KUKA Cricket Star

Project Final Report
EN.503.707 Robot System Programming
Spring 2020

Jiawen Hu, Kejia Ren, Qihao Liu, Shimin Pan

May 12th, 2020

1 Project Description

1.1 Introduction

In this KUKA Cricket Star project, our team aim to make the KUKA robot arm able to imitate a cricket player and bat a flying ball randomly thrown to it in Gazebo simulation environment. More specifically, the system we design as shown in Fig 1 mainly includes a multiple-camera (4-camera) system which detects and tracks the 3D position of the ball and then predicts the polynomial trajectory of the ball in 3D, and a 7-joint KUKA Lightweight Robt (LWR) holding our self-designed cricket bat to automatically react and hit back the ball thrown towards it.

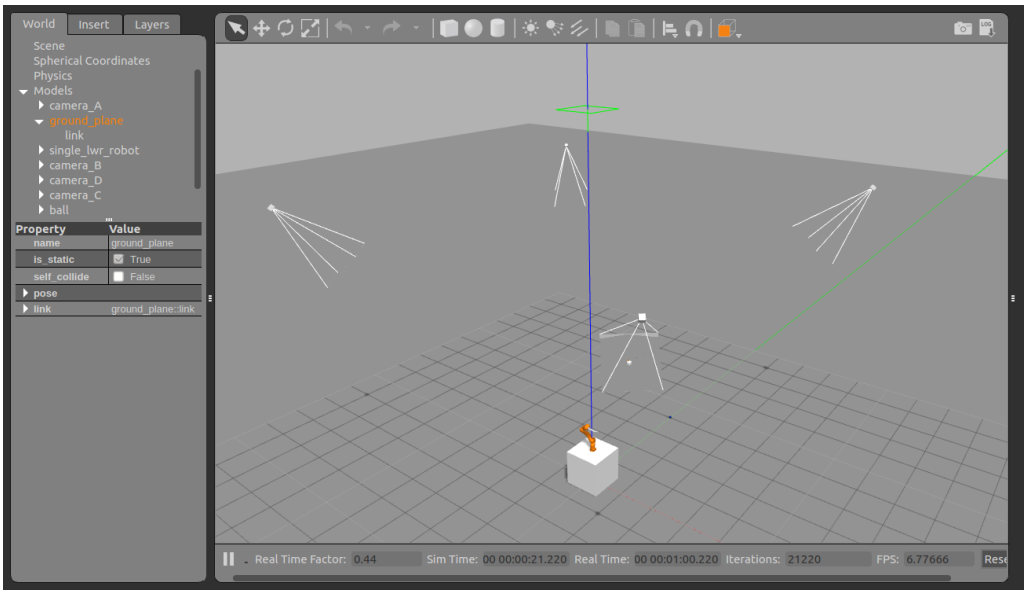


Figure 1: The whole system for our project (simulated in Gazebo).

1.2 Details

The whole pipeline of our project can be broken down into the following steps: 1. We spawn the ball at a specific position in Gazebo and throw it with a properly sampled random initial velocity towards the robot arm; 2. The multi-view system will automatically detect and track the 3D position of the ball; 3. We fit the polynomial curve using the tracked positions to predict the ball trajectory in real time and then we send valid waypoints (ie. waypoints within the reachable region of the robot arm) to guide the planning of robot motion; 4. The robot arm will automatically plan a path to bat the ball back with a specific angle by using both the positions and the velocities of the waypoints. More details about how we make each step work are illustrated below.

1.2.1 Ball throwing

With our self-created SDF blue ball model with bounce property, we can spawn the ball at a specific position in Gazebo. And we throw the ball with a proper but randomly sample initial velocity towards the robot. Figure 2 shows the screenshot for this task.

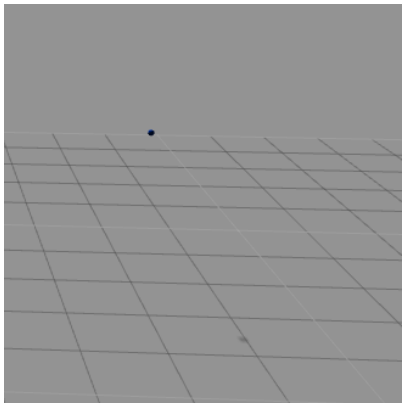


Figure 2: Screenshot for ball throwing in Gazebo.

1.2.2 Multi-view ball tracking

The multi-view system is composed of 4 cameras at different viewpoints. They are all aiming at the central experiment region as shown in Fig 1. The camera calibration process is simulated by calculating the camera projection matrices at the beginning. The ball tracking task mainly has the following three steps:

1) **Ball detection and tracking on 2D image:** This is done for each individual camera. The ball detection on 2D image is done by pixel-level range filtering on HSV colorspace (the result is a filtered binary image) as shown in Fig 3. And then the tracking on 2D image space is by calculating the center of mass for binary images. We check whether the area of each camera binary image is zero, if so, the ball is currently invisible to this camera and we will temporarily discard this image.

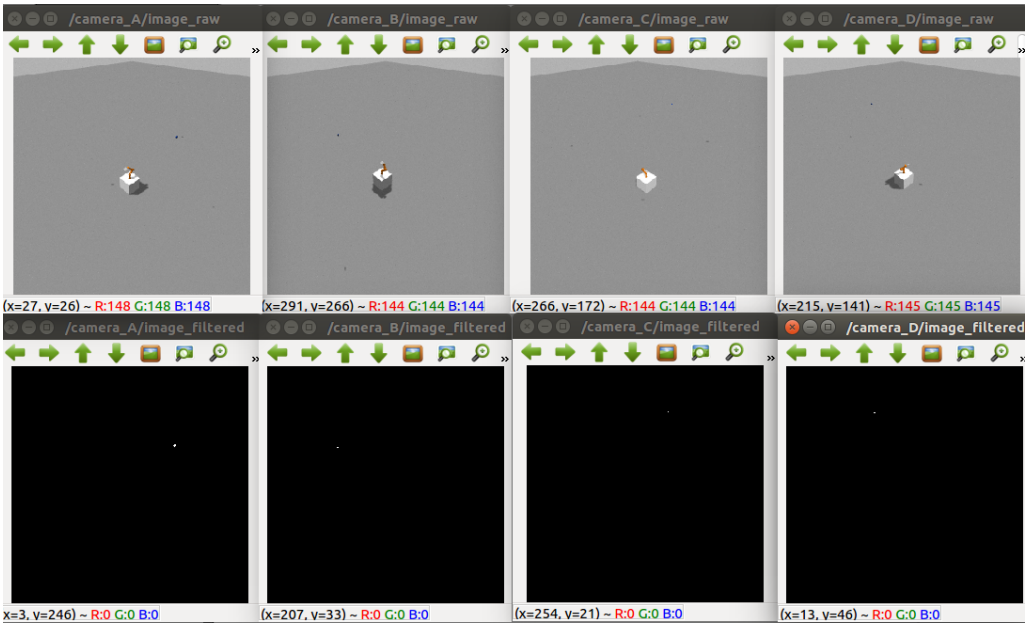


Figure 3: Raw images (first row) and filtered binary images (second row) for the four cameras (left to right: A to D), inspected by running image_view. The cameras are set up high since the ball might fly high, thus the ball is a very small spot in the image. However, if you take a closer look, you can see there are filtered white pixels in the binary images.

2) **Ball tracking in 3D:** The 3D tracking is done by synchronizing the 2D tracking results from all the cameras and doing triangulation. When the ball is currently invisible to less than 2 cameras, the single-view ambiguity will occur and thus we skip the current timestamp. A result is the green points shown in Figure 5.

3) **Kalman Filter smoothing:** The smoothing on 3D tracked results is by a Kalman Filter with a constant acceleration motion model. The state is a 6-dim vector (ball position and velocity in 3D), the observation is a 3-dim vector (3D tracked position from triangulation), and the control in state transition is a scalar (gravity). A result is the yellow points shown in Figure 5. (Note: We do not use the Kalman Filter in our final version of the project since the tracked results are sufficiently accurate, but it is believed this smoothing process would help in real world where the environments are complicated and measurements are noisy.)

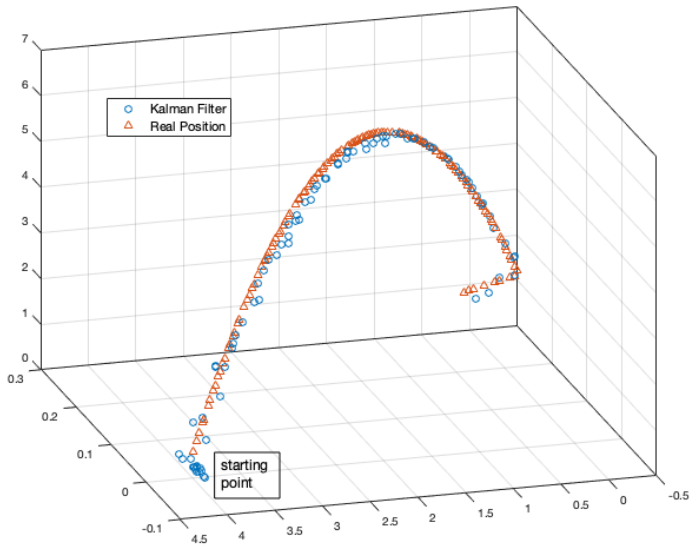


Figure 4: Kalman Filter. The blue ball shows the 3D posterior estimation from the Kalman filter, and the red triangle shows the real position.

1.2.3 Trajectory Prediction with Least Square

We use least square formulation with stacked 3D tracking results to fit second-order polynomial curves in real time (ie. three curves of X, Y, Z ball positions w.r.t. the relative timestamp). The velocity prediction of the ball is also needed since we want to use the velocity information to guide the robot batting angle, and this is done by just computing the derivatives of the polynomials. We set a height threshold, when the ball is under this height, we regard the ball as just re-spawned or just falling onto the ground and bouncing up, and this case will trigger a new polynomial. A result is the red points shown in Figure 5.

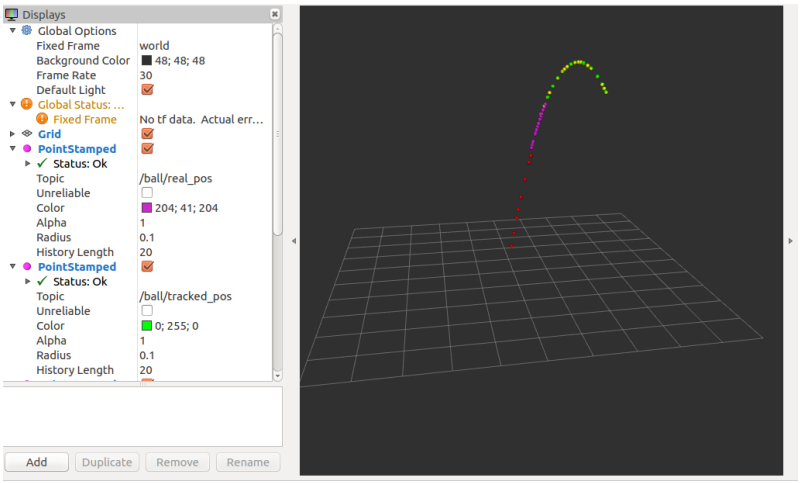


Figure 5: Rviz visualization of the results for multi-view tracking system and trajectory prediction. The purple points are the real ball positions directly obtained from Gazebo, the green points are the tracked 3D position from the multi-view system triangulation, the yellow points are the smoothed results by Kalman Filter, and the red points are the predicted ball position after 0.5 second from our estimated polynomial trajectory. We use this Rviz visualization to validate the performance of tracking and trajectory prediction.

1.2.4 KUKA LWR Control

We formulated and implemented a robot control pipeline for this ball-hitting task: Perception & Control Interface->Trajectory planning->Inverse Kinematics->Ros-control stack. By absorbing the idea of interface class, they are intentionally designed to be as independent and modular as possible. The functionalities and designing ideas about these parts are described as in Figure 6:

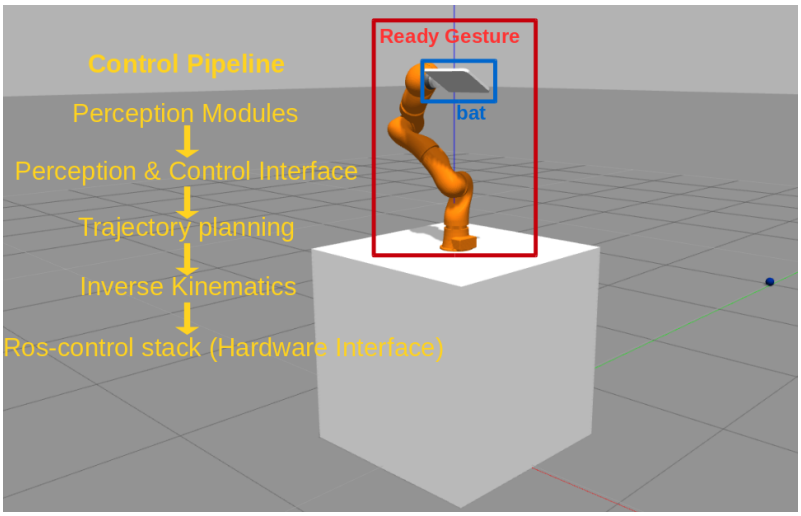


Figure 6: Control pipeline for this ball-hitting task; ready pose before moving; a self-made cricket bat is attached to the KUKA-LWR model.

For Perception & Control Interface, it is responsible for coordinating the perception part and control pipeline, acting as an administrator of the controlling process: start the arm to tracking trajectory, move back to ready gesture, etc.. Besides, it is just like an trajectory tracking module, which follows the inverse predicted trajectory of the ball and "leads" the bat to the predicted position of the ball.

For Trajectory planning, it utilizes the Reflexxes type II library to generate intermediate frames for robot arm. It makes more sense in real robot controlling because movements in physical world

need to be smooth and should comply with specific motor parameters. While in simulation environment, the meaning of this part is more on interpolation in Cartesian space.

For Inverse Kinematics, we basically implemented two ways. The first is using KDL inverse kinematics solver and send joint position to joint trajectory controller. The second is using built-in one-task-inverse-kinematics controller directly (This controller is implented in package lwr_controller).SS

For Ros-control stack, we loaded and started joint_trajectory_controller, joint_trajectory_controller etc. through controller manager for different control strategy. Besides, we configured a running simulation environment for KUKA-LWR arm's velocity controller, which did not exist in existing lwr stack, by registering and configuring velocity hardware interface and using velocity controllers in released ros_control package.

2 Software

Figure 7 shows the rqt_graph for all active nodes and topics when running our project, which illustrates how each part of the software works.

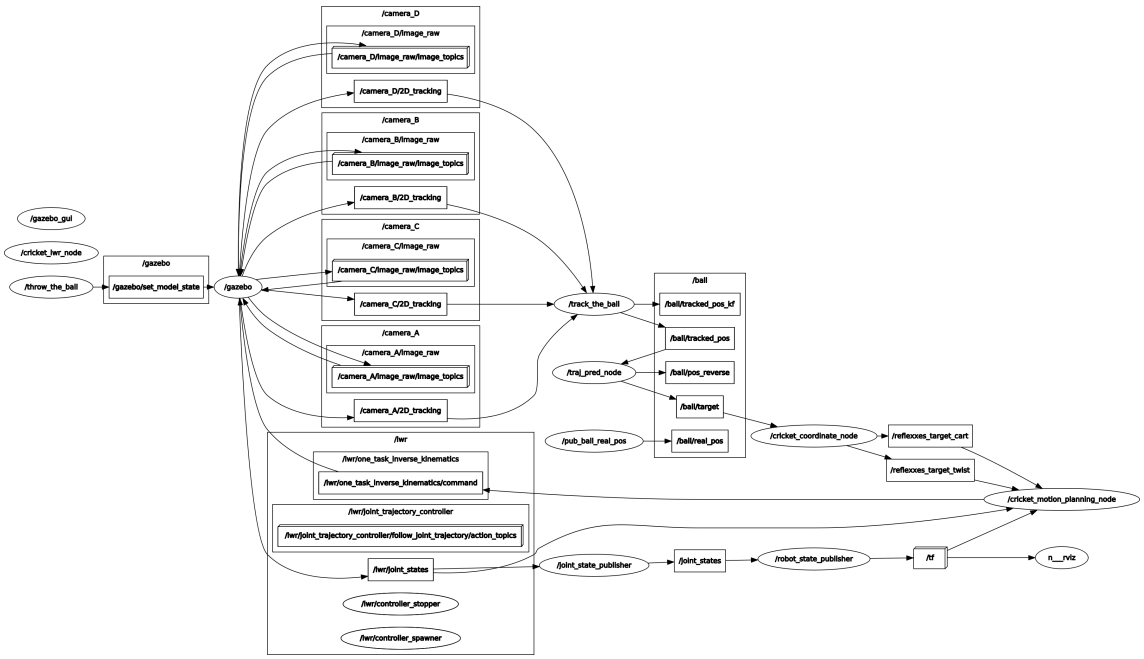


Figure 7: ROS graph for all active nodes and topics when running the project.

2.1 Existing Packages We Utilized

1) Gazebo-related (gazebo_dev, gazebo_msgs, gazebo_plugins): They are used to experiment our project in Gazebo simulation and make the connection between Gazebo and ROS.

2) Robot-related:

- reflexxes_type2: robot arm trajectory planning.
- lwr_controllers: implements lwr controllers based on effort/position interfaces. We **partially modified** it by adding velocity controllers.
- lwr_description: contrains desctiption of lwr model. We **partially modified** it by adding cricket bat model.
- lwr_hw: contains LWR hardware interface configuration. We **partially modified** it in hardware interface.

- single_lwr_launch: keep it for launch files.
- single_lwr_moveit: keep it for lwr's integration.
- single_lwr_robot: keep it for lwr's integration.
- KDL: used to calculate robot kinematics and transformation.

3) Vision-related:

- cv_bridge: It is used to make the conversion between images in OpenCV and image-related messages in ROS.
- OpenCV: It provides us with many useful functionalities helpful to our project (e.g. colorspace conversion, triangulation, least square solver, kalman filter, etc).
- message_filters: It is used to synchronize the multiple camera messages in the multi-view tracking system.

2.2 New ROS Packages We Implemented

All of our self-implemented packages are located in the "src" folder under our project repository on GitLab. You can check it by this link:
https://git-teach.lcsr.jhu.edu/span20/530_707_independent_project.
We want to mention that the "master" branch on Git is the final version of our project. We also create a branch called "inverse" to include all the intermediate work we did.

1) **ball_throwing**: This package mainly provides a ball-throwing node with a SDF ball model. Initial position to spawn, approximate target position and flying duration are passed in by launch file. Based on these parameters, it will create a ball shape target region which is within movable region of robot arm and use this region to constrain the random ball velocity range. The ball is spawned in Gazebo by calling the /gazebo/spawn_sdf_model service inside the node and the initial ball velocity is set by sending a message to /gazebo/set_model_state topic. The real ball position is also accessed from Gazebo through /gazebo/get_model_state topic and it is visualized in Rviz for validating the performance of tracking system.

2) **multi_view_plugin**: This package has an xacro file that includes a single Gazebo camera sensor (low-resolution: 300×300) and our self-implemented Gazebo model plugin "ball_tracker" used to detect and track the ball on 2D images (by range filtering on HSV colorspace and centroid calculation on binary images). This xacro model can be spawned with no gravity in Gazebo by specifying the pose. Besides, in the package, we also implement a class "multi_view_sys" (run in a ROS node) that subscribes and synchronizes (by message_filters) the 2D tracking result messages from multiple cameras, and imitates the calibration process for each camera through calculating the projection matrices by accessing the camera poses from Gazebo, and does triangulation to get 3D tracked position of the ball followed by Kalman filter smoothing. Those tracked or smoothed 3D results are published to specific type of topics that can be visualized in Rviz.

3) **traj_pred**: This package will run a node with our self-implemented trajectory predictor that subscribes the 3D tracking results from the multi-view system and use these stacked results to fit second-order polynomial curves by least square for predicting the 3D ball trajectory. It then reversely sends some valid waypoints on the trajectory to guid the motion planning of robot arm. The 3D results are published in specific topic type for visualization in Rviz.

4) **ekf**: This is a package using Kalman Filter to make estimation of the flight of the ball and his position. It receives the position of the 3D tracked position (observation) from multi_view_plugin, then it compute the posterior estimation. Due to the relatively low precision compared with the Kalman Filter provided by OpenCV and the high accuracy of the tracked result, we do not use this implemented package in our final version. (This package is not included in the final version, ie. the "master" branch of our project. We put it into the "inverse" branch.)

5) **cricket_coordinate**: This package is an interface between perception and robot control. It reads messages from perception-related packages and gets the estimated and predicted positions of the ball. Motion commands will be formulated according to this information and put into robot arm motion control pipeline sequentially.

6) **cricket_motion_planning**: This package is a trajectory planning package for LWR. Utilizing Reflexxes type II library, it generates target frame in Cartesian space in each control step. This package receives final target poses from cricket_coordinate, and breaks them into multiple intermediate frames, which has smooth position and velocity trajectory, sending them to the lower-level package in the control stack (i.e. cricket_lwr_interface or the built-in controller one_task_inverse_kinematics).

7) **cricket_lwr_interface**: This is a package calculating inverse kinematics for lwr. It receives Cartesian poses from upper-level packages, calculate corresponding position in joint space and send it to joint trajectory controller. It has a counterpart as one_task_inverse_kinematics, which is a built-in controller in an existing KUKA-related package.

8) **cricket_star**: This package is the master entrance of whole project. It ensembles all packages into one launch file and we could set several critical parameters for test. To run the whole project, you need to launch in this package.

3 Hardware and infrastructure

Due to the situation that we are not currently working in campus, we are not accessible to any real robot and sensors, so the whole project is done in the Gazebo simulation environment, and Rviz is used for visualizing some intermediate results.

4 Contributions

Each member has been actively participating in the project meetings and discussions, and has intellectual contributions to making our final project work as expected.

1) Jiawen Hu:
Principal contribution: Mainly I focused on ensemble of each part into cricket_star package and performance test after each minor ensemble. During test, I focused on minor error solution and code modification like camera setting, ball-throwing target, ball bounciness and parameters. For final test, I focused on ball hitting trajectory debugging and related parameters modification.

Lessons learned: I learned how each part could work separately, how to combine them together and understand huge amount of tricks when developing a package. For each part, I deeply understand why start-up.launch and upload.launch are all necessary. Upload.launch could be imported by its own package and also outside package. start-up.launch is helpful when we need test of one package. This trick strongly help project uncoupling and scalability and make our project work. What's more, i deeply understand how we should arrange and pass parameters properly. A complete project could have large amount of parameters. A proper arrangement could make development and test much easier and more efficient.

2) Kejia Ren:
Principal contribution: I am responsible for developing, testing and maintaining the ball tracker plugin, the multi-view system (projection matrix calculation, camera synchronizing, 2D/3D tracking, smoothing, etc) and the trajectory prediction part. I also help with the ball-throwing part implementation and experiments.

Lessons learned: By learning from lectures and doing this project, I get the opportunities to get exposed to many very useful resources and documentations. I see many powerful tools like simulation tools and problem-solving libraries that could help with many kinds of project, and this is very valuable to me. In the future, especially in a practical project, I might will encounter many situations where I need to find efficient tools to use. As I believe, knowing how to access them and use them as much as possible would be a great benefit. Thus I am also very thankful to this course and project. As for collaboration, I see how important it is. Good collaboration and effective discussion is delightful and is seen to be required for developing and maintaining a project in limited time, I am thankful to my partners and will be always seeking for good collaboration opportunities.

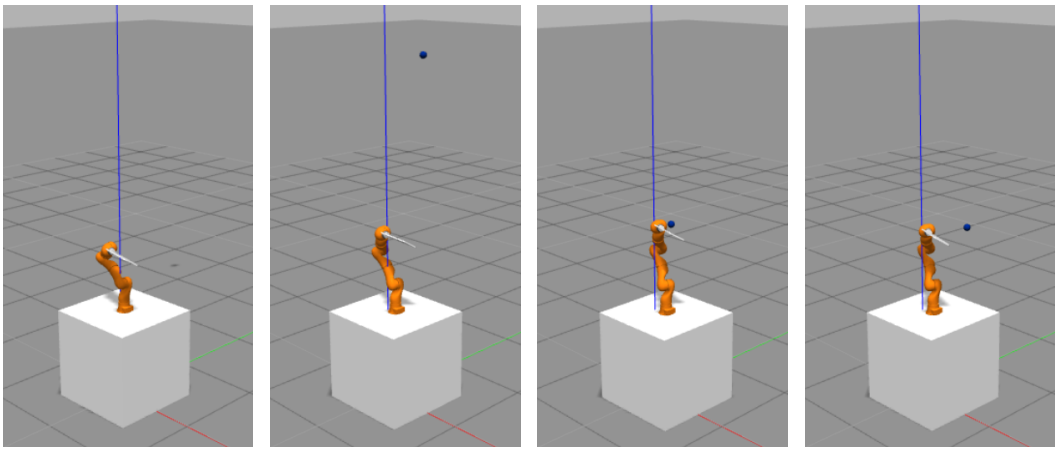
3) Qihao Liu:
Principal contribution: In this project, my main contributions include implementing the Ball Throwing part, the 3D position measurement (using kalman filter), and the trajectory prediction part. Also, I participate in the test sessions to make the whole system works better.
Wish to known: I think it will better if we have the chance to play with different kind of robot arms and more movable robots. It may help us known better about the robot we will working with, give us more background to select a topic, and most importantly, it can help us know better about the most likely outcome and the main difficulty of the project. At the beginning of this project, in consideration of the relatively small size and high speed of the moving ball, I thought the key of our project was to make accurate location and prediction of the ball. However, it turns out I am wrong, and control is the most difficult and troublesome part.
Lessons learned: This class is the most helpful course I took this semester. I was a beginneer to ROS and after taking this class, I know a lot about the ROS. I learnt the basic idea of ROS and also being able to play with different packages and toolkits. As for the future projects, I get to known a lot of powerful toolkits that are well-implemented and easy to use for solving different kind of problems. I also get more familiar with machine vision and GIT tools though the final project.

4) Shimin Pan:
Principal contribution:: I focused on KUKA-LWR arm controlling part in this project. That mainly includes an interface package between perception and controlling, trajectory planning package and inverse kinematics package. I also modified the original existing lwr-related packages to support this specific cricket-hitting task and finally got deeply involved into the overall integration and testing process. I also configured a velocity controller for LWR, which can potentially be use in inverse velocity kinematcs control in the future.
Lessons learned: I've learned a vivid and specific lesson on how to control a off-the-shelf robot arm, especially in Gazebo simulation environment. I've learned and practiced developing ros-control stack from top to bottom (i.e. from trajectory planning to hardware interface), I also learned from my partner's code on how to implement vision-based system in ROS. Besides, collaborating through GitHub and utilizing existing ROS packages are also valuable lessons I've learned. Finally, I gradually recognized that it might be better to use velocity controller in "hitting" task, so that we can control the intermediate velocity when tracking and hitting the object.

5 Results with Video URL

Even though the robot misses the ball sometimes, at most of the time the robot can successfully hit back the ball as expected. Figure 8 shows the screenshot sequence of one successful test. You are welcome to download our project and play with the robot on your own computer. A brief instruction is on the Git repository.

Besides, we also made a video of running the project and uploaded it onto YouTube. If you are interested, please have a look by this URL link: https://youtu.be/QV3v8rrRdhA



(a) Robot starts to move. (b) Robot is approaching the ball. (c) Robot is contacting the ball with bat. (d) The ball is hit back.

Figure 8: Screenshot sequence of one successful test.

6 Discussions

Since our project is simulated in Gazebo environment and the camera image rendering process by Gazebo is high load, the image frequency of the camera is dramatically bottle-necked by the computer even though we have already set the update rate of the camera sensor plugin to be very high (e.g. we set the update rate of camera sensor to be 200Hz but we only get roughly 40Hz image messages by inspecting the related topics, even though we are using a low-resolution camera, ie. 300×300). Using a low-resolution camera causes the problem that the ball is too small or even becomes invisible in some camera images. And a low image frequency causes the problem that we only get relatively sparse observations, which may lead to relatively large errors when fitting the trajectory. Fortunately, even so we still get accurate results. However, if we run the project in real world with real high-speed normal-resolution cameras, we believe the trajectory prediction could be more robust with dense observations and there would be more space for us to think about adding more ingredients.

Our project running in Gazebo uses a relatively easy physical model compared to real world (e.g. we do not add air drags), and the camera model is ideal. So if we would like to run our project on real robot in the future, we need to further think about ways to deal with camera noises and more complicated ball motions. This will involve adding an image denoising module, and using a modified Kalman Filter model (e.g. by including the acceleration or air drag imposed on the ball into the state vector to be estimated). Besides, we can consider using a higher-order polynomial fitting or solving differential equations to predict more complicated ball trajectory in real world.

For robot control part, if we are going to apply the same strategy on a real robot rather than in simulation environment, the parameters throughout the control pipeline should be adjusted accordingly (e.g. maximum velocity constraints in Reflexxes function, joint limits, etc.). Besides, collision in a real robot would be severe, so we might have to implement a self-collision avoiding package for the safe concern.

For a object hitting or tracking task, it might be better to utilize a velocity controller than a inverse kinematics controller, because we don't care about the intermediate poses and the target poses is constantly changing throughout the whole process. We have configured a hardware interface for velocity controller in Gazebo simulation environment and successfully loaded the existing velocity controller. We haven't implemented velocity-based control strategies on those controller and interface. I'm curious about the performance of velocity controllers in the future.

7 Suggestions

To give suggestions to future projects, it would be good for one to first search for what tools or open-source libraries they can use and to know what functionalities the tools provide before they decide what to do in the project. In my opinion, doing this search could help with knowing what is practical and what is impractical, and it also helps one to have a rough idea of what challenges he or she is going to meet, which is important for making a well-scheduled plan and working efficiently.

References

[1] http://gazebosim.org/tutorials.
[2] https://github.com/CentroEPiaggio/kuka-lwr.
[3] https://www.orocos.org/kdl.
[4] http://wiki.ros.org/vision_opencv.
[5] http://wiki.ros.org/message_filters.