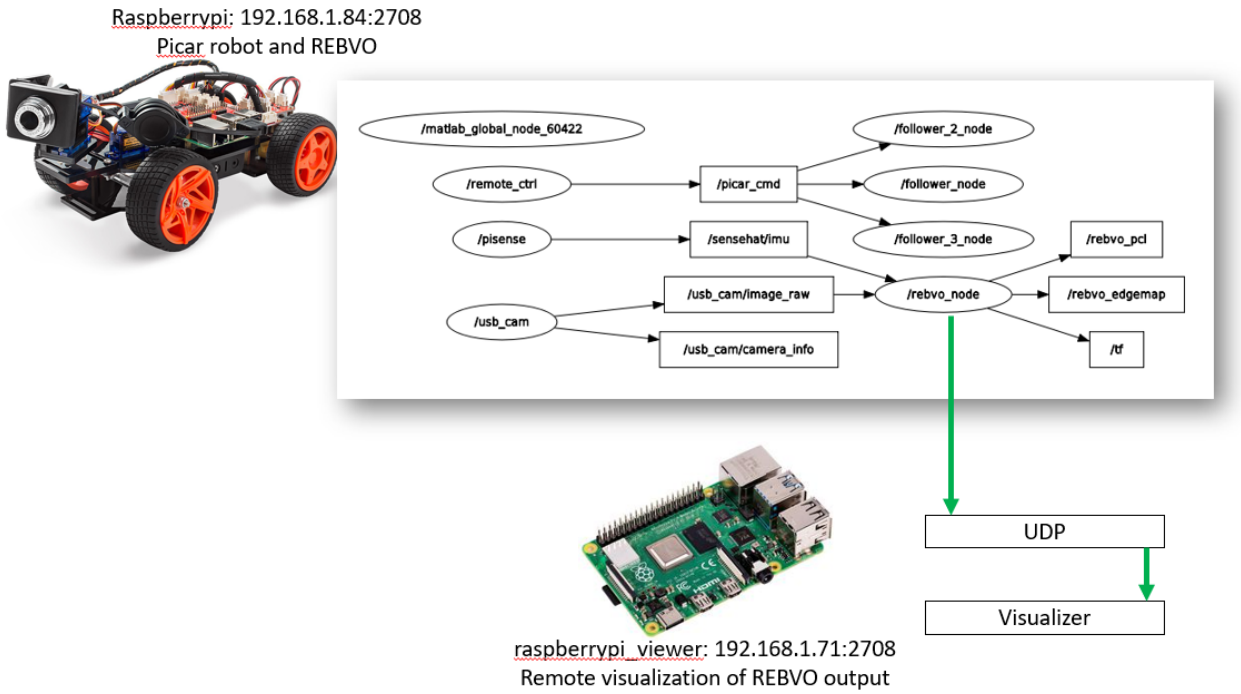

Figure 4: Shown in this figure is a contextual diagram of where what parts of the project were running. The Picar of course was the ROS master and was exporting the visualization of the REBVO algorithm's performance to another raspberry pi I had lying around my house. I only used another raspberry pi because it was the shortest path to a linux distro with the required libraries for REBVO.

4.3 Real-Time Operation with ROS

The Rebvo node shown in Figure 4 subscribes to the \imu and \image_raw topics. It publishes \rebvo_edgemap, \rebvo_pcl, and \tf. \rebvo_edgemap is a custom message defined by the REBVO package. \rebvo_pcl is type: sensor_msgs\PointCloud2, this is how depth is communicated without a special visualizer. The \tf message can be thought of as the pose and location of the camera as represented in the camera frame from the origin. If I had a better linux computer than a raspberry pi, these messages could be used in Rviz to show robot location, what it sees in its way. Rviz is too slow if it is run in situ with the rest of everything.

I have included a video file demonstrating the somewhat working REBVO depthmap. The output of the depthmap needed a lot of motion to form a full picture. This was challenging due to laggy behaviour and dropped frames. Rather than driving the car remotely as planned, I had to use the Picar more like a MAV to get the depthmap to do what it was supposed to do. Its not entirely surpising that this helped given that the author had tuned the algorithm for that application.

4.4 Whiz Bang Application of the Whole System...

I had initially proposed replicating the PID experiment and you had suggested that I use the ROS navigation stack to move the Picar autonomously. I would have liked to do either of those - but there were too many challenges that got in the way. This application would have been a really neat way to use RViz or Gazebo. I am disappointed that the basics got in the way of my exploration of ROSier work.

5 Conclusion

I bit off more than I could chew on this project. The biggest handicap was the configuration that was needed to get started. That said, I was able to replicate somewhat the function of the inertially enhanced edge based visual odometry algorithm REBVO. To refine its function I would have to dive deeper into the tuning of the REBVO algorithm parameters. It would also have helped if I had used a smaller image size because it would reduce the amount of processing and transmission time required for the system to function. I feel if I could have done a little less scrambling and perhaps if I had a better linux workstation to visualize the data in I could have gotten better results.

References

[1] J. Engel, T. Schöps, and D. Cremers, "Lsd-slam: Large-scale direct monocular slam," in *Computer Vision – ECCV 2014*, D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, Eds. Cham: Springer International Publishing, 2014, pp. 834–849.

Background on semi-dense V-Slam that was used as foundation for [2].

[2] J. J. Tarrio and S. Pedre, "Realtime Edge Based Visual Inertial Odometry for MAV Teleoperation in Indoor Environments," *Journal of Intelligent and Robotic Systems: Theory and Applications*, vol. 90, no. 1-2, pp. 235–252, 2018.

UAV application of Realtime Edge Based Visual Odometry for a Monocular Camera with benchmark against EuRoC dataset.

[3] J. J. Tarrio, C. Smitt, and S. Pedre, "SE-SLAM: Semi-Dense Structured Edge-Based Monocular SLAM," 2019. [Online]. Available: http://arxiv.org/abs/1909.03917

Most recent paper by these authors proposing and demonstrating Realtime Edge Based Visual Odometry for a Monocular Camera. It comments how the method has been iteratively improved.