National College of Ireland

Configuration Manual

MSc Research Project
MSc Artificial Intelligence

Pragat Pravin Pagariya
Student ID: x23141221

School of Computing
National College of Ireland

Supervisor: Dr. Muslim Jameel Syed

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Pragat Pravin Pagariya……
Student ID:	……x23141221……
Programme:	…Master in Artificial Intelligence…… **Year:** ……2024……
Module:	……
Lecturer:	…Dr. Muslim Jameel Syed……
Submission Due Date:	……12/08/2024……
Project Title:	Lidar – Infused YOLO: A Lidar infused computer vision model to improve the object detection for autonomous vehicle.
Word Count:	……3756…… **Page Count:** ……18……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: ……Pragat Pagariya……

Date: ……11/08/2024……

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Pragat Pravin Pagariya
Student ID: x23141221

System Requirements:
RAM: 32GB
OS: Windows
Processor: Core i9, 13th Generation GPU: 12 GB Nvidia, 3060 RTX
Platform: Online Google Colab Pro Plus

```
!wget https://s3.eu-central-1.amazonaws.com/avg-kitti/raw_data/2011_10_03_drive_0047/2011_10_03_drive_0047_sync.zip
```

Figure1. The command uses the wget to download a compressed Zip file from the specified URL. This archive should contain information related with the KITTI dataset for the drive with the label 2011_10_03_drive_0047.

```
!wget https://s3.eu-central-1.amazonaws.com/avg-kitti/raw_data/2011_10_03_calib.zip
```

Figure2. The command uses wget in which it downloads a ZIP file from the provided URL. Presumably this file should contain calibration data for KITTI dataset used for camera and sensor calibration.

```
!jar xf 2011_10_03_drive_0047_sync.zip
!jar xf 2011_10_03_calib.zip
```

Figure3. The contents of the ZIP files are extracted using the jar tool with the following commands:
The contents of the ZIP files are extracted using the jar tool with the following commands:
!jar xf 2011_10_03_drive_0047_sync. zip: With this command one can retrieve contents of 2011_10_03_drive_0047_sync. zip file.
!jar xf 2011_10_03_calib. zip: This command copies the data of the file 2011_10_03_calib. zip file.

Whereas the jar xf command is normally and primarily used for unarchiving jar files, one can also use it on zip files

```
import os
from glob import glob
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams["figure.figsize"] = (20, 10)
```

Figure4. The code allows for setting up environment for data analysis as well as providing with the libraries to handling files, to process images, to perform numerical computations, to manipulate data, and to create images and plots. Moreover, the additional code for the modification of the plotting configurations can be provided to plot the figures as being 20 inches wide by 10 inches tall by default inside the Jupyter notebooks.

```
!wget https://github.com/itberrios/CV_tracking/raw/main/kitti_tracker/kitti_utils.py
from kitti_utils import *
```

Figure5. The command downloads another Python script by the name of Kitti-utils. It must allow the user to input a URL and retrieve a python script from a GitHub repository. This script can then be imported into the current Python environment and all its functions and utilities can be used. It may be that the script uses specific helper function constructed for utilization in the KITTI dataset.

```
with open('2011_10_03/calib_cam_to_cam.txt','r') as f:
    calib = f.readlines()

# get projection matrices (rectified left camera --> left camera (u,v,z))
P_rect2_cam2 = np.array([float(x) for x in calib[25].strip().split(' ')[1:]]).reshape((3,4))


# get rectified rotation matrices (left camera --> rectified left camera)
R_ref0_rect2 = np.array([float(x) for x in calib[24].strip().split(' ')[1:]]).reshape((3, 3,))

# add (0,0,0) translation and convert to homogeneous coordinates
R_ref0_rect2 = np.insert(R_ref0_rect2, 3, values=[0,0,0], axis=0)
R_ref0_rect2 = np.insert(R_ref0_rect2, 3, values=[0,0,0,1], axis=1)


# get rigid transformation from Camera 0 (ref) to Camera 2
R_2 = np.array([float(x) for x in calib[21].strip().split(' ')[1:]]).reshape((3,3))
t_2 = np.array([float(x) for x in calib[22].strip().split(' ')[1:]]).reshape((3,1))

# get cam0 to cam2 rigid body transformation in homogeneous coordinates
T_ref0_ref2 = np.insert(np.hstack((R_2, t_2)), 3, values=[0,0,0,1], axis=0)
```

Figure6. Record data used for calibration is obtain from the calib_cam_to_cam. txt file, with specific matrices being extracted:txt file, with specific matrices being extracted:

Projection Matrix (P_rect2_cam2): Pulled out from the 26th line of the file this matrix is used to transform the coordinates from the rectified left camera to the image plane of the left camera.

Rectified Rotation Matrix (R_ref0_rect2): This matrix is obtained from the 25th line and is responsible for the transformation from the left camera to rectified left camera hence involves the transformation of image coordinates into robot coordinates. It is then extended homogeneously that is, the technique is extended to homogeneous coordinates.

Rigid Transformation (T_ref0_ref2): The values of the rotation matrix and translation vector are obtained from the lines 22 and 23 to find rigid transformation between Camera 0 and Camera 2. It is also transformed to the homogeneous coordinates as the previous transformation shown in the above equations.

These matrices and transformations are used in computer vision and specifically for calibration and rectification steps of the cameras.

```
# Rigid Transformation
T_velo_ref0 = get_rigid_transformation(r'2011_10_03/calib_velo_to_cam.txt')
T_imu_velo = get_rigid_transformation(r'2011_10_03/calib_imu_to_velo.txt')
```

Figure7. Memory updated

Here's a paraphrased version of the description: Here's a paraphrased version of the description:

The get_rigid_transformation function is employed by the code to read and compute rigid transformation matrices from calibration files: The get_rigid_transformation function is employed by the code to read and compute rigid transformation matrices from calibration files:

T_velo_ref0: This matrix which shows the relationship between Velodyne LiDAR and the reference camera (Camera 0) is obtained from the calib_velo_to_cam. txt file.

T_imu_velo: This matrix showing the change from the IMU (Inertial Measurement Unit) to the Velodyne LiDAR is gotten from the calib_imu_to_velo. txt file.

They are critical when specifying the orientation of the LiDAR and IMU measurements and merging them with the camera framework.

```
# transform from velo (LiDAR) to left color camera (shape 3x4)
T_velo_cam2 = P_rect2_cam2 @ R_ref0_rect2 @ T_ref0_ref2 @ T_velo_ref0

# homogeneous transform from left color camera to velo (LiDAR) (shape: 4x4)
T_cam2_velo = np.linalg.inv(np.insert(T_velo_cam2, 3, values=[0,0,0,1], axis=0))
```

Figure8.
The following transformations are executed by the code:
Transformation from LiDAR to Left Camera (T_velo_cam2): Transformation from LiDAR to Left Camera (T_velo_cam2):
The transformation from the LiDAR sensor to the left color camera is computed by chaining several transformations: The transformation from the LiDAR sensor to the left color camera is computed by chaining several transformations:
The required projection matrix for the transformation from the rectified left camera to the image plane is P_rect2_cam2.
The rotation matrix (R_ref0_rect2) which rectifies the left camera is applied.
The strict transform from Camera 0 to Camera 2, T_ref0_ref2 is also entered.
The transformation (T_velo_ref0) from the Velodyne LiDAR to Camera 0 is also placed in the rigid transformations list.
Hence, a matrix, T_velo_cam2 with a dimensionality of 3x4 results, which gives the velocity of the LiDAR to the left camera.
Homogeneous Transformation from Left Camera to LiDAR (T_cam2_velo): Homogeneous Transformation from Left Camera to LiDAR (T_cam2_velo):
The obtained matrix T_velo_cam2 is then converted to a 4x4 form of the homogeneous transformation matrix and the inverse of the matrix computed.
The last matrix, T_cam2_velo, gives the structure transformation from the left color camera to LiDAR system.
These transformations are used to help in the process of fusing and aligning LiDAR data with camera data for operations such as the camera and LiDAR based 3D object

detection.

```
# transform from IMU to left color camera (shape 3x4)
T_imu_cam2 = T_velo_cam2 @ T_imu_velo

# homogeneous transform from left color camera to IMU (shape: 4x4)
T_cam2_imu = np.linalg.inv(np.insert(T_imu_cam2, 3, values=[0,0,0,1], axis=0))
```

Figure9. Transformations involving the IMU and the left color camera are computed by the code: Transformations involving the IMU and the left color camera are computed by the code:
Transformation from IMU to Left Camera (T_imu_cam2): Transformation from IMU to Left Camera (T_imu_cam2):
The change in velocity from the IMU platform to the LiDAR frame is summed with the change in position from the LiDAR to the left color camera frame.
The end product in the calculations above is T_imu_cam2 and is expressed as a 3 by 4 matrix that gives the transformation from IMU to the coordinate system of the left camera.
Homogeneous Transformation from Left Camera to IMU (T_cam2_imu): Homogeneous Transformation from Left Camera to IMU (T_cam2_imu):
Subsequently, T_imu_cam2 is transformed into a homogeneous transformation matrix with 4 rows and 4 columns, and its inverse.
T_cam2_imu is the transformation matrix that convert from the left color camera coordinate to IMU frame coordinate.
These transitions are used to synchronize and fuse IMU with the camera to promote better analysis on the fused data.

```
!git clone https://github.com/ultralytics/yolov5
```

Figure10. The YOLOv5 repository is pulled from GitHub by this command. By cloning this repository, another equally famous model in the object detection domain, YOLOv5 is available for use, meaning that the codes, pre-trained models and other resources required for training or even for inference are made available.

```
!pip install -r yolov5/requirements.txt
```

Figure11.
The passive voice versions of the sentences are correctly phrased as follows:
1. The dependencies required for Python are now presented in the requirements. Option of yolov5 and other. txt files located in the yolov5 directory are installed by the command.
2. With such dependencies, YOLOv5 is performed, and these involve libraries for machine learning, data mulling, and visualization.

```
import torch

model = torch.hub.load('ultralytics/yolov5', 'yolov5s')
```

Figure12.
The passive voice versions of the sentences are:
1. First, to use the PyTorch library, it is imported and, next, the YOLOv5 small model with a name yolov5s is loaded from the Ultralytics repo using torch library. hub by the code.
2. With this pre-trained model directly, object detection can be conducted.

```
model.conf = 0.25  # confidence threshold (0-1), default: 0.25
model.iou = 0.25  # NMS IoU threshold (0-1), default: 0.45
```

Figure13.

The passive voice versions of the sentences are:

1. Hyperparameters that are used in YOLOv5 architecture are adjusted in the code section.
2. The confidence threshold of detections is set to 0. 25 with model. conf. Jul 17, 2018, predictions with a confidence score of 0. Essentially, the values of 25 or higher are considered in this case.
3. The Non-Maximum Suppression (NMS) condition of the Intersection over Union (IoU) is fixed at 0. 25 with model. Candidates that have an IoU greater than 0. Fig 6 shows the detection results. 25 are omitted to prevent cases that many boxes point to the same object in the image.

```python
def get_uvz_centers(image, velo_uvz, bboxes, draw=True):

    # unpack LiDAR camera coordinates
    u, v, z = velo_uvz

    # get new output
    bboxes_out = np.zeros((bboxes.shape[0], bboxes.shape[1] + 3))
    bboxes_out[:, :bboxes.shape[1]] = bboxes

    # iterate through all detected bounding boxes
    for i, bbox in enumerate(bboxes):
        pt1 = torch.round(bbox[0:2]).to(torch.int).numpy()
        pt2 = torch.round(bbox[2:4]).to(torch.int).numpy()

        # get center location of the object on the image
        obj_x_center = (pt1[1] + pt2[1]) / 2
        obj_y_center = (pt1[0] + pt2[0]) / 2

        # now get the closest LiDAR points to the center
        center_delta = np.abs(np.array((v, u))
                              - np.array([[obj_x_center, obj_y_center]]).T)

        # choose coordinate pair with the smallest L2 norm
        min_loc = np.argmin(np.linalg.norm(center_delta, axis=0))

        # get LiDAR location in image/camera space
        velo_depth = z[min_loc]; # LiDAR depth in camera space
        uvz_location = np.array([u[min_loc], v[min_loc], velo_depth])

        # add velo projections (u, v, z) to bboxes_out
        bboxes_out[i, -3:] = uvz_location

        # draw depth on image at center of each bounding box
        # This is depth as perceived by the camera
        if draw:
            object_center = (np.round(obj_y_center).astype(int),
                             np.round(obj_x_center).astype(int))
            cv2.putText(image,
                        '{0:.2f} m'.format(velo_depth),
                        object_center, # top left
                        cv2.FONT_HERSHEY_SIMPLEX,
                        0.5, # font scale
                        (255, 0, 0), 2, cv2.LINE_AA)

    return bboxes_out
```

Figure14. The function matches the depth data which is acquired by LiDAR with the bounding boxes that are visible in an image.

The centre of the individual bounding boxes is then computed and next LiDAR point nearest to the bounding region center is determined and then the relevant depth information is incorporated to the bounding box.

Said depth information is, if required, superimposed on the image by it.

```
def get_detection_coordinates(image, bin_path, draw_boxes=True, draw_depth=True):

    ## 1. compute detections in the left image
    detections = model(image)

    # draw boxes on image
    if draw_boxes:
        detections.show()

    # get bounding box locations (x1,y1), (x2,y2) Prob, class
    bboxes = detections.xyxy[0].cpu() # remove from GPU

    # get LiDAR points and transform them to image/camera space
    velo_uvz = project_velobin2uvz(bin_path,
                                   T_velo_cam2,
                                   image,
                                   remove_plane=True)

    # get uvz centers for detected objects
    bboxes = get_uvz_centers(image,
                             velo_uvz,
                             bboxes,
                             draw=draw_depth)

    return bboxes, velo_uvz
```

Figure15.

Here's a paraphrased version of the description:

1. The get_detection_coordinates function performs both the object detection process and the LiDAR incorporation process.
2. The objects within the image are predicted with the help of a pre-trained model.
3. Based on that, the bounding boxes may be added as an option on the image.
4. The coordinates of the bounding boxes are obtained, and the point cloud data of LiDAR is transformed into image points.
5. LiDAR depth data is integrated into the boxes and the image may be annotated with these depths with optional methods.

6. The function provides the new depth information of the bounding boxes along with the LiDAR points as the output.

```
!pip install pymap3d
```

Figure16. The command installs the pymap3d which provides a package for transforming from one geographic coordinate system to another, and for calculating different types of geodetic values like the conversion of lat/long coordinate to ECEF coordinates.

```python
import pymap3d as pm


def imu2geodetic(x, y, z, lat0, lon0, alt0, heading0):

    # convert to RAE
    rng = np.sqrt(x**2 + y**2 + z**2)
    az = np.degrees(np.arctan2(y, x)) + np.degrees(heading0)
    el = np.degrees(np.arctan2(np.sqrt(x**2 + y**2), z)) + 90

    # convert to geodetic
    lla = pm.aer2geodetic(az, el, rng, lat0, lon0, alt0)

    # convert to numpy array
    lla = np.vstack((lla[0], lla[1], lla[2])).T

    return lla
```

Figure17. The imu2geodetic function converts the IMU coordinates which are x, y z into geodetic coordinates which are the latitude, longitude and altitude according to a reference point and an angle of heading. Firstly, it transforms the coordinates into range, azimuth and elevation (RAE). After that, it applies the pymap3d package to convert these RAE values into the geodetic coordinates. The last result is returned as a NumPy array.

```python
DATA_PATH=r'2011_10_03/2011_10_03_drive_0047_sync'
#get RGB camera data
left_image_paths = sorted(glob(os.path.join(DATA_PATH, 'image_02/data/*.png')))
right_image_paths = sorted(glob(os.path.join(DATA_PATH, 'image_03/data/.png')))

#get LiDAR data
bin_paths = sorted(glob(os.path.join(DATA_PATH, 'velodyne_points/data/*.bin')))
#get GPS/IMU data
oxts_paths = sorted(glob(os.path.join(DATA_PATH, r'oxts/data**/*.txt')))
print(f"Number of left images: {len(left_image_paths)}")
print(f"Number of right images: {len(right_image_paths)}")
print(f"Number of LIDAR point clouds: {len(bin_paths)}")
print(f"Number of GPS/IMU frames: {len(oxts_paths)}")
```

Figure18. The code configures file paths for different data types within the KITTI dataset and displays the count of available files:

Paths:

left_image_paths: A list having the paths of images taken by the left camera.

right_image_paths: A list including the paths to images taken by the right camera.

bin_paths: A list where each element is a string which is a path to a LiDAR point cloud file.

oxts_paths: A list from which it is possible to identify paths to GPS/IMU data files.

Output:

Shows the counts of left images, right images, LiDAR point clouds, and GPS/IMU frames**.**

```
index = 10

left_image = cv2.cvtColor(cv2.imread(left_image_paths[index]), cv2.COLOR_BGR2RGB)
bin_path = bin_paths[index]
oxts_frame = get_oxts(oxts_paths[index])

# get detections and object centers in uvz
bboxes, velo_uvz = get_detection_coordinates(left_image, bin_path)

# get transformed coordinates of object centers
uvz = bboxes[:, -3:]

# transform to (u,v,z)
# velo_xyz = transform_uvz(uvz, T_cam2_velo) # we can also get LiDAR coordiantes
imu_xyz = transform_uvz(uvz, T_cam2_imu)

# get Lat/Lon on each detected object
lat0 = oxts_frame[0]
lon0 = oxts_frame[1]
alt0 = oxts_frame[2]
heading0 = oxts_frame[5]

lla = imu2geodetic(imu_xyz[:, 0], imu_xyz[:, 1], imu_xyz[:, 2], lat0, lon0, alt0, heading0)
```

Figure19.

The code processes a particular frame from the KITTI dataset as follows:

Data Loading:

The left image is read and digitized into the red, green and blue or the RGB format.

LiDAR data and GPS/IMU data which correspond to detection results are also loaded.

Detection and Transformation:

Thus, object detections which are characterized by their UVZ coordinates (image coordinates, and depth) are obtained.

Information in UVZ coordinates is transformed to IMU coordinates.

Geodetic Conversion:

GPS/IMU data is also used to convert IMU coordinates into Latitudes, Longitudes and Heights.

The outcome is the geodetic locations of the detected objects.

```
velo_image = draw_velo_on_image(velo_uvz, np.zeros_like(left_image))
```

Figure20.

This line of code visualizes LiDAR data on the left image:

draw_velo_on_image (velo_uvz, np. zeros_like(left_image)): The draw_velo_on_image function is used to place LiDAR points represented by the velo_uvz with the same dimension as the blank **i**mage and left_image which has been defined earlier. This process generates an image of the LiDAR's data.

```
# pymap3d seems to mess with matplotlib in colab
%matplotlib inline
plt.rcParams["figure.figsize"] = (20, 10)

# stack image with LiDAR point cloud
stacked = np.vstack((left_image, velo_image))

# display stacked iamge
plt.imshow(stacked);
```

Figure21.

The code performs the following tasks:

Configuration:

Picks settings to enable the creation of plots in a Colab notebook instead of creating a separate window.

Establishes the dimension of figures in plots by default.

Combine Images:

Stamps the left image together with the LiDAR visualization image (velo_image) on top of one another in a vertical manner.

Display:

Utilizes plt. imshow to view the resulted image for visualizing the overlay of LiDAR data on the image**.**

```
left_image_2 = cv2.cvtColor(cv2.imread(left_image_paths[index]), cv2.COLOR_BGR2RGB)
velo_image_2 = draw_velo_on_image(velo_uvz, left_image_2)

plt.imshow(velo_image_2);
```

Figure22.

The code executes the following steps:

Load and Convert Image:

Reads another left image and converts the image to RGB model.

Draw LiDAR Data:

Appends the LiDAR points (velo_uvz) on to the newly loaded left image (left_image_2) with the function draw_velo_on_image.

Display:

Uses plt. imshow to paint out the image with the LiDAR information added on it.

Lidar_Infused_YOLO

```
!wget https://s3.eu-central-1.amazonaws.com/avg-kitti/raw_data/2011_10_03_drive_0047/2011_10_03_drive_0047_sync.zip
```

Figure23. The command also downloads a synchronized data for the KITTI dataset in the form of a zip file through a specified

URL.

```
!wget https://s3.eu-central-1.amazonaws.com/avg-kitti/raw_data/2011_10_03_calib.zip
```

Figure24. The command is a bash command that downloads a ZIP file with calibration data from the KITTI dataset from a URL that is commanded from the bash terminal.

```
!jar xf 2011_10_03_drive_0047_sync.zip
!jar xf 2011_10_03_calib.zip
```

Figure25. Commands implement to extract the data of the 2011_10_03_drive_0047_sync file. zip and 2011_10_03_calib. zip files. Concerning the unzipping of the files, the use of the command jar xf is done, here it assists in the extraction of the contents of the files in the current directory.

```
import os
from glob import glob
import cv2
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams["figure.figsize"] = (20, 10)
```

Figure26.

The code imports key libraries for image processing and visualization:

Imports:

os and glob: Commands for filing and directory management and they include the basic and the advanced commands.

cv2: Used for the enhancement of images to obtain the best results.

numpy: Used in statistical computations.

pandas: Used for data manipulation purposes, but did not feature in this case.

matplotlib. pyplot: Illustration or picture plotting in aligning the images.

Configuration:

Makes sure that plots created by matplotlib are rendered in the same notebook.

Defines the physical size of figures for all plots to be set at 20 inches by 10 inches.

```
!wget https://github.com/itberrios/CV_tracking/raw/main/kitti_tracker/kitti_utils.py
from kitti_utils import *
```

Figure27. The command loads the kitti_utils script which is in the compromise_07 directory. First, this script is loaded from an Internet repository on GitHub using the aiohttp module. ClientSession identified next, and then all functions and classes of it are imported into the current Python environment. It probably has several helper methods intended for data manipulation in the framework of the KITTI dataset.

```
DATA_PATH = r'2011_10_03/2011_10_03_drive_0047_sync'

# get RGB camera data
left_image_paths = sorted(glob(os.path.join(DATA_PATH, 'image_02/data/*.png')))
right_image_paths = sorted(glob(os.path.join(DATA_PATH, 'image_03/data/*.png')))

# get LiDAR data
bin_paths = sorted(glob(os.path.join(DATA_PATH, 'velodyne_points/data/*.bin')))

# get GPS/IMU data
oxts_paths = sorted(glob(os.path.join(DATA_PATH, r'oxts/data**/*.txt')))

print(f"Number of left images: {len(left_image_paths)}")
print(f"Number of right images: {len(right_image_paths)}")
print(f"Number of LiDAR point clouds: {len(bin_paths)}")
print(f"Number of GPS/IMU frames: {len(oxts_paths)}")
```

Figure28.

The code sets up paths for various types of data from the KITTI dataset and counts the available files:

Paths:

left_image_paths: Directories containing grey images from the left camera.

right_image_paths: Directories having RGB images from the right camera.

bin_paths: Directories to which LiDAR point cloud files are saved.

oxts_paths: GPS / IMU data file locations.

Output:

Shows the count of left images, right images, LiDAR point clouds, and GPS/IMU data in the captured frame on KITTI dataset.

```
with open('2011_10_03/calib_cam_to_cam.txt','r') as f:
    calib = f.readlines()

# get projection matrices (rectified left camera --> left camera (u,v,z))
P_rect2_cam2 = np.array([float(x) for x in calib[25].strip().split(' ')[1:]]).reshape((3,4))


# get rectified rotation matrices (left camera --> rectified left camera)
R_ref0_rect2 = np.array([float(x) for x in calib[24].strip().split(' ')[1:]]).reshape((3, 3,))

# add (0,0,0) translation and convert to homogeneous coordinates
R_ref0_rect2 = np.insert(R_ref0_rect2, 3, values=[0,0,0], axis=0)
R_ref0_rect2 = np.insert(R_ref0_rect2, 3, values=[0,0,0,1], axis=1)


# get rigid transformation from Camera 0 (ref) to Camera 2
R_2 = np.array([float(x) for x in calib[21].strip().split(' ')[1:]]).reshape((3,3))
t_2 = np.array([float(x) for x in calib[22].strip().split(' ')[1:]]).reshape((3,1))

# get cam0 to cam2 rigid body transformation in homogeneous coordinates
T_ref0_ref2 = np.insert(np.hstack((R_2, t_2)), 3, values=[0,0,0,1], axis=0)
```

Figure29.

The code reads and processes camera calibration data as follows:

Load Calibration Data:

Performs the line by line reading of the calibration file and put the data into a line list.

Extract and Process Matrices:

Projection Matrix (P_rect2_cam2): Uses and transforms the rectified projection from the calibration data in the required manner based on the set parameters.

Rectified Rotation Matrix (R_ref0_rect2): Runs, scale factor and transforms rotates the rotation matrix into the right coordinate frame.
Rigid Transformation (T_ref0_ref2): Concats the rotation and translation matrices to form the homogeneous transformation matrix for going from Camera 0 to Camera 2.

```
T_velo_ref0 = get_rigid_transformation(r'2011_10_03/calib_velo_to_cam.txt')
T_imu_velo = get_r
igid_transformation(r'2011_10_03/calib_imu_to_velo.txt')
```

Figure30.
The code utilizes the get_rigid_transformation function to retrieve transformation matrices as follows:
T_velo_ref0: The rigid transformation from LiDAR to the reference camera frame by calibrating velo to cam calibration is expressed as calib_velo_to_cam. txt.
T_imu_velo: This computes the rigid transformation matrix from the IMU to the LiDAR given the calibration data from calib_imu_to_velo.
txt.

```
T_velo_cam2 = P_rect2_cam2 @ R_ref0_rect2 @ T_ref0_ref2 @ T_velo_ref0
T_cam2_velo = np.linalg.inv(np.insert(T_velo_cam2, 3, values=[0,0,0,1], axis=0))
```

Figure31.
The code performs calculations and inversions of transformation matrices as follows:
T_velo_cam2: GETS the transform matrix from from LiDAR to left camera, which is calculated using the projection matrix (P_rect2_cam2), the rectification rotation matrix (R_ref0_rect2), the coordinate translation matrix from ref_LIDAR to ref_Camera2 (T_ref0_ref2), and the matrix for transform from velo to ref_Camera1 (T_velo_ref0).
T_cam2_velo: Takes the homogeneous transformation matrix T_velo_cam2 into LiDAR coordinate and also compute its inverse in this
function.

```
T_imu_cam2 = T_velo_cam2 @ T_imu_velo
T_cam2_imu = np.linalg.inv(np.insert(T_imu_cam2, 3, values=[0,0,0,1], axis=0))
```

Figure32.
The code calculates transformation matrices that involve the IMU and the camera as follows:
T_imu_cam2: This matrix defines the transform from the IMU to the left camera and is calculated by multiplying &-Newigy; with &-Veloy;.
T_cam2_imu: Calculates the transpose of the homogeneous transformation matrix T_imu_cam2 defining the transformation from the left camera to the IMU.

```
!git clone https://github.com/ultralytics/yolov5
```

Figure33. The command fetches the yolov5 repository from the GitHub to the local system or environment, which contains the YOLOv5 object detection model alongside its code.

```
!pip install -r yolov5/requirements.txt
```

Figure34. This command will install all the dependencies of Python stated in the requirements file. At the end of the Python script the performance of the model is saved.The configuration file is saved in

yolov5 folder as a txt file with the name val80. These dependencies are important in running and in the utilization of the YOLOv5 object detection model.

```
import torch

model = torch.hub.load('ultralytics/yolov5', 'yolov5s')
```

Figure35. The code begins with the loading of the YOLOv5s model using the PyTorch framework with the help of the torch. hub. load function. It loads the original YOLOv5s (small) neural network model for object detection from the 'ultralytics/yolov5' GitHub repository.

```
# set confidence and IOU thresholds
model.conf = 0.25  # confidence threshold (0-1), default: 0.25
model.iou = 0.25  # NMS IoU threshold (0-1), default: 0.45
```

Figure36.

The code configures thresholds for the YOLOv5 model:

model. conf = 0. 25: This code sets the level of confidence to 0. 25. A confidence score less than this value will lead to a detection being abandoned.

model. iou = 0. 25: Defines the Intersection over Union (IoU) that is to be used on the Non-Maximum Suppression (NMS) algorithm from 0. 25. Bounding boxes that have an IoU greater than this will be refused to avoid cases where multiple boxes are drawn around the same object.

```python
def get_uvz_centers(image, velo_uvz, bboxes, draw=True):
    u, v, z = velo_uvz
    bboxes_out = np.zeros((bboxes.shape[0], bboxes.shape[1] + 3))
    bboxes_out[:, :bboxes.shape[1]] = bboxes

    for i, bbox in enumerate(bboxes):
        pt1 = torch.round(bbox[0:2]).to(torch.int).numpy()
        pt2 = torch.round(bbox[2:4]).to(torch.int).numpy()
        obj_x_center = (pt1[1] + pt2[1]) / 2
        obj_y_center = (pt1[0] + pt2[0]) / 2
        center_delta = np.abs(np.array((v, u))
                              - np.array([[obj_x_center, obj_y_center]]).T)
        min_loc = np.argmin(np.linalg.norm(center_delta, axis=0))
        velo_depth = z[min_loc]
        uvz_location = np.array([u[min_loc], v[min_loc], velo_depth])
        bboxes_out[i, -3:] = uvz_location
        if draw:
            object_center = (np.round(obj_y_center).astype(int),
                             np.round(obj_x_center).astype(int))
            cv2.putText(image,
                        '{0:.2f} m'.format(velo_depth),
                        object_center, # top left
                        cv2.FONT_HERSHEY_SIMPLEX,
                        0.5, # font scale
                        (255, 0, 0), 2, cv2.LINE_AA)

    return bboxes_out
```

Figure37. The get_uvz_centers is the function that generates a dictionary devoted to the association between LiDAR data and detected objects in the image. It calculates the 3D coordinates of (u, v, z) of every detected bounding box with respect to the nearest LiDAR points and can add the depth map on the image.

```python
def get_detection_coordinates(image, bin_path, draw_boxes=True, draw_depth=True):
    detections = model(image)
    if draw_boxes:
        detections.show()
    bboxes = detections.xyxy[0].cpu()
    velo_uvz = project_velobin2uvz(bin_path,
                                   T_velo_cam2,
                                   image,
                                   remove_plane=True)
    bboxes = get_uvz_centers(image,
                             velo_uvz,
                             bboxes,
                             draw=draw_depth)

    return bboxes, velo_uvz
```

Figure38.

The get_detection_coordinates function performs the following tasks:

Detects Objects: To predict the objects in the input image, it employs the YOLOv5 model.
Draws Bounding Boxes: Added the detected bounding boxes on the image if desired.
Processes LiDAR Data: Transforms the LiDAR data from the coordinate system that has been originally used into the image coordinate space.
Associates LiDAR with Detections: Associates LiDAR coordinates with the bounded boxes, it also refers to the image and possibly adds depth to the image.
Returns: A bounding box with information of its depth associated with the point cloud from the LiDAR sensor and its coordinate value.

```
!pip install pymap3d
```

Figure39. The command is pretty simple and installs the pymap3d library, which contains the functions like the conversion of coordinates between domains that include the geodetic coordinates and the local coordinates.

```
import pymap3d as pm


def imu2geodetic(x, y, z, lat0, lon0, alt0, heading0):
    ''' Converts cartesian IMU coordinates to Geodetic based on current
        location. This function works with x,y,z as vectors and lat0, lon0,
        alt0 as scalars.

        - Correct orientation is provided by the heading
        - The Elevation must be corrected for pymap3d (i.e. 180 is 0 elevation)
        Inputs:
            x - IMU x-coodinate (either scaler of (Nx1) array)
            y - IMU y-coodinate (either scaler of (Nx1) array)
            z - IMU z-coodinate (either scaler of (Nx1) array)
            lat0 - initial Latitude in degrees
            lon0 - initial Longitude in degrees
            alt0 - initial Ellipsoidal Altitude in meters
            heading0 - initial heading in radians (0 - East, positive CCW)
        Outputs:
            lla - (Nx3) numpy array of
        '''
    # convert to RAE
    rng = np.sqrt(x**2 + y**2 + z**2)
    az = np.degrees(np.arctan2(y, x)) + np.degrees(heading0)
    el = np.degrees(np.arctan2(np.sqrt(x**2 + y**2), z)) + 90

    # convert to geodetic
    lla = pm.aer2geodetic(az, el, rng, lat0, lon0, alt0)

    # convert to numpy array
    lla = np.vstack((lla[0], lla[1], lla[2])).T

    return lla
```

Figure40. The imu2geodetic function converts Cartesian IMU coordinates into geodetic coordinates due to the use of the pymap3d library. Here's an overview:
Inputs:
x, y, z: Vectors or arrays of values relating to IMU, namely x, y or z direction.
lat0, lon0, alt0: Coordinates for geodetic origin of the system (Latitude longitude and altitude).

heading0: Initial heading in radians which is in between -π and π.
Conversion Steps:
The obtained IMU coordinates are initially translated into Range, Azimuth, Elevation (RAE).
The obtained RAE coordinates are further transformed to the geodetic coordinates by applying the pymap3d
library.

```
index = 10

left_image = cv2.cvtColor(cv2.imread(left_image_paths[index]), cv2.COLOR_BGR2RGB)
bin_path = bin_paths[index]
oxts_frame = get_oxts(oxts_paths[index])
bboxes, velo_uvz = get_detection_coordinates(left_image, bin_path)
uvz = bboxes[:, -3:]
imu_xyz = transform_uvz(uvz, T_cam2_imu)


lat0 = oxts_frame[0]
lon0 = oxts_frame[1]
alt0 = oxts_frame[2]
heading0 = oxts_frame[5]

lla = imu2geodetic(imu_xyz[:, 0], imu_xyz[:, 1], imu_xyz[:, 2], lat0, lon0, alt0, heading0)
```

Figure41.
The code performs the following steps:
Load Image and Data:
Casts or converts the left image to RGB form for processing.
It is used to load LiDAR and GPS/IMU data of the given index using a specific function.
Get Detections:
Uses the Facenet model for getting the facial embeddings and YOLO model for getting the 2D Bounding boxes of objects of the image.
Get LiDAR Points:
Maps the LiDAR points on to the image plane to get the uvz (u, v, z) coordinate.
Transform Coordinates:
Transforms uvz co-ordinates from camera coordinate system to IMU co-ordinate system.
Convert to Geodetic Coordinates:
Finally uses the imu2geodetic function to bring the IMU coordinates into geodetic coordinates that are latitude, longitude, and
altitude.

```
velo_image = draw_velo_on_image(velo_uvz, np.zeros_like(left_image))
```

Figure42. The code velo_image = draw_velo_on_image(velo_uvz, np. zeros_like(left_image)) performs the following actions:The code velo_image = draw_velo_on_image(velo_uvz, np. zeros_like(left_image)) performs the following actions:
Create Blank Image: Creating an image array with all pixels set to zero and dimensions as that of the left_image using numpy. zeros_like(left_image).
Draw LiDAR Points: Takes the draw_velo_on_image() function and applies it to put the LiDAR points (velo_uvz and xz plane) onto this black image as the LiDAR visual against the black background.
What will be obtained is the velo_image where only the LiDAR points will be shown.

```
%matplotlib inline
plt.rcParams["figure.figsize"] = (20, 10)
stacked = np.vstack((left_image, velo_image))
plt.imshow(stacked);
```

Figure43.

The code executes the following steps:

Visualization Setup: Sets Matplotlib to render the plots in the same notebook as the python code is being written in and sets default figure size.

Image Combination: This function vertically combines two images into one image using the NumPy method.

Combined Image Display: Uses Matplotlib in order to display the fused image which will enable the user to compare the original image and the LiDAR data laid on top of the image.

```
left_image_2 = cv2.cvtColor(cv2.imread(left_image_paths[index]), cv2.COLOR_BGR2RGB)
velo_image_2 = draw_velo_on_image(velo_uvz, left_image_2)

plt.imshow(velo_image_2);
```

Figure44.

The code carries out the following tasks:

1.1 Image Processing: Reads an image from a given path and rescales the color format of the image to RGB using OpenCV.

1.2 Data Overlay: Boosts the contrast and brightness of the image and then overlays LiDAR data with the function key.

1.3 Image Display: I use an external library namely, Matplotlib and translate the final image that combines the visual and LiDAR data into it.