Object Velocity Detector

Austin Alderton
ada0022@uah.edu
Terryn Fredrickson
taf0004@uah.edu
Christopher Eric Whatley
cew0028@uah.edu

Fall 2018

A project in partial fulfillment of CPE 656 Software Engineering Studio, the capstone course for the Masters in Software Engineering program at the University of Alabama in Huntsville

Project Description

The purpose of this capstone project is to lay the groundwork for an autonomous vehicle platform by creating its computer vision. Computer vision in an autonomous vehicle is the software systems and algorithms that acquire, process, analyze, and extract dimensional data from peripheral sensors. It serves as the eye's of the autonomous vehicle, allowing for the creation of steering, braking, and speed control, obstacle avoidance, and the implementation of driving rules. This project will detect objects around the future vehicle and calculate their velocities relative to the sensor array.

The computer vision developed uses four major components: a Velodyne VLP-16 LIDAR, two PT Grey Blackfly BFLY-U3-2356C-C cameras, and a TensorFlow convolutional neural network to perform object identification and classification. Camera images are passed through the object detection neural network, generating bounding boxes in pixels around identified content. A point cloud of 3D samples is generated per complete revolution of the VLP-16. These points are translated and transformed to camera field of view. The point cloud is then projected to the 2D image coordinate system using the camera specific projection matrix. The visible points in the cloud are then matched to the bounding boxes, and a distance is calculated from the LIDAR points painting the detected object. The bounding boxes and distances are then stored and time stamped, and if the next cycle's bounding boxes are determined to represent the same object, a velocity is calculated.

Mathematical Overview

This project uses simple Euclidean transformations, translations, and projection image geometry to map LIDAR points to image pixels. Below is a brief summary on the mathematics performed in this project.

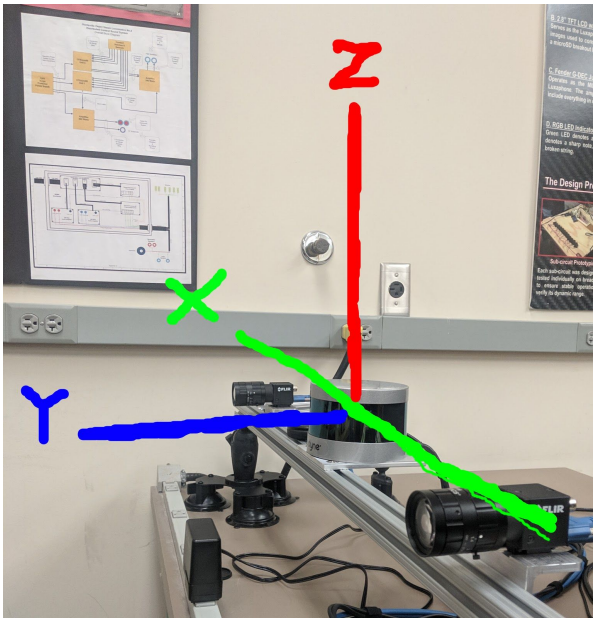
Point Cloud Transformations

Point clouds are groupings of 3D points representing returned samples from laser scans. Devices including LIDARs enumerate returned laser reflections as 3D cartesian coordinates with the device emitter as the origin for the graph. These clouds of points can be treated as rigid bodies, meaning every point in the cloud can be rotated and translated as if the cloud was a single rigid object. Two of these rigid body transformations were performed to map the point clouds to the camera's field of view in 3D space.

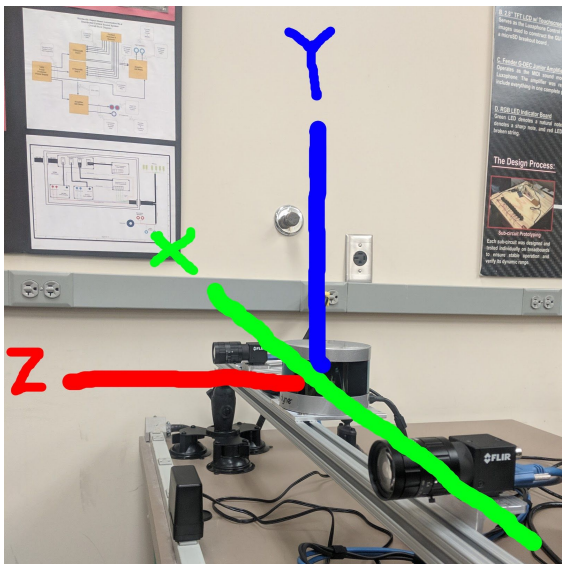
Rotation

After a point cloud is generated from a full 360 degree sweep, the first transformation performed is an affine rotation. An affine transformation performs collinearity, meaning all points lying on a line initially still lie on a line after the transformations. Affine transformations

also preserve ratios of distances between points. This kind of transformation is used for axis manipulation of the 3D coordinate system used in the scan from the LIDAR. When the point cloud is generated and returned from the VLP-16, the axis structure has the Y-axis pointing straight ahead out of the LIDAR puck, the X-axis on the horizontal plane, and the Z-axis pointing up to the ceiling.



To properly perform a point projection onto the camera image coordinate system, the Z-axis needs to represent the image depth as opposed to height, and the Y-axis needs to represent image height. The affine transformation performs this rotation. The corrected axis is shown below.



Translation

To properly align LIDAR points with respect to the camera field of view, the entire point cloud must also provide an offset as if the camera, not the LIDAR, was the origin. Since the camera and LIDAR are attached to the same stabilization beam, the offset is simply the distance from the camera to the LIDAR. The desired movement is then a simple rigid body translation to the left or right to center the point cloud's origin over the camera instead of the LIDAR, with the movement amount equal to the distance between the LIDAR and camera.

Projecting points onto camera image using a project matrix

A projection matrix is a matrix that takes a coordinate from a vector space to a subspace of a lower dimension. In the context of a camera, it takes real-world 3D coordinates, converts them to a 3D coordinate system relative to the camera, and then maps the camera coordinates to a 2D pixel coordinate. To perform this calculation, a 3x4 matrix is multiplied by a vector of real-world coordinates, and a pixel mapping is produced. To summarize, a real-world coordinate (X,Y,Z) is multipled by the projection matrix to create (u,v), which corresponds to a pixel in an image. A camera projection matrix is shown below.

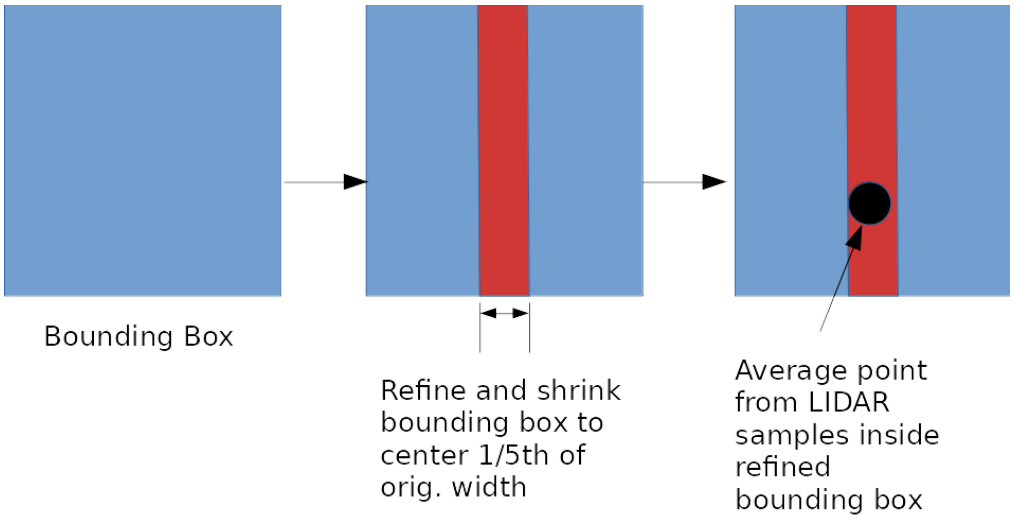
$$\mathbf{P} = \begin{bmatrix} f & 0 & p_x & 0 \\ 0 & f & p_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In the projection matrix, the top left f is equivalent to f*(X/Z), where f is the focal length of the camera, Z is the real-world distance from the camera, and X is the real-world x-component of the point in cartesian space. The f in the second column is equivalent to f*(Y/Z), where f is the

focal length of the camera, Z is the real-world distance from the camera, and Y is the real-world y-component of the point in cartesian space. Px an and py are the x and y offsets from the ideal center of the projection.

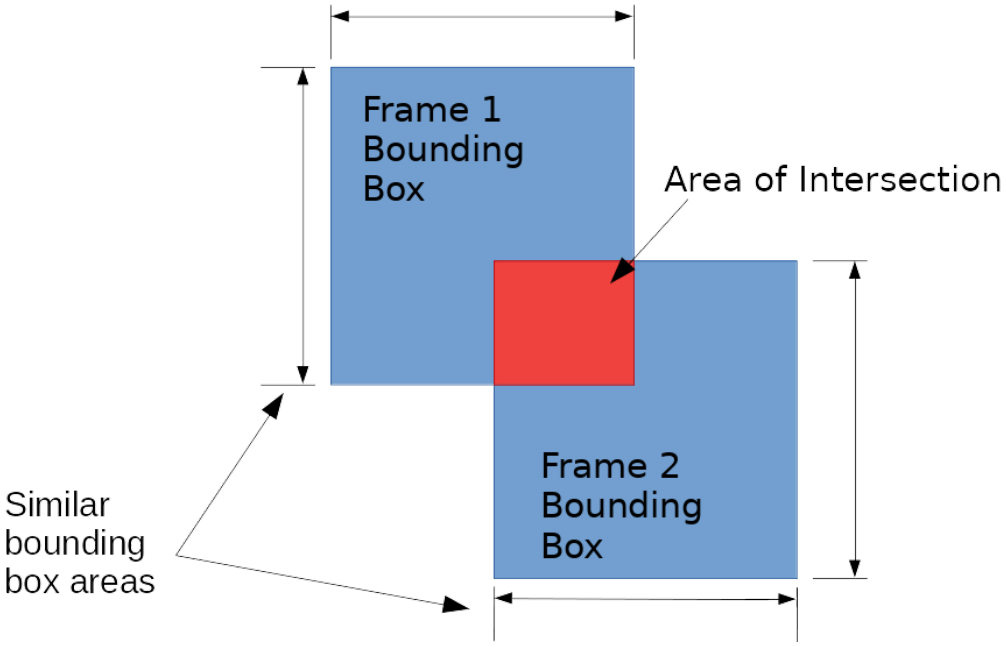
Object surface distance calculation

Once LIDAR points are collected that correspond to a bounding box surrounding a detected object, a refinement of the bounding box occurs. The bounding box is refined to 1/5th of the original width, while the height is preserved. From there, the LIDAR point cloud sub-set inside the bounding box is further refined to those points inside the new bounding box. Each vector component of the 3D cartesian points (ie, (x,y,z)) inside the refined box are averaged to create a single, average point representing the surface distance from the LIDAR origin. Below is a figure visually demonstrating the algorithm.



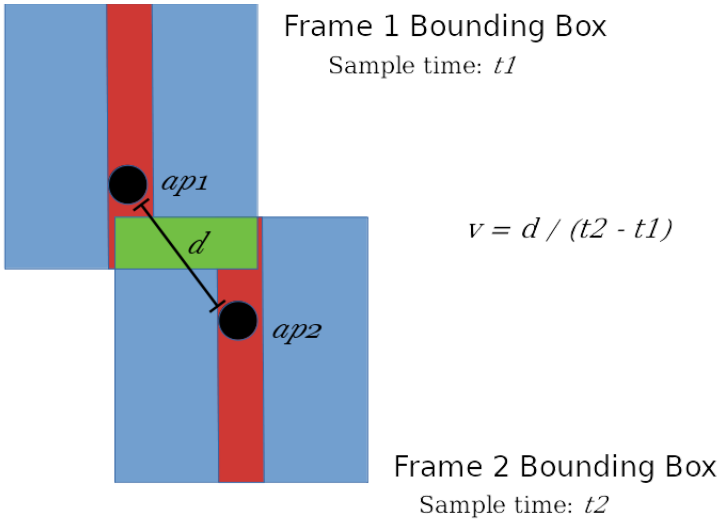
Tracking detected objects across frames

TensorFlow provides no mechanism to maintain unique identifiers across each image frame the neural network performs work on. As a result, a rudimentary heuristic was developed to introduce confidence in object identification across image frames. The approach used comprises of two comparisons. First, a current image frame bounding box is compared to bounding boxes from the previous frame. If the areas of the bounding boxes are within 80 to 120 percent of each other, and the the previous and current bounding box overlap, then they are considered the same object. Below is a visual representation of the described algorithm.



Calculating Object Velocity

With the ability to correlate objects across image frames and a means to generate a surface distance representation, a velocity can be calculated assuming that each sample is assigned an absolute timestamp. Subtracting the last correlated sample's timestamp to its previous one, a near spontaneous, absolute velocity across two camera frames can be calculated. A summary image is shown is below.



Assumption: The bounding boxes have been determined to represent the same object across image frames

Software Description

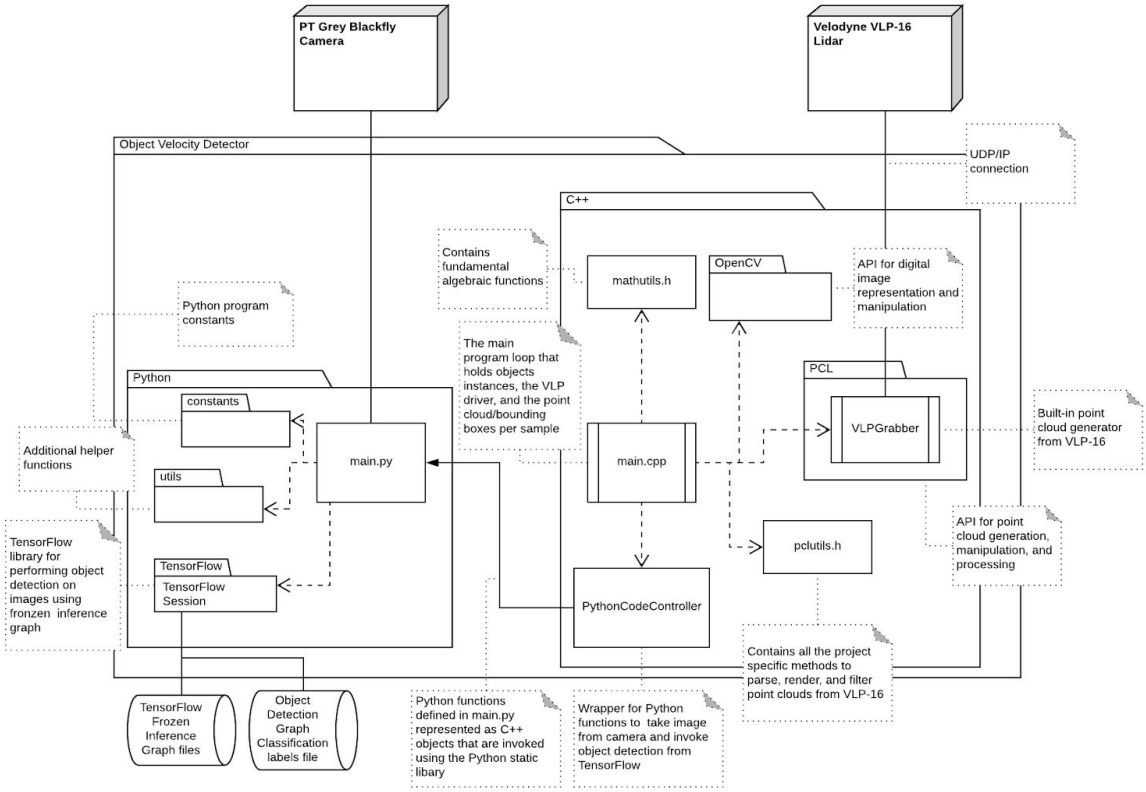
System Overview

The devised software system takes a camera image from a connected PT Grey Blackfly camera and passes the image through an instance of the TensorFlow neural network operating on a frozen inference graph that is used to classify and identify objects. Simultaneously, a point cloud of relative 3D points is collected from a full rotation of the Velodyne VLP-16 lidar. Once the point cloud has been generated, two rigid body transformations are performed on the entire point cloud. First, the point cloud is translated to the left or right to center the point cloud in front of the aperture of either camera (as the VLP-16 lies at the center between to the two cameras). After the initial translation, another transformation is made. When the point cloud is initially generated, the point cloud is rendered in a manner where the Y-axis is pointing out of the camera aperture perpendicular to the attachment bar where the LIDAR and cameras are placed, but on the same horizontal plane. The second transformation moves the points in the cloud to position the X-axis as parallel to the attachment bar, the Y-axis perpendicular to the attachment bar piercing the ceiling, and the Z-axis pointing out from the camera lens. These two transformations must be performed to apply a projection matrix to each point to map the lidar sample to the camera image.

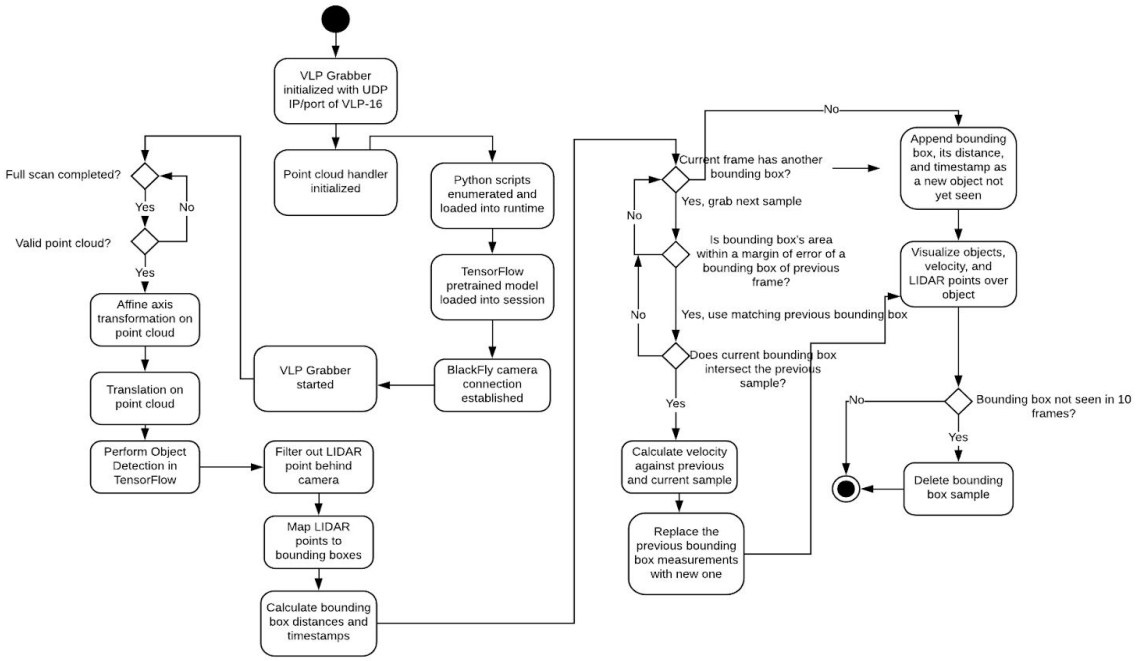
Once an image has been processed through the neural network, the resulting output includes the image-specific bounding boxes surrounding the detected objects. The program then collects these detections and stores them temporarily. The point cloud is then filtered for points not potentially visible in the camera field of view. The bounding boxes are then are turned into rectangular frames, and for each point in the point cloud, the camera projection matrix is applied to map the 3D location of the point to the 2D camera frame of reference. If the LIDAR point lies within the bounding box, the relative distance from the center of the emitter to the point in the cloud is generated.

After the LIDAR points are mapped to the detect object bounding boxes, the visible points are colored based on the minimum and maximum distances detected among the visible points. These points and bounding boxes are then stored. Upon next algorithm iteration, the stored bounding boxes are tested against the latest iteration of collected boxes and visible clouds, and if a historical correlation can be established, the object velocity is calculated using a normalization of the collected LIDAR distance samples in the bounding boxes and timestamps.

System Block Diagram



Activity Diagram



File Descriptions

ObjectVelocityDetector/ObjectVelocityDetector/src/main.cpp - contains the brains behind the entire project. Has the main loop that processes image data and lidar data and merges them.

main() - First, the initial arguments are parsed, the point cloud is initialized, and then the point cloud color handler is initialized. A frame and boxes are also defined, then the Point Cloud Callback Function and mutex is defined. Afterward, there is some transformation logic that is explained in the **Mathematical Overview** section of this document. The point cloud color handlers is then defined as well as the VLP Grabber. Then the Callback Function is defined. The transforms and the python script controllers are initialized and then started. This start method is defined in **main.py** and called via the PythonCodeController object which is defined in **pythoncodecontroller.cpp**.

Afterward comes the main loop. The mutex is locked. Then, if the mutex locked and the cloud exists, we begin the main processing. The main processing starts by doing a spinOnCamera1 which sets the image and the bounding boxes on the pcc object. Then the image and bounding boxes are gotten from the pcc object. Afterward, a frame is defined and the image and visible points are projected onto the frame via the **pclutils.h** file.. Then the window displaying the output data is updated. The handler is then updated to the current cloud and the

loop repeats. After the loop is exited by pressing q, all the objects are cleaned up and the connection closed.

ObjectVelocityDetector/ObjectVelocityDetector/src/pythoncodecontroller.cpp - is the wrapper that defines the mapping between the C++ and the calls that are written in python in the **main.py** file.

PythonCodeController() - constructor. The constructor starts by creating the **start** method of the **main.py** file and then creates the **execute_session** method of the **main.py** file for the camera. Then **get_image** and **get_boxes** is created which will get the latest images and boxes that are generated by the **execute_session** call.

ObjectVelocityDetector/ObjectVelocityDetector/main.py - is the main python code that accesses the cameras and produces the image data and bounding boxes via TensorFlow.

start() - loads the TensorFlow graph, the categorization indices, the TensorFlow sessions, and the PTGrey cameras.

stop() - stops the camera captures.

execute_session() - runs a session of TensorFlow on a particular image frame given a camera id. The image is first converted to a numpy array and then expanded to expected dimensions. Then the boxes are initialized as well as the scores (level of confidence) and classes (what the object is). Then the count or num_detections is initialized of how many objects have been detected. Finally, after all these objects have been defined, the actual detection is gotten. Once all of that is done, the final image is rendered that contains the bounding boxes and all the information that was collected.

get_image() - gets the last image that was generated by **execute_session**.

get_boxes() - gets the last set of bounding boxes that were generated by **execute_session**

ObjectVelocityDetector/ObjectVelocityDetector/include/mathutils.h - is the main module that contains that math that calculates and returns camera angles and distance vectors.

ObjectVelocityDetector/ObjectVelocityDetector/include/pclutils.h - is the main module that contains all the math having to do with the Point Cloud Library and Lidar. It projects the points onto a frame, generates heat maps, filters bounding boxes, generates PointClouds, performs transforms, and several other important functions that are used in the main methods.

Dependencies and Requirements

Basic Overview of Used Libraries

Before examining the source code, a fundamental understanding of the library dependencies is recommended to better understand what the source code's operations do.

TensorFlow

TensorFlow is an open source software library for high performance numerical computation and machine learning. What this translates to is a platform from which neural networks can be developed to perform data analysis/ These neural networks can then be trained using deep or hierarchical learning to fine tune indeterministic results by providing expected input to drive future output It is platform agnostic in both operating system and computer architecture. It can be installed on Windows, Mac OS X, and GNU/Linux distributions and can flawlessly run on any X86 chipset or NVIDIA GPU supporting CUDA 9.0 or greater for additional parallelized acceleration. TensorFlow is written in C++ with its primary language of operation being Python.

One of the major application focuses of TensorFlow is object detection. There are many available neural network structructures to perform object classification and identification on an input image or camera frame. TensorFlow provides an expansive repository of learning models for many different trainable behaviors, including object detection. These kinds of neural networks take input images and if the network detects an object, bounds the object in a rectangular box, classifies it, and returns the list of all present objects. TensorFlow also provides pre-trained neural networks to download an use, avoiding the necessity to train them for immediate use.

PCL

PCL, or Point Cloud Library, is an open source library of algorithms for point cloud processing tasks and 3D geometric processing. Some of its capabilities include feature estimation, surface reconstruction, point cloud generation, and point cloud geometric transformation. The strength of PCL comes from its built in driver classes that generate point cloud instances from common external hardware, including the most commonly used LIDAR peripheral sensors. PCL has useful, optimized algorithms to manipulate, transform and translate point clouds as an entire rigid bod, as well as, parse point clouds in separate entities

OpenCV

OpenCV is an open source library that is to aid in the development of computer vision. The library contains an extensive list of optimized algorithms to display, filter, perform feature extraction, overlay, recognize scenery, create markers, stitch stereoscopic images together to generate a 3D model, and edit image frames in real-time.

CMake

CMake is a cross platform software suite that manages the build process. It is platform agnostic, making CMake build script definitions an intermediary format to define executables, library dependencies, select output directories, compiler and linker directives, apply compiler

extensions, and perform other optional compiler features. CMake is used to generate operating system specific build files, such as Makefiles for Linux and Visual Studio projects in Windows.

PyCapture2 and FlyCapture SDK

FlyCapture SDK and its Python language binding PyCapture2 is a software development kit to interface with PT Grey's entire line of camera systems. FlyCapture SDK allows user applications to take raw images from a connected PT Grey sensor, configure image resolution, provide color filtering, change encoding, change zoom level, and countless other instrument controls.

Dependency Installation

System Requirements

The system requirements for this project is listed below. Installation guidelines will be provided in the following sections and written from the perspective of installing the software on Ubuntu 16.04

Hardware requirements

- Ubuntu 16.04 (the installation instructions were written for this version of Linux, although most of the dependencies are compatible with 18.04 through recompilation
- A computer with 2x USB 3.0 ports
- Optional: NVIDIA GPU compatible with Compute Capability 3.5 or higher (to meet any real-time requirements for data processing, the GPU accelerated version of TensorFlow should be used)

Software requirements

- Python 2.7.12 or greater**
- TensorFlow (installed using pip, the Python package manager)
- TensorFlow model research libraries
- FlyCapture SDK
- PyCapture
- NVIDIA proprietary drivers for Linux 384.x and greater*
- CUDA Toolkit 9.0 (this is the only version currently working with TensorFlow*
- cuDNN SDK 7.0 or greater compatible with CUDA 9.0*
- PCL 1.8.1
- OpenCV 2.x

* Required for the GPU version of TensorFlow. If the CPU version is chosen instead, these dependencies are unnecessary.
** Python 2.7.x should be standard with any Ubuntu 16.04 install. Type *python* in the terminal to verify.

Overview

The software system is divided between two runtimes: Python and natively compiled C++. The C++ program drives the Python code by representing Python functions as objects and individually invoking them. This is done using the embedded Python runtime, which is an API defined in every Python installation through packaged header files. The Python runtime generates an image from the connected PT Grey cameras, and passes the image through the TensorFlow object detection neural network. Additional functions return the last iteration results of the object detection and the image to the native C++ call stack. From there, the point clouds are generated, transformed, and filtered for visibility in the camera field of view, and then mapped to the image representation using the camera projection matrix. The C++ source code performs the historical mapping, displaying, and velocity determination as well. Below are descriptions, functional roles, and installation directions for the open source libraries used in this project.

Installing NVIDIA Dependencies

The following installation process is to fulfil the software dependencies for the GPU version of TensorFlow. If the CPU version is intended to be used instead, this section can be ignored.

1) Install the NVIDIA Graphics driver:

    sudo apt-get install nvidia-384 nvidia-modprobe

Make sure that secure boot is disabled. A prompt may appear to reboot the system to do this. This option is accessible in the computer BIOS.

2) Download CUDA 9.0:

    wget https://developer.nvidia.com/compute/cuda/9.0/Prod/local_installers/cuda_9.0.176_384.81_linux-run

3) There are three installers contained in the package downloaded in 2). Only two of them need to be installed. Extract them:

    chmod +x cuda_9.0.176_384.81_linux-run

    ./cuda_9.0.176_384.81_linux-run --extract=<directory to extract to>

4) Install the CUDA Toolkit 9.0:

    sudo ./cuda-linux.9.0.176-22781540.run

5) Install the CUDA samples:

    sudo ./cuda-samples.9.0.176-22781540-linux.run

6) Configure the runtime library:

    sudo bash -c "echo /usr/local/cuda/lib64/ > /etc/ld.so.conf.d/cuda.conf"

    sudo ldconfig

7) Add the CUDA executables to the path in */etc/environments*. The path to add is */usr/local/cuda/bin.* Reboot the computer

8) Compile the samples to make sure the installation worked:

cd /usr/local/cuda-9.0/samples
sudo make

9) Run the device query to check the installation, At least one CUDA device should be found if the installation was successful.

cd /usr/local/cuda/samples/bin/x86_64/linux/release
./deviceQuery

10) Download the cuDNN v7.0.x library for Linux. An NVIDIA developer account must be created to download it. Make sure the cuDNN version is compatible with CUDA 9.0. There should be 3 .deb files to download: libcudnn7, libcudnn7-dev, and libcudnn7-doc

11) Install each package using dpkg

sudo dpkg -i libcudnn7_7.0.5.15–1+cuda9.0_amd64.deb
sudo dpkg -i libcudnn7-dev_7.0.5.15–1+cuda9.0_amd64.deb
sudo dpkg -i libcudnn7-doc_7.0.5.15–1+cuda9.0_amd64.deb

12) Make the cuDNN libraries available across the system:

sudo bash -c "echo /usr/local/cuda/extras/CUPTI/lib64 > /etc/ld.so.conf.d/cudnn.conf"
sudo ldconfig

13) Reboot

Installing TensorFlow

This section installs the TensorFlow platform.

1) Begin by installing pip, the Python dependencies manager:

sudo apt-get install python-pip

2) Install TensorFlow:

pip install tensorflow-gpu

3) Verify the TensorFlow installation:

python -c "import tensorflow as tf; tf.enable_eager_execution(); print(tf.reduce_sum(tf.random_normal([1000, 1000])))"

Installing Necessary Python Libraries

This project and its dependencies, mainly the TensorFlow models research repository, requires some Python libraries. This section describes how to install him.

1) Use pip to install the necessary libraries:

pip install --user Cython
pip install --user contextlib2
pip install --user pillow
pip install --user lxml
pip install --user jupyter
pip install --user matplotlib
pip install --user numpy
Pip install --user opencv-python

2) Using apt, install the Python shared runtime library, static library, and header files.

sudo apt-get install libpython
sudo apt-get install libpython-dev

Installing TensorFlow Object Detection Models

1) Clone or download the following git repository:

https://github.com/tensorflow/models/

2) Download the protoc compiler here, and make sure the path to the executable is known. Use the 3.5.X version.

https://github.com/protocolbuffers/protobuf/releases/tag/v3.5.1

3) Compile the .proto interface definitions using the downloaded protoc compiler.

Set the current director to *<path to>models/research :*

cd <path to>/models/research

First, try to recursively find all *.protos and run the compiler on them:

[path to protoc_executable] object_detection/protos/.proto --python_out=.*

If that doesn't work, from the *models/research/object_detection/protos* directory, run the following commands:

[path to protoc_executable] anchor_generator.proto --python_out=.
[path to protoc_executable] argmax_matcher.proto --python_out=.
[path to protoc_executable] bipartite_matcher.proto --python_out=.
[path to protoc_executable] box_coder.proto --python_out=.
[path to protoc_executable] box_predictor.proto --python_out=.
[path to protoc_executable] eval.proto --python_out=.
[path to protoc_executable] faster_rcnn.proto --python_out=.
[path to protoc_executable] faster_rcnn_box_coder.proto --python_out=.
[path to protoc_executable] graph_rewriter.proto --python_out=.
[path to protoc_executable] grid_anchor_generator.proto --python_out=.
[path to protoc_executable] hyperparams.proto --python_out=.
[path to protoc_executable] image_resizer.proto --python_out=.
[path to protoc_executable] input_reader.proto --python_out=.

[path to protoc_executable] keypoint_box_coder.proto --python_out=.
[path to protoc_executable] losses.proto --python_out=.
[path to protoc_executable] matcher.proto --python_out=.
[path to protoc_executable] mean_stddev_box_coder.proto --python_out=.
[path to protoc_executable] model.proto --python_out=.
[path to protoc_executable] multiscale_anchor_generator.proto --python_out=.
[path to protoc_executable] optimizer.proto --python_out=.
[path to protoc_executable] pipeline.proto --python_out=.
[path to protoc_executable] post_processing.proto --python_out=.
[path to protoc_executable] preprocessor.proto --python_out=.
[path to protoc_executable] region_similarity_calculator.proto --python_out=.
[path to protoc_executable] square_box_coder.proto --python_out=.
[path to protoc_executable] ssd.proto --python_out=.
[path to protoc_executable] ssd_anchor_generator.proto --python_out=.
[path to protoc_executable] string_int_label_map.proto --python_out=.
[path to protoc_executable] train.proto --python_out=.

4) Add the full directory paths to the OS system environment variables:
   [fully qualified path here]/models/research
   [fully qualified path here]/models/research/slim

In Ubuntu 16.04, this is done by putting the following line in the */etc/environment* variable:

export PYTHONPATH=$PYTHONPATH:[fully qualified path here]/models/research:[fully qualified path here]/models/research/slim

5) Restart your computer, and attempt to run following command to verify the configuration:
   python [fully qualified path here]/models/research/object_detection/builders/model_builder_test.py

Installing OpenCV

1) Use apt to install the OpenCV2 library:
   sudo apt-get install libopencv-dev

Installing CMake

1) Use apt to install CMake
   sudo apt-get install cmake

Compiling and Installing PCL 1.8.1

PCL 1.8 does not come in the apt package manager on Ubuntu 16.04. It is required because version 1.8 contains a particular class, VLPGrabber, that serves as an all-in-one driver for the VLP-16. This means it must be compiled and installed manually.

1) Download the PCL 1.8.1 source code here:
   https://github.com/PointCloudLibrary/pcl/archive/pcl-1.8.1.zip

2) Install the following dependencies using apt:

sudo apt-get install g++
sudo apt-get install cmake cmake-gui
sudo apt-get install doxygen
sudo apt-get install mpi-default-dev openmpi-bin openmpi-common
sudo apt-get install libflann1.8 libflann-dev
sudo apt-get install libeigen3-dev
sudo apt-get install libboost-all-dev
sudo apt-get install libvtk6-dev libvtk6.2 libvtk6.2-qt
sudo apt-get install 'libqhull'*
sudo apt-get install libusb-dev
sudo apt-get install libgtest-dev
sudo apt-get install git-core freeglut3-dev pkg-config
sudo apt-get install build-essential libxmu-dev libxi-dev
sudo apt-get install libusb-1.0-0-dev graphviz mono-complete
sudo apt-get install qt-sdk openjdk-9-jdk openjdk-9-jre
sudo apt-get install phonon-backend-gstreamer
sudo apt-get install phonon-backend-vlc
sudo apt-get install libopenni-dev libopenni2-dev
sudo apt-get install libproj9
sudo apt-get install libproj-dev
sudo apt-get install libpcap0.8

3) Unzip the downloaded source code, and set it to the current directory:

cd <unzipped directory>

4) Create a directory called release, and set it to the current directory:

mkdir release
cd release

5) Set the CMake environment flags for the compilation and generate the make file:

cmake -DCMAKE_BUILD_TYPE=None -DBUILD_GPU=ON -DBUILD_apps=ON -DBUILD_examples=ON ..

6) Compile the library using make:

make

7) Install the compiled libraries:

sudo make install

Installing FlyCapture SDK and PyCapture

1) Create a PT Grey account to download the necessary software from here:
https://www.ptgrey.com/Downloads/

2) Select Blackfly for the Product Families, BFLY-U3-2356C-C for the Camera Models, and Linux for the Operating Systems

3) Download the FlyCapture Full SDK for Ubuntu 16.04
4) Download the FlyCapture Python Wrapper for Ubuntu 16.04

5) Install the FlyCapture SDK by decompressing the downloaded package and following the contained README.

6) Install the PyCapture wrapper by decompressing the downloaded package and following the contained README.

Compiling the Program

This program uses CMake to manage the build process. CMake provides a level of system agnosticism to make it simpler to handler build constraints, dependency resolution, and linker options. CMake also allows for better integrated development environment (IDE) support. This project has been successfully developed in KDevelop, CLion, and QT Creator. Each IDE has slightly different ways to integrate with CMake, so they will not be discussed. This section will only describe how to compile the program using CMake to generate a Linux Makefile, the traditional mechanism to compile native code.

To compile this program, a Makefile must first be generated. The simplest way to do this is to use the CMake GUI. Ensure the CMake and the CMake GUI are installed using apt:

sudo apt-get install cmake
sudo apt-get install cmake-qt-gui

Next, launch the CMake GUI. It should be visible in the Ubuntu 16.04 start menu it its searched for.

Navigate to the the location of the source code after selecting the “Browse Source” button. The final path should like something like this:

~/<some accessible path after the home directory>/ObjectVelocityDetector/ObjectVelocityDetector

Set the desired location to build the compiled binaries. It is highly recommended to place the binaries into a folder called “build” in the source code folder, because a CMake task will copy over the binary to the location it must be executed in. An example of the recommended path is below:

~/<some accessible path after the home directory>/ObjectVelocityDetector/ObjectVelocityDetector/build

Once the build directory and source code directory have been set, press the “Configure” button. After the dependency resolution occurs, most of the names and values of the compile environment variables will show up colored red.

If the necessary dependencies were all installed correctly from the previous section, the only variable that should need to be set is *CMAKE_BUILD_TYPE*. For the sake of simplicity and to make sure the program can be attached to a debugger, this variable should be set to “Debug”

Once the build type has been set, press the “Generate” button. If “Generating Done” is seen in the program output with no error prompt, the Makefile generation was successful. Navigate to the set build directory:

cd ~/<some accessible path after the home directory>/ObjectVelocityDetector/ObjectVelocityDetector/build

If the files are listed, a filed called “Makefile” should be visible. Use the make command to generate the executable:

make

Once the program is compiled, an executable called “ObjectVelocityDetector” should be in the “build’ directory. This executable should have also been automatically copied to the directory below the “build” directory, at:

~/<some accessible path after the home directory>/ObjectVelocityDetector/ObjectVelocityDetector

If the executable was copied to this parent directory successfully, the compilation process was successful.

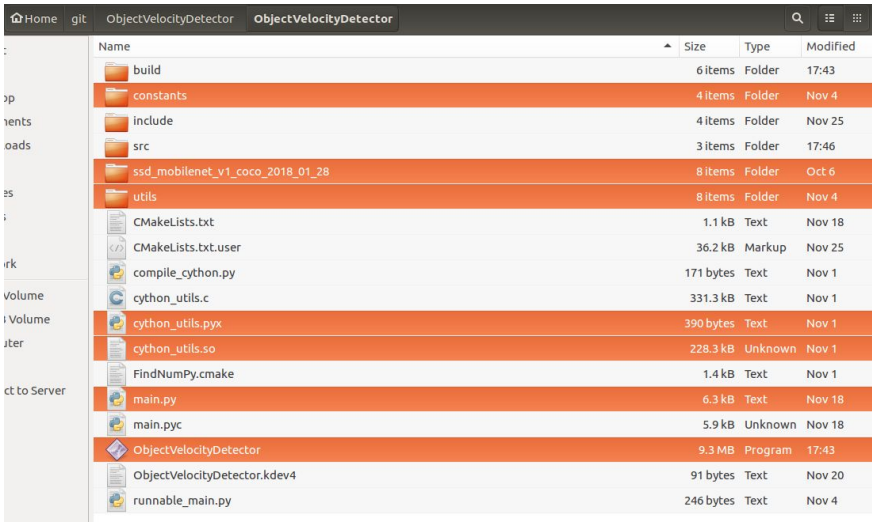
Running the Program

Requirements to run the program

To run the program successfully, a few runtime conditions must be met. First, for Ubuntu 16.04 to effectively retrieve information from a connected PT Grey Blackfly camera, the USB 3.0 driver buffer size must be expanded. A shell script was written to perform this. It can be found in the root source code directory. To run it:

sudo ./expand_usb_core_mem.sh

Additionally, the compiled executable called *ObjectVelocityDetector* MUST be executed in the same directory as the main.py python file. This is because the Python files are manually loaded into a generated Python runtime within the ObjectVelocityDetector process, and the file paths are hard-coded. The expected directory structure is shown below, with the Python dependencies and their necessary locations highlighted.



Finally, a pre-trained TensorFlow object detection model was used as the neural network to generate the bounding boxes and classifications. A pretrained model was used to avoid the necessity to make train a model ourselves. The pretrained model can be downloaded from the Tensor Flow Object Model Zoo on GitHub. The model chosen for this project was a CNN (Convolutional Neural Network) trained on the Common Objects in Context data set because of its high average processing speed. Unzip and untar the COCO trained model into the *<some path>/ObjectVelocityDetector/ObjectVelocityDetector* directory. The program expects the pre-trained TensorFlow model to be in the following location:

~!/<some accessible path after the home directory>/ObjectVelocityDetector/ObjectVelocityDetector/C:\Users\ADA00\Downloads\ssd_mobilenet_v1_coco_2018_01_28/

The object classification label file must also be added to the *ssd_mobilenet_v1_coco_2018_01_28* folder for the program to work. That label can be downloaded from here and placed in the *ssd_mobilenet_v1_coco_2018_01_28* folder.

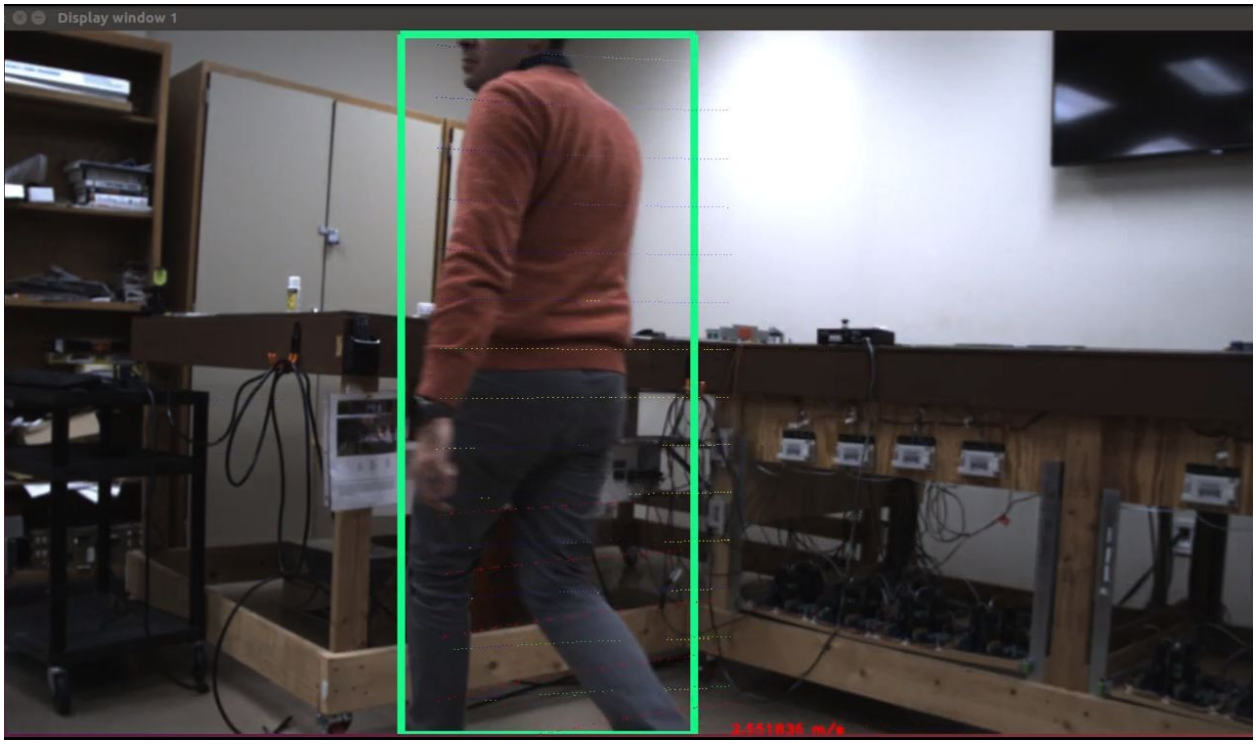
Run the Program

After the run requirements have been met, to run the program, navigate to the project directory and execute the binary in the *ObjectVelocityDetectory/ObjectVelocityDetector/* folder:

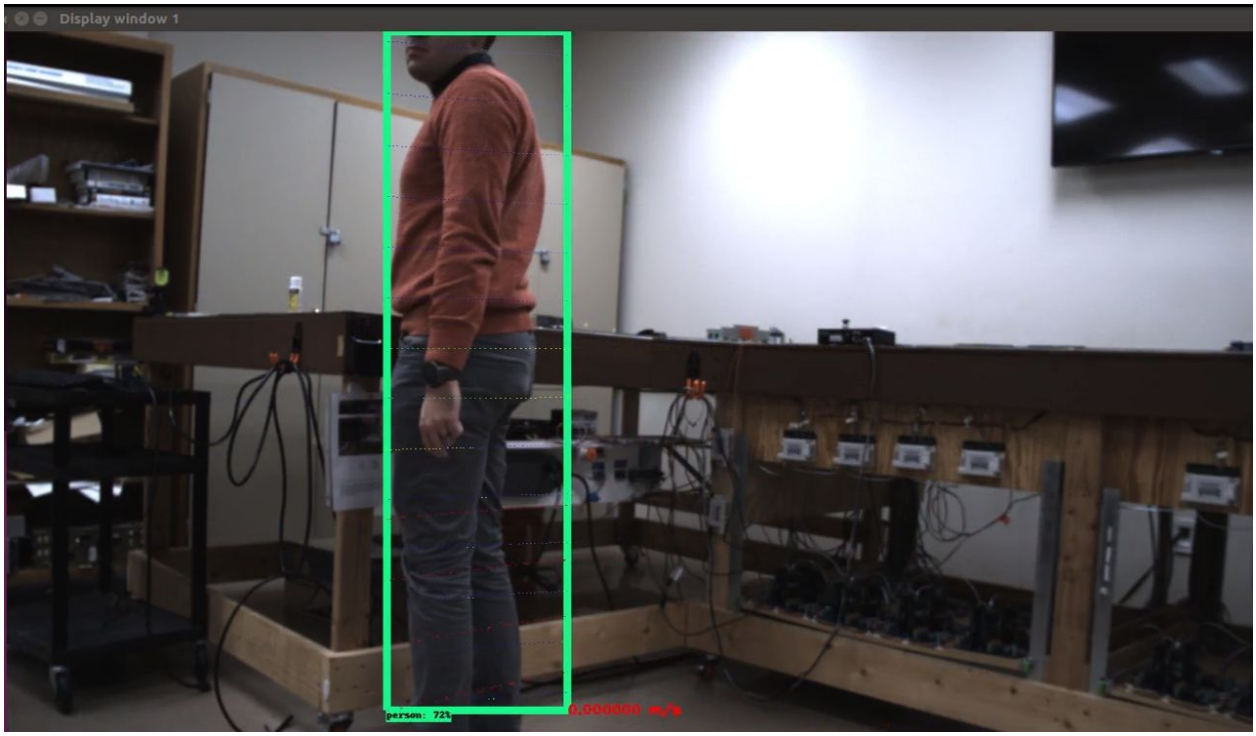
cd <some path in home directory>/ObjectVelocityDetector/ObjectVelocityDetector
./ObjectVelocityDetector

Screenshots

To demonstrate a proof of concept, live video output is provided in a separate window when the program is executed. Properly operating output has the classification label and bounding box surrounding a detected object, with the LIDAR points contained in the bounding box displayed as a colored heat map by proximity.



Person moving 2.55 m/s.



Person standing still

Person moving 0.56 m/s across field of view

Conclusions

Below are the major observations, deficiencies, and potential points of improvement collected from the development of this project.

1) Camera projection / transformations for the right camera
    a) During this project, one thought that was brought up was using the 2 camera setup for fidelity, and we never really got to this point in the software we were developing.
    b) If the future implementers do go with this route, they will also need to apply the correct lidar transformation to the right camera, which should be a similar matrix to what we provided, but a negative in the column that has a 0.3 value.
2) Near-Instantaneous velocity needs to be normalized on two parts
    a) The velocity generated by our software will appear to be very sporadic. This is because object velocity is not “normalized” in any way. There are two things that need to be done with this velocity.
    b) The distance needs to be correctly normalized. There are still going to be points used for the “object location average” calculations that may not actually lie on the object. This can be fixed by doing a normalization with the following steps:
        i) Take the XYZ-distance average of all points within the thinned box.
        ii) Use this average to filter out all points that are further away from this average, which would only keep points that “Should” be on the object.
        iii) Take a 2nd XYZ-distance average and make sure that the points being used fall within a desired standard deviation, discard any that fall outside of this bound.
        iv) These steps should allow a much more accurate distance calculation for the user.
    c) The second thing that will assist in velocity normalization is taking an object's velocity over some number of frames and using the average of those to take the average. This could be done by using a capped std::deque, and taking the average velocity from this structure, rather than displaying the average velocity from frame to frame.
3) More accurate object detection pretrained model
    a) Object detection neural networks, depending on how they’re trained and structured, have two major characteristic trade-offs that must be optimized to fit the needs of the system it is used in: speed and accuracy in detection. A fast object detection inference graph may only pass through each section of the input image once. This could lead to dropped bounding boxes across image frames and fuzzy object edges. If high accuracy in object identification and classification is desired, more intense filtering may be required and result time will suffer. The specific pretrained inference graph chosen was the fastest available in the Tensor Flow model zoo, and as a result, the bounding boxes “jumped” due to

inaccuracy in the detection edges. This lead to noise in the velocity calculation when an object was at rest. This could be mitigated with a slightly more accurate object detection model.

4) More robust object determinator across frames
    a) We are using a very “Brute force” method of tracking an object. It would be much more helpful to have the backing of the neural network to track objects. If a future developer does use more of the python objects to gain further control of data coming from the object detector, make sure to clean up the Python memory properly!! Also our object detection is not using the point cloud in any way to track objects. There are neural nets in production that actually use the PCL data to track objects that are moving around the lidar. These may be something for future developers to look into for this project.
    b) Other things to note is that the tensorflow object detection is very flaky overall, and a more refined detector may provide much better. Keep in mind that there is a horsepower limit to many of these, and this is why we chose this specific tensorflow model, because of the speed. A car moving at 120 mph moves 53 m/s. This is well over 2 meters per frame with some object detection models, and may move even further with slower models.
    c) Occasionally an object will ether vanish or will drop and be made into a new “tracked” object after a second frame. There’s not a lot our current algorithm can do to solve this, and it’s a challenge to further use this, and it may be necessary to look at how this is done to find a way to create more permanence from frame to frame for an object.