IMAGE-BASED FLIGHT CONTROL OF UNMANNED AERIAL VEHICLES (UAVS) FOR MATERIAL HANDLING IN CUSTOM MANUFACTURING

A Thesis

by

YUHAO ZHONG

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

May 2020

Major Subject: Industrial Engineering

ABSTRACT

This study introduces an approach for and the challenges in employing unmanned aerial vehicles (UAVs) for material handling in the emerging industrial custom manufacturing environments. Compared with conventional industrial robotic systems, UAVs offer enhanced flexibility for the design and on-the-fly variation of the pathways and workflow to optimally perform multiple tasks on demand, besides offering favorable cost and dimensional footprint factors. A fundamental challenge to the deployment of UAVs in manufacturing and other indoor industrial settings lies in ensuring the accuracy of a drone's localization and flight path. Earlier approaches based on using multiple sensors (e.g., GPS, IMU) to improve the localization accuracy of UAVs are considered ineffective in indoor environments. In fact, few investigations have tackled the issues arising due to the limited space and complicated components and moving entities, human presence in shop-floor environments. Towards addressing this challenge, a pose estimation method that employs just a single camera onboard with a UAV, together with multiple ArUco markers positioned strategically over the shop-floor is implemented to track the real-time location of a UAV. A Kalman filter is applied to mitigate noise effects for pose estimation. To assess the performance of this method, several experiments were carried out in Texas A&M University's manufacturing labs. The result suggests that Kalman filter can reduce the variance of pose estimation by 88.48% compared to a conventional camera and marker-based motion tracking method (~ 27 cm), and can localize (via averaging) the position to within 8 cm of the actual target location.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Satish Bukkapatnam [advisor] and Professor Yu Ding of the Department of Industrial and Systems Engineering and Professor Srikanth Saripalli of the Department of Mechanical Engineering.

The drone used in the experiment was provided by Professor Srikanth Saripalli. The literature review in Section 1 and 2, along with some preliminary experiments were conducted in part by Aakash Yadav and Ravi Srivatsa of Indian Institute of Technology Tirupati and Aditya Yalamanchili of Indian Institute of Information Technology (Kancheepuram) and Zimo Wang of Department of Industrial and Systems Engineering.

All other work conducted for the thesis was completed by the student independently.

Funding Sources

The advisor's time was supported by Rockwell International Professorship and Texas A&M Xgrants project.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

Recent advances in 3D printing and cyberinfrastructure are enabling a Manufacturing-as-a-Service (MaaS) paradigm, to deliver custom products manufactured on-demand [1]. Manufacturing is becoming more customized than ever as each part is personalized to satisfy the specifications of an individual customer. In fact, 3D printing could potentially restructure and localize manufacturing and offer a Cybermanufacturing kiosks model, that my loosely termed the Kinkos for manufacturing [2]. The authors have recent provided an initial demonstration of Cybermanufacturing kiosks employing laser kirigami process to realize personalized freeform structures with custom and personalized functionalities [3], [4], as well as a smart manufacturing platform for integrated user-machine interaction [5]. In this context, dynamic and highly variable material and workflow pattern pose significant challenges to the deployment of custom manufacturing as a service. It is imperative to take a radically different material handling approach to achieve efficiency as high as those of mass production. Material handling is an indispensable yet often overlooked issue in production, more so in custom manufacturing environments.

Material handling is defined by Material Handling Industry of America (MHI) as the movement, protection, storage and control of materials and products throughout the process of manufacture and distribution, consumption and disposal [6]. Material handling forms a significant portion of the total production cost and is estimated to be around 20-25% of the total direct cost in the United States [7]. This however, depends on the type of production and degree of automation in material handling. Material handling must be performed efficiently, safely, at low cost, in a timely manner, accurately (the right materials in the right quantities to the right locations), and without damage to the materials.

In order to efficiently produce individually stylized and personalized product features, the workflow for successive jobs can be vastly different, and part routing patterns would be complex and highly variable. The shop-floor in custom manufacturing must therefore be endowed with flexibility especially to dealing with variations in the parts or products produced. Material handling system should efficiently perform the functions of (i) random independent movement of work parts between stations, (ii) handling a variety of work part configurations, (iii) convenient access for loading and unloading and (iv) compatibility with computer control for automation. Conventional material handling system employed industrial robots, Automated Guided Vehicles (AGVs) and Automated Storage & Retrieval Systems (AS/RS) to address this imperative. An AS/RS system uses cranes running through the aisles to store or retrieve objects in racks automatically without a significant intervention of a human operator [8].

Material handling in such scenarios is shared between two systems, viz. (i) a primary handling system responsible for moving parts between stations and (ii) a secondary handling system consisting of transfer devices at every workstation. Currently, separate robotic systems are employed for the primary and secondary material handling systems, also requiring the parts to be propositioned and pre-oriented for robotic application. Further, selections of material handling equipment and layout are closely related. The use of immobile and inflexible robotic systems restricts layout configurations, stipulating a robot-centered or station-centered layout. As an alternative AGVs have begun to be deployed, especially to deal with material transport needs in a warehouse environment. However, they impose a strict motion pathway that is suboptimal to the production for a vast majority of the custom products. Moreover, as custom manufacturing as a service paradigm takes off, material handling tasks will become increasingly complicated, which will require higher levels of intelligence and decision-making capabilities of the robots and will also require the 'robust mobility', the capability to move around the work area without relying on rails or moving platforms to execute actions. Thus, current material handling systems, at large, are rendered inflexible to adjust for rapidly changing workflows in custom manufacturing, and hence a smarter alternative material handling system is needed.

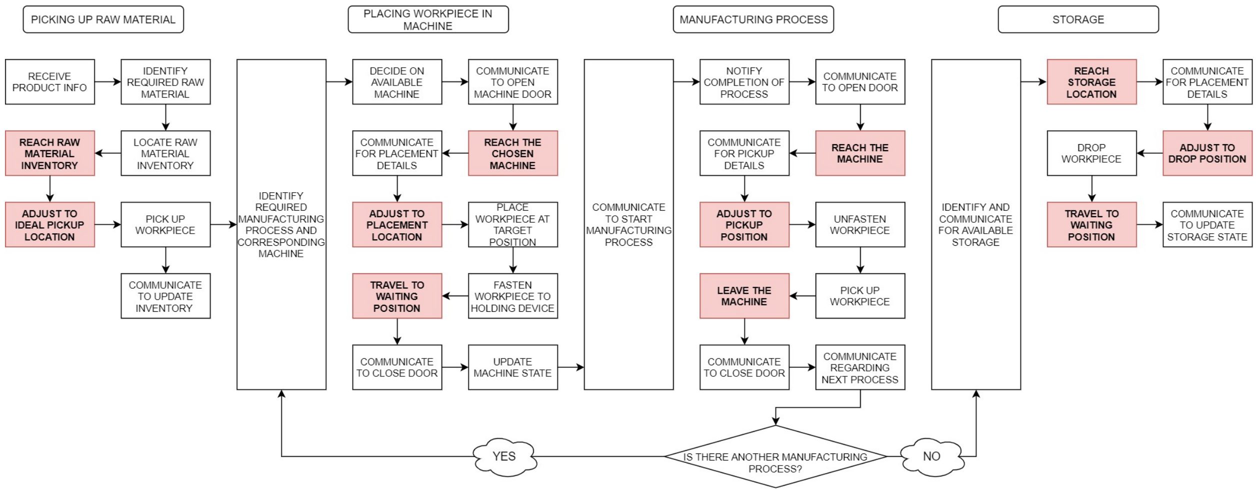
Our study presents UAVs as an alternative and superior material handling tool that can be efficiently employed in manufacturing shop floors. Using UAVs grants the potential to (i) substitute the primary and secondary handling systems with just one, (ii) achieving true variable routing and random order in a custom manufacturing (iii) lenient constraints on process layout/ open field layout, (iv) more efficient part routing, scheduling and dispatching, and (v) scalability and ultimately. Beyond these, manufacturing system layout to-date has aimed to optimize the utilization of the (2-D) floor space; much of the ceiling space in a shop is highly underutilized. A major opportunity exists to fundamentally rethink the way we optimize the layout and material handling and transport paths over the entire 3-D space of the shop-floor. A vast majority of real-world applications for UAVs have been for open environments. As noted in the following sections, adaptation of UAVs for indoor settings, including manufacturing environments is at a very nascent stage. Central to the realization of a UAV material handler for custom manufacturing is to endow the UAVs with pose estimation ability. This is essential to sustain a precise motion pattern, especially in the absence or unviability of location sensors (e.g., GPS and IMUs). We develop an approach based on processing the images gathered from on-board camera of a UAV of the ArUco markers, strategically placed in a shop-floor environment to estimate the location and guide the path of a UAV. We demonstrate and assess the motion accuracy of the UAV based on a case study conducted in Texas A&M Manufacturing Labs. The remainder of the thesis is organized as follows: Section 2 introduces the prior work on UAVs for manufacturing systems; the work plan for UAVs for material handling is presented in Section 3; Section 4 presents the methodology of pose estimation of a UAV in a manufacturing system; Section 5 presents the performance assessment based on the case study, followed by conclusions in Section 6.

2. PRIOR WORK ON UAV FOR MANUFACTURING SYSTEM

Although UAVs have been increasingly considered for real-world applications, such as in surveillance, mapping, inspection, and theater operations, their use in the manufacturing industry, especially the indoor settings have been rather limited. There have been a considerable number of attempts in utilizing the UAVs for outdoor delivery as in the hyperlocal delivery market, i.e. Amazon drone delivery, yet they have not been exploited for material handling in an industrial environment. The potential of UAVs as material handling equipment in custom manufacturing has not been unleashed yet. Most of the real-world UAVs are designed for reconnaissance applications, especially in situations that are unsafe or inaccessible to humans. Load carrying capacity of most of these UAVs is much lower than what is considered typical for industrial material handling applications. Those with the necessary capacity tend to be expensive and large in size. Almost all of the commercially available drones need to be retrofitted with the material carrying apparatus (e.g., a gripper). Currently, the pickup, delivery and placement of materials with UAVs are nascent research topics. Furthermore, when it comes to the safety, as machines in a manufacturing environment are distributed densely and of complicated dimensions, the collision tolerance and resilience of drones should be considered in contrast with the customary obstacle/collision avoidance to guarantee a fail-safe operation [9]. Use of UAVs with vertical take-off and landing (VTOL) is a relatively new research area still in infancy with a potential for various novel applications such as coordinated pickup of heavy items and part assembly.

Deploying UAVs in the industry environment requires a minimal number of fixtures and hardware, as it is possible for a single drone to handle multiple tasks in a sequence. Also, UAVs can be very flexible and efficient in the changing manufacturing workflows because of their dynamic characteristics. Several research efforts are underway to design aerial manipulators and grasping mechanisms for the UAV [10], [11]. This is one of the significant challenges that need to be addressed while designing such a UAV-based material handling system. Heredia et al. [12] and Jimenez-Cano et al. [13] have demonstrated advanced manipulation capabilities and assembly tasks using multiple degrees of freedom manipulators. There have been several instances where UAVs have been utilized in carrying out the construction work, Augugliaro et al. [14] demonstrate the use of UAVs for creating a 3D building structure where multiple UAVs work simultaneously. Research efforts of multi-robot systems have gained momentum in recent times because they have better performance and space utilization. Arbanas et al. [15] provide a decentralized planning and control strategy for a UGV-UAV system. These early efforts clearly point to the potential of employing UAVs for material handling in custom manufacturing environments. Based on these, we present, the following section, a workplan delineating how UAVs will be deployed, what tasks will they perform, how they will carry out these tasks and what are the key challenges in this context.

3. WORKPLAN OF UAV FOR MATERIAL HANDLING



PICKING UP RAW MATERIAL
PLACING WORKPIECE IN MACHINE
MANUFACTURING PROCESS
STORAGE
RECEIVE PRODUCT INFO
IDENTIFY REQUIRED RAW MATERIAL
LOCATE RAW MATERIAL INVENTORY
REACH RAW MATERIAL INVENTORY
ADJUST TO IDEAL PICKUP LOCATION
PICK UP WORKPIECE
COMMUNICATE TO UPDATE INVENTORY
IDENTIFY REQUIRED MANUFACTURING PROCESS AND CORRESPONDING MACHINE
DECIDE ON AVAILABLE MACHINE
COMMUNICATE TO OPEN MACHINE DOOR
REACH THE CHOSEN MACHINE
COMMUNICATE FOR PLACEMENT DETAILS
ADJUST TO PLACEMENT LOCATION
PLACE WORKPIECE AT TARGET POSITION
FASTEN WORKPIECE TO HOLDING DEVICE
TRAVEL TO WAITING POSITION
COMMUNICATE TO CLOSE DOOR
UPDATE MACHINE STATE
COMMUNICATE TO START MANUFACTURING PROCESS
NOTIFY COMPLETION OF PROCESS
COMMUNICATE TO OPEN DOOR
REACH THE MACHINE
COMMUNICATE FOR PICKUP DETAILS
ADJUST TO PICKUP POSITION
UNFASTEN WORKPIECE
PICK UP WORKPIECE
LEAVE THE MACHINE
COMMUNICATE TO CLOSE DOOR
COMMUNICATE REGARDING NEXT PROCESS
IDENTIFY AND COMMUNICATE FOR AVAILABLE STORAGE
REACH STORAGE LOCATION
COMMUNICATE FOR PLACEMENT DETAILS
ADJUST TO DROP POSITION
DROP WORKPIECE
TRAVEL TO WAITING POSITION
COMMUNICATE TO UPDATE STORAGE STATE
IS THERE ANOTHER MANUFACTURING PROCESS?
YES
NO

Figure 3.1: Workplan of UAV for material handling

The workplan of a UAV-based material handling system for custom manufacturing environment can be very complicated as it involves not only interactions among the components (users, machines, inventories and the drone) within the system, but also the various tasks to assure the integrity of operation of an autonomous “drone” (UAV). As summarized in Figure 3.1, we consider the following four broad set of activities as part of the workplan: picking up (an input) material, placing and loading the workpiece into the machine, interacting with the manufacturing process, storage and information systems. We present these activities in the context of a typical custom manufacturing (make-to-order) jobshop as follows:

Picking up material: Upon receiving a manufacturing order, the product information is processed, and the required raw material is identified. Then the UAV receives a notification to fly from its base (e.g., battery charging station) or from its course of flying back to the base (after completing a task) to the location of the corresponding available inventory. After adjusting its position above the material, the drone reached out, picks up the material and then sends a signal to notify the inventory change and proceeds to the destination (a machine or a work-in-process or a storage location). The destination location is determined based on which manufacturing process is required per the process plan, and the location, from among the available cells is communicated to the UAV.

Placing and loading workpiece in the machine: Once the input material is picked up, the drone flies to the machine, communicates to open the machine, and places the workpiece at the required position at the specific machine and process. After the placement, the drone moves to the base or a waiting area (alternate base) and notifies the machine to close the door and start the process.

Manufacturing process: During the process, the drone can be in a standby mode for battery charging. Once the process is finished, it receives a signal indicating completion of the process. It then flies to the machine, communicates to open the machine, and picks up the workpiece. Then it employs a built-in sensor and intelligence to determine the successive process and chooses to move to another machine or storage place accordingly.

Storage: The drone drops the prototype at the target location, that can be either a tote that holds the work-in-process, a (automated) storage system, or another material handler, and sends a signal to notify the storage change. Then it returns to the waiting position. The prototypes will be examined later.

Towards executing even this relatively simple workplan, the UAV needs to be endowed with several capabilities. First, the drone must be able to be programmed to fly autonomously. It must allow the estimation of its real-time location. Towards this end computer vision system with a camera with at least 10 frames-per-second and a high enough (depending on the size and lighting conditions in a shop floor) resolution, with little distortion and preferably image stabilization should be equipped. To accurately move to a specific location or hover at the same place by observing a specified speed and motion profile, the drone should have as little as possible drift, which requires a well calibrated gyroscope (helping balance the drone), good motors, propellers, electronic speed controller (ESCs) and flight controllers. To achieve material handling, the drone must have the ability to carry the load of the input materials and workpieces (by itself or in collaboration with other drones). Also, due to vast variations in the shape and sizes of the custom-products to be made, a versatile robotic arm may be required for holding the material. To assure a reliable operation of the drone for a long enough and continuous time, one should choose the battery capacity according to the workload, while also maintaining a reasonable overall weight. To interact with the machines and computers, the drone is preferred to be wireless. Moreover, if the drone will be working in a complicated environment or even with humans, a collision resilience apparatus should be equipped.

Based on the foregoing, the UAV is desired to have the ability of carrying, grasping, sending and receiving signals, moving accurately and safely. Most fundamental challenge here is to achieve the accurate and autonomous flightpath of the UAV, so that it can pick up, move and drop materials at the planned location by itself (highlighted workflow in Figure 3.1). In this study, as we mainly focus on improving the accuracy of the drone's motion, a Parrot Bebop II drone is employed as the UAV for the case study. This UAV can offer 25-minute flying time and can be wirelessly connected to the computer for controlling and programming. It is equipped with a 1920 x 1080 wide-angle camera with a fisheye lens. The autonomous control programming is based on ROS (Robot Operating System) with Python in Ubuntu OS 16.04. We leverage these capabilities to solve the crucial problem of computer vision-based real-time localization, which can locate the drone within the work space via an array of binary square fiducial markers. This allows the UAV to adjust its motion quickly based on its position, even in a limited space where conventional sensors like GPS may not work.

4. REAL-TIME ESTIMATION OF A UAV LOCATION IN A MANUFACTURING SYSTEM

In a manufacturing environment, facilities such as machines, computers, and inventory racks, etc., are regarded as obstacles to the drone. They are distributed densely and are of complicated shapes, so the motion of the drone should be as accurate (close to the target) and precise (little variation from flight-to-flight) as possible. If we preprogram the motion path (with known distance and velocity, move until total moving time equals the expected time), it would not be accurate since there is an inevitable noise in the drone's motion, such as the drift and fluctuation in the actual velocity. These noise effects cause the drone to move a longer or shorter distance in the expected time period. Instead, we can let the drone control its path itself, by estimating its real-time location, then updating its real-time distance to the destination, and looping until it reaches the target place.

4.1 Camera Calibration

In order to estimate the location of a UAV, a camera-based pose estimation algorithm is implemented. As a first step, the camera on-board the UAV needs to be calibrated to get the pose estimation parameters, such as the intrinsic matrix and distortion coefficients [16]. Normally, the extrinsic matrix would change when the camera moves, so we only make use of the intrinsic matrix and distortion coefficients from calibration. Using all these parameters, we can find the correspondence between a 3D point in the real world and its projection in the 2D image.

Among the different methods available (e.g., [16], [17], [18], [19]) we adapted Zhang's method [20] for calibrating the camera. Zhang's method utilizes a planar pattern of known dimensions, i.e. a chessboard, on which each corner (intersection point of two squares) is a 3D point with known coordinates. This method only requires the camera to observe the pattern at different (at least two) orientations by moving the pattern or the camera. The location change and motion imaging are not needed. Moreover, one pattern can provide multiple correspondences as it has multiple corners. This can be improved further to consider the difference in the focus and issues in the optical setup.

The intrinsic matrix depends only on the camera. It projects the point from the camera-centered coordinate system onto the image plane. Figure 4.1 shows an example of projecting a point (X, Y, Z) regarding the image plane's y-axis. First, we need the focal length f_y expressed in terms of pixels (since the unit in image coordinates is pixel). Since the camera and image plane have different coordinate systems, the offset c_y between their principal points is required as well. The projection regarding the x-axis is similar, with corresponding focal length and offset.

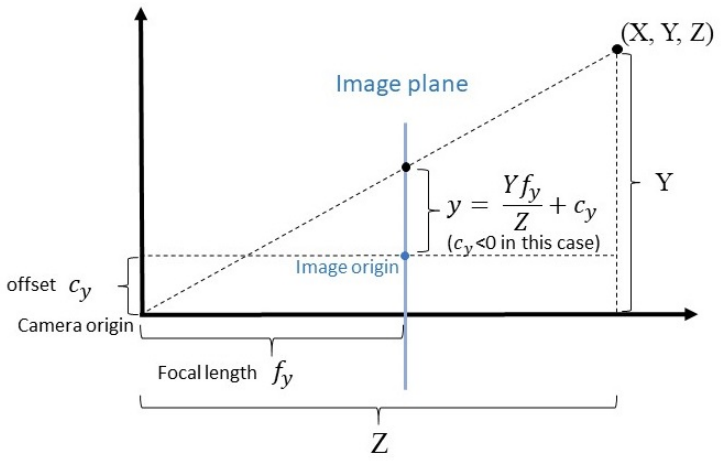
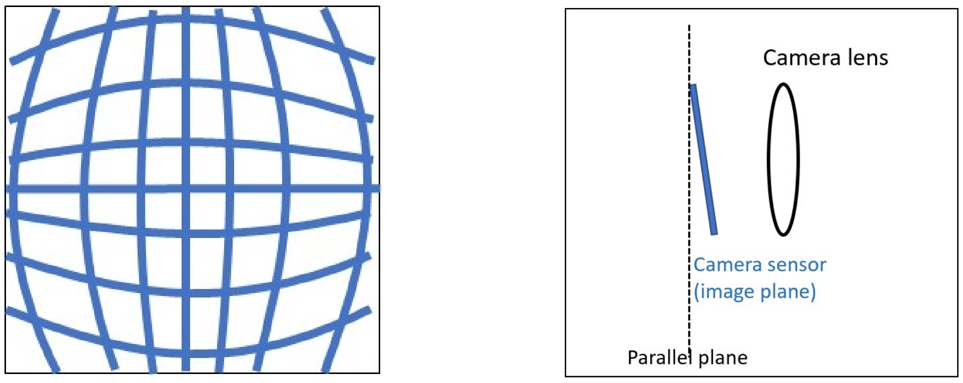


Figure 4.1: Project a 3D point onto the image in terms of Y axis



(a) radial distortion (b) tangential distortion

Figure 4.2: Camera distortion

The distortion coefficients are used to correct the distortion caused by camera lens. In 1966, Brown classified the distortion into radial distortion and tangential distortion [21]. The former one is due to light rays bend more near the edges of a lens than they do at its optical center, while the latter one occurs when the lens and the image plane are not parallel. Examples are shown in Figure 4.2. After obtaining the parameters and clarifying the distortion of the image, we can perform the pose estimation.

4.2 Pose estimation based on ArUco marker system

One of the common pose estimation approaches uses ArUco markers (see Figure 4.3), a type of binary square fiducial markers that are widely used in augmented reality literature. The method is based on the correspondence between the points in a marker-centered coordinate system and their 2D projections in the image. There are two main benefits of using these markers. One is that a single marker can provide enough correspondences (its four corners' known coordinates) to obtain the camera location. Second, different markers can carry non-redundant information due to the robust and unique binary encoding inside each marker [22].

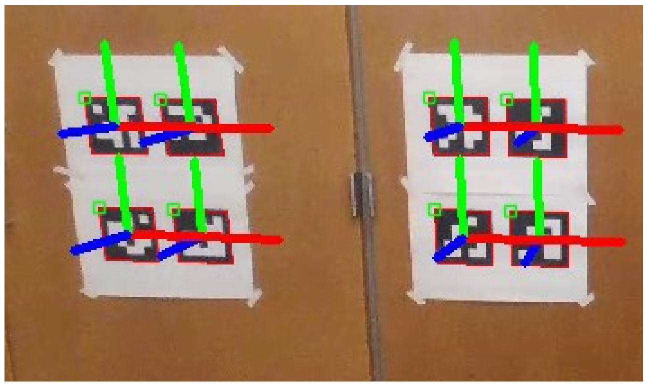


Figure 4.3: An example of markers with their coordinate systems

Based on the intrinsic matrix and distortion coefficients obtained from the camera calibration, once we know the coordinates of corners and their corresponding image coordinates, we can calculate the extrinsic camera matrix. Finally, the coordinates of the camera in the marker space can be derived by multiplying the inverse of the extrinsic matrix and the coordinates of the camera in camera-centered space, which is the origin.

To get the camera position in terms of the real world, we just need to add the coordinates of the origin of the marker system (the center of the marker) in the real-world space.

In this pose estimation method, our inputs are, the coordinates of each marker's center in the real-world space we defined, each corner's coordinates in terms of its marker (calculated from the marker's dimension), images of markers, the intrinsic camera matrix and distortion coefficients. The first two can be measured with little error, while the latter two largely rely on the properties of the camera. Therefore, the variance in this method sometimes can be considerable.

4.3 One-step prediction of UAV location

Traditionally, a drone's motion and pose estimates tend to have dramatically high variances. The variance in motion is mainly due to the drift, while the variance in pose estimation is attributed to the camera's failure get stable images during the flight or its resolution not being high enough. In this thesis, we study the linear motion of the drone, and we assume its speed and location are normally distributed. Based on above, a Kalman filter can be applied to mitigate the error by integrating the information from both the motion and pose estimation, but weighing more on the information that has less uncertainty. Kalman filter is commonly applied for this task, also referred to as Simultaneous Localization and Mapping (SLAM) in the autonomous vehicles literature (e.g., [23], [24], [25], [26]). Specifically, Kalman filter is an iterative data fusion algorithm that includes two steps: prediction and then update. In prediction, the current state is predicted from the previous updated state. Each state contains two components: a state estimate and its error covariance matrix. In update, the current state is updated based on the measurement. Specifically, the predicted means and covariances are updated based on the means and covariances of the latest measurement according to Bayes Rule. Then the updated states are used for prediction in the next iteration.

5. DEMONSTRATION AND PERFORMANCE ASSESSMENT BASED ON A CASE STUDY

A case study is carried out to assess the performance of our method combining the ArUco marker-based pose estimation and Kalman filter. In the experiment (Figure 5.1), the drone moves for one second every time. After each movement, it stops and captures images of markers and sends them to the computer. After processing the images, the computer sends back the location results. The purpose of letting drone stop to collect images is to account for the delay caused by computation and communications. By comparing the estimated location and the target location, the drone flies from raw material inventory to the machine and then to the product inventory.

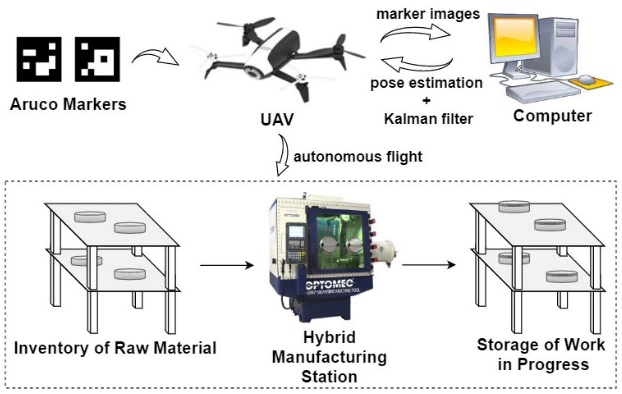

Aruco Markers
UAV
marker images
pose estimation + Kalman filter
Computer
autonomous flight
Inventory of Raw Material
Hybrid Manufacturing Station
Storage of Work in Progress

Figure 5.1: Experiment overview

5.1 Experiment setup

A simulated manufacturing environment is built for assessing the performance of the drone in a real working environment. The planned motion path is shown in Figure 5.2. Specifically, the drone will take off at the raw material inventory, and then fly past the machine, and eventually land at the product storage. The origin of the real-world coordinate system is defined as O and shown in Figure 5.2. The axes X, Y are determined relative to the plan of the markers and they connote the directions in which the drone moves forward/backward and left/right, respectively.

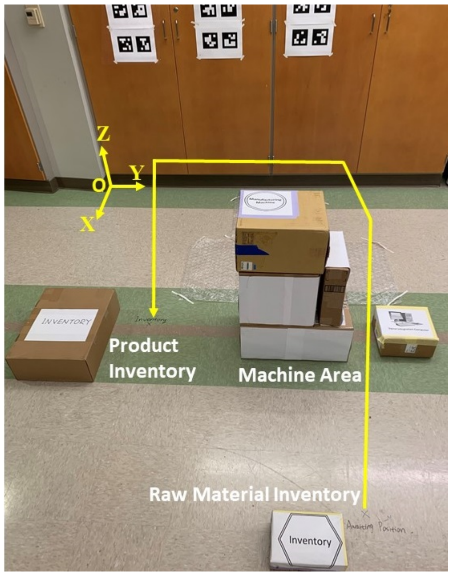


Figure 5.2: Simulated manufacturing environment

5.2 Parameters for the pose estimation and Kalman filter

The camera is calibrated before implementing the pose estimation. Since we are using a Parrot Bebop 2 drone, which has a fisheye lens on the wide-angle camera, we need to get the distortion coefficients of the lens other than intrinsic camera matrix from the calibration. A 6x9 grid chessboard of known dimension (Figure 5.3) is used. During calibration, the camera position is fixed while it captures the chessboard in different orientations and positions. The calibration is conducted 12 times, followed by a pose estimation for a static drone for each time. We choose the calibration result that leads to relatively less error in the static pose estimation. After calibration, the intrinsic matrix and distortion coefficients are obtained.

In pose estimation, a total of 12 markers are used, so we can get 48 correspondences from each image to calculate the coordinates of the camera in terms of the markers. Then by adding the real-world coordinates of the centers of the markers, we get the location of the camera in the real world.

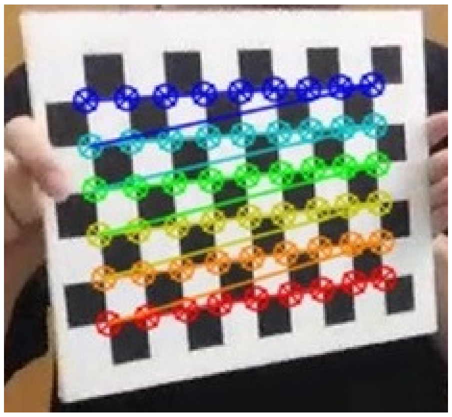


Figure 5.3: Calibration chessboard with a drawn 6x9 grid pattern

In the experiment, the drone moves in orthogonal three axes, namely upward/ downward, left/ right and forward/ backward. We therefore assume that there's no covariance among movements in different directions. The Kalman filter is applied to each direction respectively.

In prediction, the state estimate S_t is a 2x1 vector of the drone's predicted coordinate and velocity on corresponding axis for time t.

$$S_t = \begin{bmatrix} s_t & vs_t \end{bmatrix}^T \tag{5.1}$$

The real-world location after takeoff is used as initial state s_0. As the drone will first move forward, the velocities in other directions are set as 0. But in fact, velocities in other directions are non-zero due to the drift. This error is considered in prediction covariance matrix P_t and thus these velocities can be updated later. P_t can be presented as:

$$P_t = \begin{bmatrix} \sigma_s^2 & \sigma_s \sigma_{vs} \\ \sigma_s \sigma_{vs} & \sigma_{vs}^2 \end{bmatrix} \tag{5.2}$$

In the initial state, each σ is set according to empirical observations, but after a few iterations, they would be updated and become closer to the actual distribution. The prediction is then given by:

$$S_t = AS_{t-1}^n \tag{5.3}$$

$$P_t = AP_{t-1}^n A^T + Q \tag{5.4}$$

Here, the superscript n indicates the variable has been updated, Q is the process noise, which accounts for the error in the prediction model. For instance, we assume the drone can suddenly change speed at the beginning of each iteration and maintain constant within each iteration, while in fact, it should have an acceleration period and may not keep constant afterwards. The transition matrix A,

$$A = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \tag{5.5}$$

has the time between each iteration Δt is set to 1 to simplify computations.

In update, every measurement vector M represents an average of the results of pose estimation based on one frame, namely an average of 48 results since we have 12 markers in each frame. M can be written as:

$$M = \begin{bmatrix} s_t^m & vs_t^m \end{bmatrix}^T \tag{5.6}$$

wherein s_t^m is the x-coordinate of the camera in the real world. vs_t^m is calculated as the first derivative of s_t^m with respect to Δt. Since Δt is 1, vs_t^m can be written as:

$$vs_t^m = s_t^m - s_{t-1}^m \tag{5.7}$$

Kalman Gain K is calculated based on P_t and N, the covariance matrix of the prediction and measurement, respectively.

$$K = P_t H^T (HP_t H^T + N)^{-1} \tag{5.8}$$

N is set according to prior observations and is considered constant because it mainly depends on the fixed configurations of the measuring instrument, namely the camera. H is an identity matrix to format the P_t for matrix operation, and its dimension is determined by the number of variables we are going to update in S_t based on the measurement M. In equation (5.8), if P_t increases or N decreases, which means larger variance in prediction or less variance in measurement, K would be larger. Thus, K can be used as the weight for measurement, while $(1 - K)$ can be used as the weight for prediction. Then S_t can be updated by:

$$S_t^n = (I - KH)S_t + KM \tag{5.9}$$

5.3 Results

The experiments without and with Kalman filter are conducted separately and are repeated multiple times with the same settings (velocity, covariance matrix, takeoff and target location, etc.). Figure 5.4 shows the real-time estimation of the virtual path of the UAV from one of the experiments. In Figure 5.4 (a), each red dot represents an ordinary pose estimation result. The green dots refer to the pose estimates updated with Kalman filter (each green dot is obtained by applying Kalman filter to an average of 48 ordinary results). Figure 5.4 (b) shows the virtual motion paths based on average of the ordinary pose estimation results (red) and results with Kalman filter (green). As we can see, by taking the average of results from 12 markers, the variance of pose estimation is considerably reduced and Kalman filter can further mitigate the noise. The path based on Kalman filter results (the green curve) is visibly smoother and more accurate.

Next, we examined the accuracy of the method in terms of the distance between the point where the UAV is estimated to land for pickup/drop off, and the target point for various experiments. The landing points are shown in Figure 5.5. (The X and Y axes are defined in Figure 5.2). As summarized in the figure, the average distance in the experiments with Kalman filter is 0.08 m, and 0.27 m in the experiments without Kalman filter. The estimated variance of the ArUco marker-based pose estimation is 0.031243 m^2, while that of the one with Kalman filter is 0.0036 m^2. The variance is effectively reduced by approximately 88.48% after applying a Kalman filter.

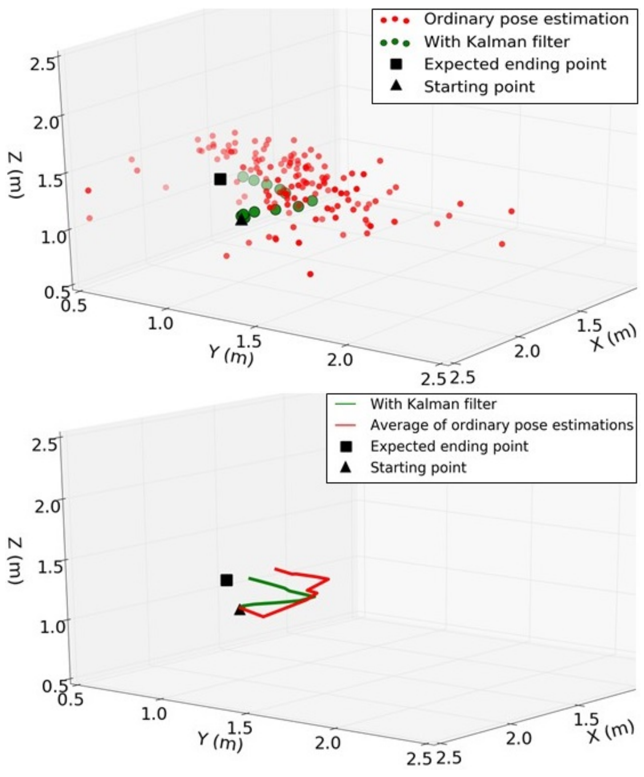


Figure 5.4: (a) Scatter plot of Kalman filter results and results from each marker (b) virtual paths for Kalman filter results and the average results from 12 markers

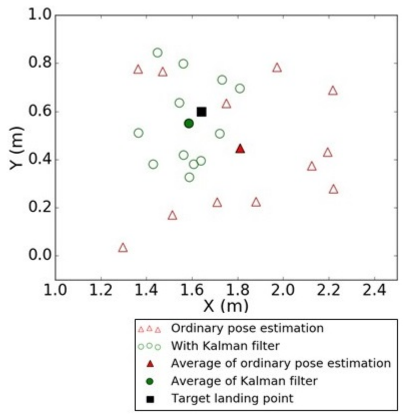


Figure 5.5: Landing locations of all the experiments

6. CONCLUSIONS

In this study, we introduce the concept of using a UAV for material handling in custom manufacturing. Compared with conventional handling methods, UAV is more flexible at dealing with multiple tasks regarding diverse workflows. It also has a lower cost and smaller dimension. Furthermore, we also delineated an elementary four-step workplan for its application. To implement the workplan, many challenges should be identified and solved. One of the most fundamental challenges is to achieve the accurate and autonomous movement of the drone in a complicated indoor manufacturing environment. Towards addressing the challenge, a computer vision-based pose estimation is used to estimate the real time location of the drone. Specifically, Kalman filter is applied to improve the pose estimation accuracy. A case study is carried out to evaluate and compare the methods mentioned above. The result suggests that applying Kalman filter can reduce the variance of pose estimation by 88.48% compared to a conventional ArUco marker-based method, and can localize (via averaging) the position to within 8 cm of the actual target location, which is more accurate than the method without Kalman filter (approximately 27 cm).

However, several other issues remain unsolved regarding the localization of UAV movement. For example, we have not studied the nonlinear process in which the drone can have angular rotation, as well as the dependences among the locations along X, Y, and Z directions. Also, many components of the errors in the localization are largely due to the quality of the camera, and can be addressed by a proper consideration of the camera parameters. Additionally, to comprehensively realize material handling in custom manufacturing, the design of load capacity and collision-tolerant mechanism of the drone should be further studied. These issues are addressed as part of our future research.

REFERENCES

[1] S. Neil, *Manufacturing-as-a-service: Are we there yet? almost.* http://www.apriso.com/blog/2013/06/manufacturing-as-a-service-are-we-there-yet-almost/, 2013.

[2] R. Bilton, *The future of 3d printing looks a lot like kinkos*, https://venturebeat.com/2012/08/16/the-future-of-3d-printing-looks-a-lot-like-kinkos/, 2012.

[3] Z. Wang, A. S. Iquebal, and S. T. Bukkapatnam, "A vision-based monitoring approach for real-time control of laser origami cybermanufacturing processes," *Procedia Manufacturing*, vol. 26, pp. 1307 –1317, 2018, 46th SME North American Manufacturing Research Conference, NAMRC 46, Texas, USA, ISSN: 2351-9789. DOI: https://doi.org/10.1016/j.promfg.2018.07.135. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2351978918308114.

[4] A. S. Iquebal, Z. Wang, W.-H. Ko, Z. Wang, P. Kumar, A. Srinivasa, and S. T. Bukkapatnam, "Towards realizing cybermanufacturing kiosks: Quality assurance challenges and opportunities," *Procedia Manufacturing*, vol. 26, pp. 1296–1306, 2018.

[5] B. Botcha, Z. Wang, S. Rajan, N. Gautam, S. Bukkapatnam, A. Manthanwar, M. Scott, D. Schneider, and P. Korambath, "Implementing the transformation of discrete part manufacturing systems into smart manufacturing platforms," Jun. 2018, V003T02A009. DOI: 10.1115/MSEC2018-6726.

[6] MHI, *Definition of material handling and logistics*, http://www.mhi.org/about, 2019.

[7] R. M. Eastman, *Materials handling*, English. M. Dekker New York, 1987, ISBN: 0824775961.

[8] K. J. Roodbergen and I. F. Vis, “A survey of literature on automated storage and retrieval systems,” *European journal of operational research*, vol. 194, no. 2, pp. 343–362, 2009.

[9] D. Floreano and R. J. Wood, “Science, technology and the future of small autonomous drones,” *Nature*, vol. 521, no. 7553, p. 460, 2015.

[10] P. E. Pounds, D. R. Bersak, and A. M. Dollar, “Practical aerial grasping of unstructured objects,” in *2011 IEEE Conference on Technologies for Practical Robot Applications*, IEEE, 2011, pp. 99–104.

[11] D. Mellinger, M. Shomin, N. Michael, and V. Kumar, “Cooperative grasping and transport using multiple quadrotors,” in *Distributed autonomous robotic systems*, Springer, 2013, pp. 545–558.

[12] G. Heredia, A. Jimenez-Cano, I Sanchez, D. Llorente, V Vega, J Braga, J. Acosta, and A. Ollero, “Control of a multirotor outdoor aerial manipulator,” in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, IEEE, 2014, pp. 3417–3422.

[13] A. Jimenez-Cano, J. Martin, G. Heredia, A. Ollero, and R Cano, “Control of an aerial robot with multi-link arm for assembly tasks,” in *2013 IEEE International Conference on Robotics and Automation*, IEEE, 2013, pp. 4916–4921.

[14] F. Augugliaro, S. Lupashin, M. Hamer, C. Male, M. Hehn, M. W. Mueller, J. S. Willmann, F. Gramazio, M. Kohler, and R. D’Andrea, “The flight assembled architecture installation: Cooperative construction with flying machines,” *IEEE Control Systems Magazine*, vol. 34, no. 4, pp. 46–64, 2014.

[15] B. Arbanas, A. Ivanovic, M. Car, M. Orsag, T. Petrovic, and S. Bogdan, “Decentralized planning and control for uav–ugv cooperative teams,” *Autonomous Robots*, vol. 42, no. 8, pp. 1601–1618, 2018.

[16] R. Tsai, “A versatile camera calibration technique for high-accuracy 3d machine vision metrology using off-the-shelf tv cameras and lenses,” *IEEE Journal on Robotics and Automation*, vol. 3, no. 4, pp. 323–344, 1987.

[17] B. P. Selby, G. Sakas, W.-D. Groch, and U. Stilla, "Patient positioning with x-ray detector self-calibration for image guided therapy," *Australasian physical & engineering sciences in medicine*, vol. 34, no. 3, p. 391, 2011.

[18] T. A. Clarke and J. G. Fryer, "The development of camera calibration methods and models," *The Photogrammetric Record*, vol. 16, no. 91, pp. 51–66, 1998.

[19] Y. Abdel-Aziz, H. Karara, and M. Hauck, "Direct linear transformation from comparator coordinates into object space coordinates in close-range photogrammetry," *Photogrammetric Engineering & Remote Sensing*, vol. 81, no. 2, pp. 103–107, 2015.

[20] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 22, 2000.

[21] D. H. Brown, "Decentering distortion of lenses," 1966.

[22] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez, "Automatic generation and detection of highly reliable fiducial markers under occlusion," *Pattern Recognition*, vol. 47, no. 6, pp. 2280–2292, 2014.

[23] E. Lefferts, F. Markley, and M Shuster, "Kalman filtering for spacecraft attitude estimation," in *20th Aerospace Sciences Meeting*, 1982, p. 70.

[24] S. Chen, "Kalman filter for robot vision: A survey," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 11, pp. 4409–4420, 2011.

[25] S. Rezaei and R. Sengupta, "Kalman filter-based integration of dgps and vehicle sensors for localization," *IEEE Transactions on Control Systems Technology*, vol. 15, no. 6, pp. 1080–1088, 2007.

[26] B. P. Larouche, Z. H. Zhu, and S. A. Meguid, "Development of autonomous robot for space servicing," in *2010 IEEE International Conference on Mechatronics and Automation*, IEEE, 2010, pp. 1558–1562.

Supplemental Sources

[1] M. P. Groover, *Automation, production systems, and computer-integrated manufacturing*. Prentice Hall Press, 2007.

[2] M. P. Groover, M. Weiss, and R. N. Nagel, *Industrial Robotics: Technology, Programming and Application*, 1st. McGraw-Hill Higher Education, 1986, ISBN: 007024989X.

[3] M. P. Groover, *Principles of Modern Manufacturing: Materials, Processes, and Systems*. John Wiley & Sons Inc, 2017, ISBN: 978-1-119-24912-2.

[4] G. Bishop, G. Welch, *et al.*, "An introduction to the kalman filter," 2001.

[5] S. Saripalli, J. F. Montgomery, and G. S. Sukhatme, "Visually guided landing of an unmanned aerial vehicle," *IEEE transactions on robotics and automation*, vol. 19, no. 3, pp. 371–380, 2003.

[6] Wikimedia Foundation, *Camera resectioning*, `https://en.wikipedia.org/wiki/Camera_resectioning`, 2019.

[7] MATLAB & Simulink, *Camera calibrator: What is camera calibration?* `https://www.mathworks.com/help/vision/ug/camera-calibration.html`, 2019.

APPENDIX A

EXPERIMENT RESULTS

Camera calibration results

Intrinsic camera matrix:

$$\begin{bmatrix} 519.40427524 & 0 & 422.37794908 \\ 0 & 506.11336746 & 246.85083267 \\ 0 & 0 & 1 \end{bmatrix}$$

Distortion Coefficient:

$$\begin{bmatrix} 0.00116584 & -0.030782 & 0.00175287 & -0.00248902 & 0.01731515 \end{bmatrix}$$

Pose estimation results

a) Pose estimation results when moving only on one direction (with and without Kalman filter):

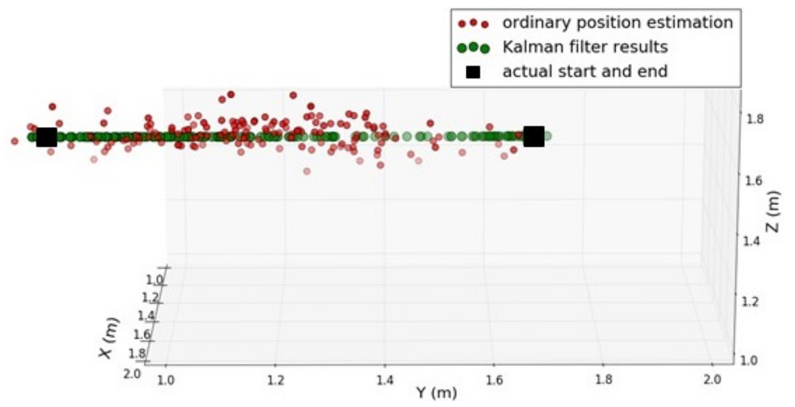


Figure A.1: Pose estimation results when the drone moves only along Y axis

b) Landing location results of ordinary pose estimation (unit:m):

Experiment number	Location Coordinate X	Location Coordinate Y	Euclidean Distance
1	1.88	0.77	0.30
2	1.65	0.41	0.18
3	1.68	0.39	0.21
4	1.59	0.20	0.40
5	1.66	0.59	0.02
6	2.05	0.32	0.49
7	1.60	0.67	0.10
8	1.65	0.75	0.16
9	1.47	0.08	0.54
10	2.02	0.39	0.42
11	1.61	0.65	0.07
12	2.01	0.38	0.42
Average	0.27	Variance	0.031243 m^2

Table A.1: Landing location results of ordinary pose estimation (corresponding to Fig. 5.5)

c) Landing location results of estimation with Kalman filter (unit:m):

Experiment number	Location Coordinate X	Location Coordinate Y	Euclidean Distance
1	1.78	0.65	0.15
2	1.66	0.58	0.03
3	1.45	0.59	0.19
4	1.55	0.75	0.16
5	1.68	0.52	0.10
6	1.63	0.71	0.10
7	1.64	0.56	0.05
8	1.66	0.59	0.03
9	1.73	0.69	0.12
10	1.65	0.58	0.03
11	1.64	0.60	0.01
12	1.66	0.57	0.04
Average	0.08	Variance	0.0036 m^2

Table A.2: Landing location results of estimation with Kalman filter (corresponding to Fig. 5.5)

APPENDIX B

CODES

Camera calibration codes

```
# System information:
# - Linux Mint 18.1 Cinnamon 64-bit
# - Python 2.7 with OpenCV 3.2.0
# Resources:
# - OpenCV-Python tutorial for calibration: http://opencv-python-tutroals.
   readthedocs.io/en/latest/py_tutorials/py_calib3d/py_calibration/
   py_calibration.html
#   - Variable names were changed for clarity

import numpy
import cv2
import pickle
import glob

# Create arrays you'll use to store object points and image points from all
   images processed
objpoints = [] # 3D point in real world space where chess squares are
imgpoints = [] # 2D point in image plane, determined by CV2

# Chessboard variables
CHESSBOARD_CORNERS_ROWCOUNT = 9
CHESSBOARD_CORNERS_COLCOUNT = 6
square_size=0.025

# Theoretical object points for the chessboard we're calibrating against,
# These will come out like:
```

```
#     (0, 0, 0), (1, 0, 0), ...,
#     (CHESSBOARD_CORNERS_ROWCOUNT-1, CHESSBOARD_CORNERS_COLCOUNT-1, 0)
# Note that the Z value for all stays at 0, as this is a printed out 2D image
# And also that the max point is -1 of the max because we’re zero-indexing
# The following line generates all the tuples needed at (0, 0, 0)
objp = numpy.zeros((CHESSBOARD_CORNERS_ROWCOUNT*CHESSBOARD_CORNERS_COLCOUNT,3)
    , numpy.float32)
# The following line fills the tuples just generated with their values (0, 0,
    0), (1, 0, 0), ...
objp[:,:2] = numpy.mgrid[0:CHESSBOARD_CORNERS_ROWCOUNT,0:
    CHESSBOARD_CORNERS_COLCOUNT].T.reshape(-1, 2)
objp*=square_size

# Need a set of images or a video taken with the camera you want to calibrate
# I’m using a set of images taken with the camera with the naming convention:
# ’camera-pic-of-chessboard-<NUMBER>.jpg’
images = glob.glob(’*.jpg’)
# All images used should be the same size, which if taken with the same camera
     shouldn’t be a problem
imageSize = None # Determined at runtime

a=0

# Loop through images glob’ed
for iname in images:
    # Open the image
    img = cv2.imread(iname)
    # Grayscale the image
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Find chessboard in the image, setting PatternSize(2nd arg) to a tuple of
        (#rows, #columns)
    board, corners = cv2.findChessboardCorners(gray, (
       CHESSBOARD_CORNERS_ROWCOUNT,CHESSBOARD_CORNERS_COLCOUNT), None)

    # If a chessboard was found, let’s collect image/corner points
    if board == True:
        a=a+1
        # Add the points in 3D that we just discovered
        objpoints.append(objp)

        # Enhance corner accuracy with cornerSubPix
        corners_acc = cv2.cornerSubPix(
                image=gray,
                corners=corners,
                winSize=(11, 11),
                zeroZone=(-1, -1),
                criteria=(cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,
                   30, 0.001)) # Last parameter is about termination criteria
        imgpoints.append(corners_acc)

        # If our image size is unknown, set it now
        if not imageSize:
            imageSize = gray.shape[::-1]

        # Draw the corners to a new image to show whoever is performing the
           calibration
        # that the board was properly detected
        img = cv2.drawChessboardCorners(img, (CHESSBOARD_CORNERS_ROWCOUNT,
           CHESSBOARD_CORNERS_COLCOUNT), corners_acc, board)
        # Pause to display each image, waiting for key press
        cv2.imwrite(’Chessboard.jpg’, img)

    else:
        print("Not␣able␣to␣detect␣a␣chessboard␣in␣image:␣{}".format(iname))
```

```
# Destroy any open CV windows
cv2.destroyAllWindows()

# Make sure at least one image was found
if len(images) < 1:
    # Calibration failed because there were no images, warn the user
    print("Calibration␣was␣unsuccessful.␣No␣images␣of␣chessboards␣were␣found.␣
       Add␣images␣of␣chessboards␣and␣use␣or␣alter␣the␣naming␣conventions␣used
       ␣in␣this␣file.")
    # Exit for failure
    exit()

# Make sure we were able to calibrate on at least one chessboard by checking
# if we ever determined the image size
if not imageSize:
    # Calibration failed because we didn’t see any chessboards of the
       PatternSize used
    print("Calibration␣was␣unsuccessful.␣We␣couldn’t␣detect␣chessboards␣in␣any
       ␣of␣the␣images␣supplied.␣Try␣changing␣the␣patternSize␣passed␣into␣
       findChessboardCorners(),␣or␣try␣different␣pictures␣of␣chessboards.")
    # Exit for failure
    exit()

# Now that we’ve seen all of our images, perform the camera calibration
# based on the set of points we’ve discovered
calibration, cameraMatrix, distCoeffs, rvecs, tvecs = cv2.calibrateCamera(
        objectPoints=objpoints,
        imagePoints=imgpoints,
        imageSize=imageSize,
        cameraMatrix=None,
        distCoeffs=None)

# Print matrix and distortion coefficient to the console
```

```
print(cameraMatrix)
print(distCoeffs)

# Save values to be used where matrix+dist is required, for instance for
    posture estimation
# I save files in a pickle file, but you can use yaml or whatever works for
    you
f = open('calibration4.pckl', 'wb')
pickle.dump((cameraMatrix, distCoeffs, rvecs, tvecs), f)
f.close()

# Print to console our success
print('Calibration␣successful.␣Calibration␣file␣used:␣{}'.format('calibration4
    .pckl'))
print(a)
```

Pose estimation codes (with Kalman filter)

```
#! /usr/bin/python

# rospy for the subscriber
import rospy
# ROS Image message
from sensor_msgs.msg import Image
# ROS Image message -> OpenCV2 image converter
from cv_bridge import CvBridge, CvBridgeError
# OpenCV2 for saving an image
import cv2
import time
import datetime
import numpy as np
import pandas as pd
import cv2
import cv2.aruco as aruco
import os
import pickle
from geometry_msgs.msg import Twist
from std_msgs.msg import Empty
from numpy.linalg import inv


def restPovec(id):
    if id == 1:
        mkpos = [0.3048,1.2509,0,1]
    #elif id == 3:
        #mkpos = [0.4255,1.2509,0,1]
    #elif id == 5:
        #mkpos = [0.3048,1.0541,0,1]
    #elif id == 7:
        #mkpos = [0.4255,1.0541,0,1]
    elif id == 9:
        mkpos = [0.8303,1.2287,0,1]
    elif id == 11:
        mkpos = [0.9477,1.2287,0,1]
    elif id == 13:
        mkpos = [0.8303,1.0541,0,1]
    elif id == 15:
        mkpos = [0.9477,1.0541,0,1]
    elif id == 25:
        mkpos = [1.3676,1.2287,0,1]
    elif id == 27:
        mkpos = [1.4859,1.2287,0,1]
    elif id == 29:
        mkpos = [1.3676,1.0541,0,1]
    elif id == 31:
        mkpos = [1.4859,1.0541,0,1]
    else:
        mkpos = [0,0,0,0]
    return mkpos

 if not os.path.exists('./calibration4.pckl'):
     print("You␣need␣to␣calibrate␣the␣camera␣you'll␣be␣using.␣See␣calibration␣
        project␣directory␣for␣details.")
     exit()
 else:
     f = open('calibration4.pckl', 'rb')
     (cameraMatrix, distCoeffs, _, _) = pickle.load(f)
     f.close()
     if cameraMatrix is None or distCoeffs is None:
         print("Calibration␣issue.␣Remove␣./calibration.pckl␣and␣recalibrate␣
             your␣camera.")
         exit()

 # Create constant markers
 ARUCO_PARAMETERS = aruco.DetectorParameters_create()
 ARUCO_DICT = aruco.Dictionary_get(aruco.DICT_5X5_1000)

 # Create vectors we'll be using for rotations and translations for postures
 rvecs, tvecs = None, None

 # Instantiate CvBridge
 bridge = CvBridge()



 def pose_estimation(msg):

     global num
     global aland
     i = 0
     camx = []
```

```
camy = []
camz = []

# Convert your ROS Image message to OpenCV2
cv2_img = bridge.imgmsg_to_cv2(msg, "bgr8")
QueryImg = cv2_img
ret = True
if (QueryImg is not None):

            # grayscale image
            gray = cv2.cvtColor(QueryImg, cv2.COLOR_BGR2GRAY)

            # Detect Aruco markers
            corners, ids, rejectedImgPoints = aruco.detectMarkers(gray,
               ARUCO_DICT, parameters=ARUCO_PARAMETERS)

            # Initialize the camera coordinate
            camcord=np.zeros((3,6))

            # Require 8 markers in a photo   i=i+1
            if ids is not None and len(ids) > 0:
                num = num+1

                #Estimate the posture from each Aruco marker
                rvecs, tvecs,_ = aruco.estimatePoseSingleMarkers(corners,
                   0.07, cameraMatrix, distCoeffs)

                for rvec, tvec in zip(rvecs, tvecs):
                    #QueryImg = aruco.drawAxis(QueryImg, cameraMatrix,
                       distCoeffs, rvec, tvec, 0.1)
                    povec=restPovec(ids[i])
                    Rt,_Jacobian=cv2.Rodrigues(rvec)
                    tvec=np.transpose(tvec[0])
                    tvec=tvec.reshape(3,1)
                    temp=np.concatenate(Rt)
                    temp=temp.reshape(3,3)
                    transformation=np.append(temp,tvec,axis=1)
                    append=np.array([0,0,0,1]).reshape(1,4)
                    transformation=np.append(transformation,append,axis=0)
                    transformation=np.linalg.inv(transformation)
                    # Get camera center location in marker coordinate
                       system
                    arucocord=np.matmul(transformation,np.array([0,0,0,1])
                       )
                    # Get camera center location in real world coordinate
                       system
                    cameracord=arucocord+povec
                    cameracord=cameracord[0:3]

                    i=i+1

                    # For display, x is toward the camera, y is along the
                       closet, z is the height
                    camx=np.append(camx,cameracord[2])
                    camy=np.append(camy,cameracord[0])
                    camz=np.append(camz,cameracord[1])
                # Write the photo
                cv2.imwrite(str(num)+".png",QueryImg)     ## should check
                   every time to avoid overwrite
            else:
                print("no␣image␣detected")
                aland = 1
            camcord = [camx,camy,camz,aland]
return camcord
```

```
def land():
    rate = rospy.Rate(10)
    while land_pub.get_num_connections()<1:
        rospy.loginfo_throttle(2,"waiting␣for␣landing")
        rospy.sleep(0.1)
    land_pub.publish(Empty())

def takeoff():
    rate = rospy.Rate(10)
    while takeoff_pub.get_num_connections()<1:
        rospy.loginfo_throttle(2,"waiting␣for␣takeoff")
        rospy.sleep(0.1)
    takeoff_pub.publish(Empty())

# The real world xyz system is consistant with the marker coordinate system
# In the marker system, y is the height, x is along the closet, z is toward
   the camera

def prediction2d(x,vx):
    A = np.array([[1,1],
                  [0,1]])
    X = np.array([[x],
                  [vx]])
    X_prime = A.dot(X)       # Here we don’t consider adding acceleration a
    return X_prime

def covariance2d(sigma_x, sigma_vx):
    sigma = np.array([[sigma_x, sigma_vx]])
    cov_matrix = (sigma.T).dot(sigma)
    return cov_matrix              # let covariance = 0



def main():
```

```
kf_camx = []
kf_camy = []
kf_camz = []
kf_vx = []
kf_vy = []
a_x = []
a_y = []
a_z = []
move_time = []


global land_pub
global aland
global num

camx = []
camy = []
camz = []
num = 0
speedx = 0
speedy = 0
target_x = 1.65 * 100
target_y = 0.8 * 100
target_z = 1.35 * 100

# Initial Estimation Covariance Matrix
# Process / Estimation Errors
error_est_x = 0.01*100
error_est_vx = 0.005*100
error_est_y = 0.05*100
error_est_vy = 0.025*100
error_est_z = 0*100
```

```
error_est_vz = 0*100

# Observation Errors (To avoid S being singular matrix, cannot use the
   same multiply)
error_obs_x = 5 *100 # Uncertainty in the measurement
error_obs_vx = 10 *100
error_obs_y = 0.1 *100
error_obs_vy = 0.2 *100
error_obs_z = 0.2 *100
error_obs_vz = 0.4 *100

A = np.array([[1,1],
              [0,1]])
H = np.identity(2)
Px = covariance2d(error_est_x, error_est_vx)
Rx = covariance2d(error_obs_x,error_obs_vx)
Py = covariance2d(error_est_y, error_est_vy)
Ry = covariance2d(error_obs_y,error_obs_vy)
Pz = covariance2d(error_est_z, error_est_vz)
Rz = covariance2d(error_obs_z,error_obs_vz)

# Get initial location
image_topic = "/bebop/image_raw"
msg = rospy.wait_for_message(image_topic, Image)
camcord = pose_estimation(msg)
camx = camcord[0]
camy = camcord[1]
camz = camcord[2]
aland = camcord[3]
if(aland == 0):
    image_topic = "/bebop/image_raw"
    msg = rospy.wait_for_message(image_topic, Image)
    camcord = pose_estimation(msg)
    camx = np.append(camx,camcord[0])
    camy = np.append(camy,camcord[1])
    camz = np.append(camz,camcord[2])
    aland = camcord[3]
if(aland == 1):
   land_pub = rospy.Publisher("bebop/land",Empty,queue_size=1)
   land()

avg_x = np.mean(camx) * 100
avg_y = np.mean(camy) * 100
avg_z = np.mean(camz) * 100
pos_x = avg_x
pos_y = avg_y
pos_z = avg_z
a_x = np.append(a_x,avg_x/100)
a_y = np.append(a_y,avg_y/100)
a_z = np.append(a_z,avg_z/100)

with open('camx.txt','a') as fz:
        np.savetxt(fz, camx, delimiter=",",fmt='%.4f')
with open('camy.txt','a') as fz:
        np.savetxt(fz, camy, delimiter=",",fmt='%.4f')
with open('camz.txt','a') as fz:
        np.savetxt(fz, camz, delimiter=",",fmt='%.4f')

# initial kalman filter state (x,y,z,vx,vy,vz)
X = np.array([[pos_x],
              [-3]])        # !!!The direction of speed in kalman filter
                 is the reverse of the drone's speed
Y = np.array([[pos_y],
              [0]])
Z = np.array([[pos_z],
              [0]])
```

```
kf_camx = np.append(kf_camx, X[0][0]/100)
kf_camy = np.append(kf_camy, Y[0][0]/100)
kf_camz = np.append(kf_camz, Z[0][0]/100)
kf_vx = np.append(kf_vx, X[1][0]/100)
kf_vy = np.append(kf_vy, Y[1][0]/100)

current_x = X[0,0]      # all elements are x*100 (unit:cm)

#True or False
isMove = input("Sure␣to␣move␣x?:␣")
if(isMove):

  # Move on x direction (towards the marker)
  while  (current_x < target_x-20 or current_x > target_x-6):
                                          ####

     # within a safe x range (just pose estimation result, in case kalman
        filter not good)
     if (pos_x > target_x - 75  and pos_x < target_x + 95 and aland == 0):
       # within a safe y range
       if (pos_y > target_y - 35  and pos_y < target_y + 100 and aland ==
          0):              ####

          prepose_x = pos_x
          prepose_y = pos_y
          prepose_z = pos_z

          # set speed
          if (current_x < target_x-20):
              speedx = -0.01                     #!!!!Notice that the
                 coordinate of world and drone are reversed
              distance = target_x-20-current_x
          if (current_x > target_x-6):
```

```
    speedx = 0.03                       #!!!!Notice that the
        coordinate of world and drone are reversed
    distance = current_x-target_x

# Prediction in Kalman filter
X = prediction2d(X[0][0], X[1][0])
Y = prediction2d(Y[0][0], Y[1][0])
Z = prediction2d(Z[0][0], Z[1][0])
Px = A.dot(Px).dot(A.T)
Py = A.dot(Py).dot(A.T)
Pz = A.dot(Pz).dot(A.T)

# Calculate the Kalman gain
Sx = H.dot(Px).dot(H.T) + Rx +0.0001*H     # H represents the
    random error
Sy = H.dot(Py).dot(H.T) + Ry +0.0001*H
Sz = H.dot(Pz).dot(H.T) + Rz +0.0001*H
Kx = Px.dot(H.T).dot(inv(Sx))
Ky = Py.dot(H.T).dot(inv(Sy))
Kz = Pz.dot(H.T).dot(inv(Sz))

# Move
vel_msg.linear.x = speedx
vel_msg.linear.y = 0
vel_msg.linear.z = 0
vel_msg.angular.x = 0
vel_msg.angular.y = 0
vel_msg.angular.z = 0
# Set the timer
t0 = rospy.Time.now().to_sec()
t1 = rospy.Time.now().to_sec()
while(t1-t0 <= 1):       # either move 2 secs
      #Publish the velocity
      velocity_publisher.publish(vel_msg)
      #Takes actual time to velocity calculus
      t1=rospy.Time.now().to_sec()

# stop the robot
vel_msg.linear.x = 0
velocity_publisher.publish(vel_msg)
t2=rospy.Time.now().to_sec()
move_time = np.append(move_time,t2-t0)
with open('move_time.txt','a') as fx:
     np.savetxt(fx, move_time, delimiter=",",fmt='%.6f')

# Pose estimation (measurement)
image_topic = "/bebop/image_raw"
msg = rospy.wait_for_message(image_topic, Image)
camcord = pose_estimation(msg)
camx = camcord[0]
camy = camcord[1]
camz = camcord[2]
aland = camcord[3]
if(aland == 0):
     image_topic = "/bebop/image_raw"
     msg = rospy.wait_for_message(image_topic, Image)
     camcord = pose_estimation(msg)
     camx = np.append(camx,camcord[0])
     camy = np.append(camy,camcord[1])
     camz = np.append(camz,camcord[2])
     aland = camcord[3]
if(aland == 1):
     land_pub = rospy.Publisher("bebop/land",Empty,queue_size
        =1)
     land()
avg_x = np.mean(camx) * 100
```

```
avg_y = np.mean(camy) * 100
avg_z = np.mean(camz) * 100
pos_x = avg_x
pos_y = avg_y
pos_z = avg_z
a_x = np.append(a_x,avg_x/100)
a_y = np.append(a_y,avg_y/100)
a_z = np.append(a_z,avg_z/100)

with open('camx.txt','a') as fz:
     np.savetxt(fz, camx, delimiter=",",fmt='%.4f')
with open('camy.txt','a') as fz:
     np.savetxt(fz, camy, delimiter=",",fmt='%.4f')
with open('camz.txt','a') as fz:
     np.savetxt(fz, camz, delimiter=",",fmt='%.4f')

# Update in Kalman filter
datax = [pos_x,pos_x-prepose_x]
datay = [pos_y,pos_y-prepose_y]
dataz = [pos_z,pos_z-prepose_z]
Mx = H.dot(datax).reshape(2, -1)
My = H.dot(datay).reshape(2, -1)
Mz = H.dot(dataz).reshape(2, -1)
X = X + Kx.dot(Mx - H.dot(X))
Y = Y + Ky.dot(My - H.dot(Y))
Z = Z + Kz.dot(Mz - H.dot(Z))
Px = (np.identity(len(Kx)) - Kx.dot(H)).dot(Px)
Py = (np.identity(len(Ky)) - Ky.dot(H)).dot(Py)
Pz = (np.identity(len(Kz)) - Kz.dot(H)).dot(Pz)

current_x = X[0,0]
kf_camx = np.append(kf_camx, X[0][0]/100)
kf_camy = np.append(kf_camy, Y[0][0]/100)
```

```
            kf_camz = np.append(kf_camz, Z[0][0]/100)
            kf_vx = np.append(kf_vx, X[1][0]/100)
            kf_vy = np.append(kf_vy, Y[1][0]/100)
        else:
          print("y␣is␣out␣of␣the␣safe␣range")
          current_x = target_x
          Y[0,0] = target_y
    else:
      print("x␣is␣out␣of␣the␣safe␣range")
      current_x = target_x
      Y[0,0] = target_y

print("finish␣moving␣x,␣now␣move␣on␣y")
# initial kalman filter state (x,y,z,vx,vy,vz)
X = np.array([[X[0,0]],
            [0]])        # !!!The direction of speed in kalman filter is
                the reverse of the drone’s speed
Y = np.array([[Y[0,0]],
            [-8]])
Z = np.array([[Z[0,0]],
            [0]])
kf_vx = np.append(kf_vx, X[1][0]/100)
kf_vy = np.append(kf_vy, Y[1][0]/100)
error_est_x = 0.00005*100
error_est_vx = 0*100
error_est_y = 0.001*100
error_est_vy = 0.0005*100
Px = covariance2d(error_est_x, error_est_vx)
Py = covariance2d(error_est_y, error_est_vy)
current_y = Y[0,0]

# Move on y direction
while(current_y < target_y-8 or current_y > target_y+10):
```

```
# within a safe x range (just pose estimation result, in case kalman
   filter not good)
if (X[0,0] > target_x - 75  and X[0,0] < target_x + 45 and aland ==
   0):
  # within a safe y range
  if (pos_y > target_y - 35  and pos_y < target_y + 90 and aland ==
     0):              ####

      prepose_x = pos_x
      prepose_y = pos_y
      prepose_z = pos_z

      # set speed
      if (current_y < target_y-8):
          speedy = -0.01                      #!!!!Notice that the
             coordinate of world and drone are reversed
          distance = target_y-8-current_y
      if (current_y > target_y+10):
          speedy = 0.02                       #!!!!Notice that the
             coordinate of world and drone are reversed
          distance = current_y-target_y-10

      # Prediction in Kalman filter
      X = prediction2d(X[0][0], X[1][0])
      Y = prediction2d(Y[0][0], Y[1][0])
      Z = prediction2d(Z[0][0], Z[1][0])
      Px = A.dot(Px).dot(A.T)
      Py = A.dot(Py).dot(A.T)
      Pz = A.dot(Pz).dot(A.T)

      # Calculate the Kalman gain
      Sx = H.dot(Px).dot(H.T) + Rx +0.0001*H     # H represents the
   random error
Sy = H.dot(Py).dot(H.T) + Ry +0.0001*H
Sz = H.dot(Pz).dot(H.T) + Rz +0.0001*H
Kx = Px.dot(H.T).dot(inv(Sx))
Ky = Py.dot(H.T).dot(inv(Sy))
Kz = Pz.dot(H.T).dot(inv(Sz))

# Move
vel_msg.linear.x = 0
vel_msg.linear.y = speedy
vel_msg.linear.z = 0
vel_msg.angular.x = 0
vel_msg.angular.y = 0
vel_msg.angular.z = 0
# Set the timer
t0 = rospy.Time.now().to_sec()
t1 = rospy.Time.now().to_sec()
while(t1-t0 <= 1):       # either move 2 secs or move to target
      #Publish the velocity
      velocity_publisher.publish(vel_msg)
      #Takes actual time to velocity calculus
      t1=rospy.Time.now().to_sec()

# stop the robot
vel_msg.linear.y = 0
velocity_publisher.publish(vel_msg)
t2=rospy.Time.now().to_sec()
move_time = np.append(move_time,t2-t0)
with open('move_time.txt','a') as fx:
     np.savetxt(fx, move_time, delimiter=",",fmt='%.6f')

# Pose estimation (measurement)
image_topic = "/bebop/image_raw"
```

```
msg = rospy.wait_for_message(image_topic, Image)
camcord = pose_estimation(msg)
camx = camcord[0]
camy = camcord[1]
camz = camcord[2]
aland = camcord[3]
if(aland == 0):
     image_topic = "/bebop/image_raw"
     msg = rospy.wait_for_message(image_topic, Image)
     camcord = pose_estimation(msg)
     camx = np.append(camx,camcord[0])
     camy = np.append(camy,camcord[1])
     camz = np.append(camz,camcord[2])
     aland = camcord[3]
if(aland == 1):
     land_pub = rospy.Publisher("bebop/land",Empty,queue_size
        =1)
     land()

avg_x = np.mean(camx) * 100
avg_y = np.mean(camy) * 100
avg_z = np.mean(camz) * 100
pos_x = avg_x
pos_y = avg_y
pos_z = avg_z
a_x = np.append(a_x,avg_x/100)
a_y = np.append(a_y,avg_y/100)
a_z = np.append(a_z,avg_z/100)

with open('camx.txt','a') as fz:
     np.savetxt(fz, camx, delimiter=",",fmt='%.4f')
with open('camy.txt','a') as fz:
     np.savetxt(fz, camy, delimiter=",",fmt='%.4f')
```

```
            with open('camz.txt','a') as fz:
                 np.savetxt(fz, camz, delimiter=",",fmt='%.4f')

            # Update in Kalman filter
            datax = [pos_x,pos_x-prepose_x]
            datay = [pos_y,pos_y-prepose_y]
            dataz = [pos_z,pos_z-prepose_z]
            Mx = H.dot(datax).reshape(2, -1)
            My = H.dot(datay).reshape(2, -1)
            Mz = H.dot(dataz).reshape(2, -1)
            X = X + Kx.dot(Mx - H.dot(X))
            Y = Y + Ky.dot(My - H.dot(Y))
            Z = Z + Kz.dot(Mz - H.dot(Z))
            Px = (np.identity(len(Kx)) - Kx.dot(H)).dot(Px)
            Py = (np.identity(len(Ky)) - Ky.dot(H)).dot(Py)
            Pz = (np.identity(len(Kz)) - Kz.dot(H)).dot(Pz)

            current_y = Y[0,0]
            kf_camx = np.append(kf_camx, X[0][0]/100)
            kf_camy = np.append(kf_camy, Y[0][0]/100)
            kf_camz = np.append(kf_camz, Z[0][0]/100)
            kf_vx = np.append(kf_vx, X[1][0]/100)
            kf_vy = np.append(kf_vy, Y[1][0]/100)
        else:
          print("y␣is␣out␣of␣the␣safe␣range")
          current_y = target_y
      else:
        print("x␣is␣out␣of␣the␣safe␣range")
        current_y = target_y



with open('kf_camx.txt','a') as fz:
      np.savetxt(fz, kf_camx, delimiter=",",fmt='%.4f')
```

```
    with open('kf_camy.txt','a') as fz:
          np.savetxt(fz, kf_camy, delimiter=",",fmt='%.4f')
    with open('kf_camz.txt','a') as fz:
          np.savetxt(fz, kf_camz, delimiter=",",fmt='%.4f')
    with open('avgx.txt','a') as fx:
          np.savetxt(fx, a_x, delimiter=",",fmt='%.4f')
    with open('avgy.txt','a') as fy:
          np.savetxt(fy, a_y, delimiter=",",fmt='%.4f')
    with open('avgz.txt','a') as fz:
          np.savetxt(fz, a_z, delimiter=",",fmt='%.4f')
    with open('kf_vx.txt','a') as fz:
          np.savetxt(fz, kf_vx, delimiter=",",fmt='%.4f')
    with open('kf_vy.txt','a') as fz:
          np.savetxt(fz, kf_vy, delimiter=",",fmt='%.4f')
    print("finish␣moving")
    time.sleep(1)


if __name__ == '__main__':

    global aland
    aland = 0

    # Starts a new node
    rospy.init_node('parrot_bebop2', anonymous=True)

    # publisher for takeoff
    takeoff_pub = rospy.Publisher("bebop/takeoff",Empty,queue_size=1)
    isTakeoff = input("Sure␣to␣takeoff?:␣")   #True or False
    if (isTakeoff):
       takeoff()
       time.sleep(4)    #3s for takeoff
```

```
# Fly to the planned height
velocity_publisher = rospy.Publisher('/bebop/cmd_vel', Twist,
   queue_size=10)
vel_msg = Twist()
current_height = 0
height = 0.36           #m
speedz = 0.06           #m/s
#isMove = input("Sure to move z?: ")    #True or False
isMove = True
if (isMove):
   vel_msg.linear.x = 0
   vel_msg.linear.y = 0
   vel_msg.linear.z = speedz
   vel_msg.angular.x = 0
   vel_msg.angular.y = 0
   vel_msg.angular.z = 0
   t0 = rospy.Time.now().to_sec()
   while(current_height < height):       #distance can substract a
      certain number to allow drift
          #Publish the velocity
          velocity_publisher.publish(vel_msg)
          #Takes actual time to velocity calculus
          t1=rospy.Time.now().to_sec()
          #Calculates distancePoseStamped
          current_height= speedz*(t1-t0)

# Reached the height, stop
vel_msg.linear.z = 0
velocity_publisher.publish(vel_msg)
time.sleep(1)

# Move and collect photos
main()
```

```
#Land the robot
#landing = input("Ready to land?: ")   #True or False
landing = True
print("start␣landing!")
if (landing and aland == 0):
    landspeed = 0.06
    landdistance = 0.54
    currentz = 0
    vel_msg.linear.z = -landspeed
    t2 = rospy.Time.now().to_sec()
    while(currentz < landdistance):
          velocity_publisher.publish(vel_msg)
          t3=rospy.Time.now().to_sec()
          currentz = landspeed*(t3-t2)
    vel_msg.linear.z = 0
    velocity_publisher.publish(vel_msg)
    # publisher for landing
    land_pub = rospy.Publisher("bebop/land",Empty,queue_size=1)
    land()
else:
    print("already␣landed!")
```