Multi-Agent Distributed Coverage Control using Quadcopters

Gerardo Gonzalez
University of California, San Diego
Summer 2015

1. Summary

This summer, the Cortes and Martinez lab at UCSD began to work on the implementation of a quadcopter testbed. For this project, I focused on the development of the network controls and a network path following algorithm. In order to implement this functionality, we used the Robot Operating System (ROS) to handle all communication and control processing throughout the network. In every step of the project, we came across code bugs and optimization problems that are documented in this report. Since the beginning of the summer, much progress has been made in the establishment of a control and localization system for the quadcopter network, and we are now ready to begin scaling the network with more quadcopters.

2. Big Picture

A distributedly controlled system is a control system in which each robot is in charge of its own controls; thus, each robot is self-sufficient and only requires its own sensed information to control itself. This type of control system is especially useful when considering formation control, a topic which has much application in the real world, including exploration and surveillance. For example, search and rescue missions would benefit greatly from this type of network. Rather than deploying a team of humans into a danger zone, quadcopters equipped with cameras would be used to do the required sensing. The world of distributed control is an exciting one, but before any of this can become reality, a basic question must be addressed: how are we going to test these algorithms?

In general, distributed control works best when applied to active sensor networks, which are networks of autonomous agents that are specifically configured to sense the environment. Thus, it is beneficial for us to develop a testbed that is in itself comprised of an active sensor network. That is why the Cortes and Martinez lab at UCSD has decided to build a network of quadcopters that can communicate with each other in order to simulate distributed control algorithms. For this to happen, we want the robots to know how to communicate, localize, and control, autonomously. Communication will be established using the Robot Operating System

(ROS). Localization will be mainly implemented using the Simultaneous Localization and Mapping (SLAM) algorithm. Finally, movement will depend on the specific control algorithm being used, which for our purposes will be a formation control one that uses Voronoi cells to distribute the quadcopters. More detail will be provided in the methodology section.

3. My Contributions

3.1 Controller

One of my tasks this summer was to aid in the implementation of a control law for the quadcopter system. Initially, we opted to use a proportional-integral-derivative controller (PID controller) as our default control law. During this time, I was in charge of coding the controller and tuning its parameters in order to reduce overshoot, oscillation and steady-state error. Thereafter, we decided to try out the pseudo-sliding mode (PSM) controller and compare it to the PID controller. For the PSM controller, I was in charge of coding the controller in c++ and ROS. Upon comparing the two controllers, we found that the PSM controller had a faster response time in stabilizing the error. As a result, we made PSM our default controller. From then on, I was in charge of tuning the PSM to reduce oscillation and overshoot. More detail will be provided in the methodology section.

3.2 Path Following Algorithm

My last project this summer was to develop a path following algorithm for the testbed. For this project, I was in charge of coding the node that executes the algorithm. I organized the main ROS loop so that it can determine whether the path is open or closed. From there, the algorithm executes a series of functions that instruct the quadcopter to stay on the path at a constant velocity along the direction of the path. This is accomplished by publishing a goal position for the quadcopter that is closest to the path. This closest point on the path is determined using interpolation along the path and distance measurements from path points to the actual quadcopter position..

4. Methodology

Before detailing the methodology behind my work, it is important to understand ROS and how it works. ROS is the heart of communications for the network; it works by using a system

known as a Publisher/Subscriber model. In this system, there are executable files of code called "nodes" which can publish (send data) and subscribe (extract data) to a "topic" (communication bus), which serves as a data center and intermediary step for node communication. For example, in our system, there is a node that is in charge of calculating the voronoi centroids and reporting them as goal positions to the quadcopters. This node is both a subscriber and publisher; it will subscribe to a topic that collects information about the current location of the quadcopters and also publish to a topic that collects information about where the quadcopters should navigate to. The entire quadcopter network communicates using this model.

4.1 Controller

The PSM controller depends on the error of the quadcopter state and on its current estimated velocity and outputs a velocity in order to converge the error to zero. We use ROS to update the current estimated position of the quadcopter, the user-defined goal position for the quadcopter, velocity estimations, and PSM parameters. Both the estimated position and the velocity estimation are updated by the extended-kalman filter node while the goal position is determined by the android application. The controller node works by subtracting the goal position from the estimated position to determine the error. This error is used along with the velocity estimation and the pre-tuned PSM parameters to control the quadcopter.

4.2 Path Following Algorithm

The path following node works by processing an array of positions that trace out the path and are published by the android application. Once these positions are processed, the path following algorithm executes and controls the movement of the quadcopter along the path. The main loop of the node works as follows: the first thing the node checks for is a new path and new updates on the state of the quadcopter. After spinning ROS and retrieving this information, the loop checks an updated flag to see if a new path has been received and, if so, whether the path is open or closed. Afterwards, some more pre-processing of the path array is done again to determine the length of the array and its last index value. Also, the index to the closest point on the path is determined, and the quadcopter is instructed to move towards it. A series of instructions are then looped that tell the quadcopter to continue moving along the path at a constant velocity until the end of the path is reached (that is, the last index is seen). More detail can be seen under the Code section.

5. Tips

When first implementing a controller, it is particularly difficult to tune its gains in order to provide maximum efficiency. What facilitated this process for me was using a callback to allow real-time manipulation of gain values. The way this works is by defining a ROS topic and message that, when called, automatically update the value of the gains. This is the callback I for the PSM controller:

```
// Updates slider gain values
void sliderGainCallback(const geometry_msgs::Vector3::ConstPtr& gainPtr)
{
   sliderSlope = (double) gainPtr -> x;
   sliderGain = (double) gainPtr -> y;
   kd = (double) gainPtr -> z;
}
```

The purpose of this callback is to manually update the parameters of the control law while testing it on the quadcopter. The way the parameters work is as follows: changing the slider slope will affect the response time and overshoot, the gain will affect the response time, and the derivative gain (kd) will affect the oscillation of the control system. When used correctly, this tool can be very efficient at quickly finding stabilizing gains for the PSM controller. To implement this callback in real-time, use the ROS command “rosrun pub” to send a message to the specific topic designated for the callback.

6. Pitfalls

There are still bugs in the path following node that have not yet been resolved. For example, publishing more than one path for the quadcopter to follow will cause the path following node process to terminate on ROS, killing the quadcopter driver. This problem has yet to be resolved. Furthermore, we found that ORB SLAM (the computer vision algorithm we use to localize the quadcopter), while consistent in its localization data, is susceptible to network and landmark loss interference, which is due to the algorithm’s high dependence on both. While the network problem is resolved with the addition of an extended kalman filter (which keeps localizing the quadcopter using the accelerometer data), landmark loss interference proves to be a bigger challenge because there are no alternatives; the algorithm requires initialization time for the landmarks and depends on these landmarks for localization. A possible solution for the landmark loss problem is to give the algorithm sufficient time to initialize over a larger area landmark.

7. Conclusions & Future Directions

The main objective of this paper is to outline the methodology behind the development of a testbed to be used in the testing of distributed control algorithms. So far, we have developed localization, distributed path planning and motion planning subsystems to the point where they can be tested using a single quadcopter. Future directions include the implementation of various distributed control algorithms, the path planning and obstacle avoidance problem, and the addition of deep learning algorithms. For the distributed control algorithms, it is only a matter of time before we are able to scale the testbed and directly implement them. This will provide valuable feedback on the nature of the algorithms and answer questions regarding their applicability outside the theory. Further on, it will be interesting to observe the path planning and obstacle avoidance problem when dealing with a distributedly controlled network. This research question is of high practical value, especially for autonomous agents that are dependent on their own decision making while constantly being situated in uncertain environments. Finally, deep learning is a research topic that comes at a low cost for us to investigate due to the monocular cameras that are already available on our quadcopters and the vast, open source libraries of software that have already been developed for it. Indeed, this testbed has fascinating potential applications for research.

8. Code

ROS main function (PSM controller):

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "Quadcopter Hover: version 1, EKF pose estimations only"); //Ros Initialize
  ros::start();
  ros::Rate loop_rate(T); //Set Ros frequency to 50/s (fast)
  srand (time(NULL));
  ros::NodeHandle n;
  ros::Subscriber poseEstSub;
  ros::Subscriber poseGoalSub;
  ros::Subscriber velEstSub;
  ros::Subscriber pidGainSub;
  ros::Subscriber pidGainZSub;
  ros::Subscriber pathVelSub;
  ros::Publisher velPub;
  ros::Publisher poseSysIdPub;
  ros::Publisher velPoseEstXPub;
```

```
poseEstSub = n.subscribe<geometry_msgs::PoseStamped>("/poseEstimation", 1, poseEstCallback);
velEstSub = n.subscribe<geometry_msgs::Twist>("/velocityEstimation", 1, velocityEstCallback);
poseGoalSub = n.subscribe<geometry_msgs::PoseStamped>("/goal_pose", 1, poseGoalCallback);
pidGainSub = n.subscribe<geometry_msgs::Vector3>("/pid_gain", 1, pidGainCallback);
pidGainZSub = n.subscribe<geometry_msgs::Vector3>("/pid_gainZ", 1, pidGainZCallback);
pathVelSub = n.subscribe<geometry_msgs::Twist>("/path_vel", 1, pathVelCallback);
velPub = n.advertise<geometry_msgs::Twist>("/cmd_vel", 1000, true);
poseSysIdPub = n.advertise<geometry_msgs::Vector3>("/sys_id", 1000, true);
velPoseEstXPub = n.advertise<geometry_msgs::Vector3>("/vel_poseEstX", 1000, true);

// Initialize msgs
poseGoal.pose.position.x = 0;
poseGoal.pose.position.y = 0;
poseGoal.pose.position.z = 0;
poseError.pose.position.x = 0;
poseError.pose.position.y = 0;
poseError.pose.position.z = 0;
velocity.angular.x = 1;
velocity.angular.y = 0;
pastError.pose.position.x = 0;
pastError.pose.position.y = 0;
pastError.pose.position.z = 0;
pathVel.linear.x = 0;
pathVel.linear.y = 0;
pathVel.linear.z = 0;
pathVel.angular.x = 0;
pathVel.angular.y = 0;
pathVel.angular.z = 0;

velPoseEstX.z = 0;

while (ros::ok())
{

   ros::spinOnce();

   calculateError();

   PID();

   velPub.publish(velocity);
   poseSysIdPub.publish(poseSysId);
   velPoseEstXPub.publish(velPoseEstX);

   std::cout<<"Twist: \n"<<velEstimation.linear<<"\n\n";
   std::cout<<"----------------------------------------------------------------";
   loop_rate.sleep();
```

```
    }
}
```

ROS main function (path following algorithm):

```
int main(int argc, char **argv)
{
    ros::init(argc, argv, "path_follower"); //Ros Initialize
    ros::start();
    ros::Rate loop_rate(T); //Set Ros frequency to 50/s (fast)

    ros::NodeHandle n;
    ros::Subscriber pathSub;
    ros::Subscriber poseEstSub;
    ros::Publisher goalPub;
    ros::Publisher velPub;

    pathSub = n.subscribe<geometry_msgs::PoseArray>("/path", 1, pathCallback);
    poseEstSub = n.subscribe<geometry_msgs::PoseStamped>("/poseEstimation", 1, poseEstCallback);
    goalPub = n.advertise<geometry_msgs::PoseStamped>("/goal_pose", 1000, true);
    velPub = n.advertise<geometry_msgs::Twist>("/path_vel", 1000, true);

    // Initialize msgs and flags
    newPath = false;
    closestPointOnLine.pose.position.x = 0;
    closestPointOnLine.pose.position.y = 0;
    constVelTerm.linear.x = 0;
    constVelTerm.linear.y = 0;
    constVelTerm.linear.z = 0;
    constVelTerm.angular.x = 0;
    constVelTerm.angular.y = 0;
    constVelTerm.angular.z = 0;

    while (ros::ok())
    {
        ros::spinOnce();

        if(newPath) // check if a new path has been set
        {
            findIndexOfLastPointOnPath();
            if(isOpenLoop) // path given is OPEN
            {
                newPath = false; // reset flag
                goalPose.pose = (pathPose.poses)[0]; // publish first point on path
                goalPose.pose.orientation = tf::createQuaternionMsgFromYaw(0);
                goalPub.publish(goalPose);
```

```
            while( distanceFormula(pathPose.poses[0].position.x, poseEst.pose.position.x,
                                pathPose.poses[0].position.y, poseEst.pose.position.y) >= BOUNDARY_RADIUS ) // FIXME:
Implement this sleep cycle as a function
              {
                 ros::spinOnce();
                 loop_rate.sleep();
                 if(newPath || !ros::ok())
                 {
                    break;
                 }
              }
            closestPointIndex = 0; // initialize to first point in path
            while(closestPointIndex != lastPointOnPathIndex) // use interpolation
            {
              if(newPath || !ros::ok()) // FIXME: break out if a different pose is published
              {
                 break;
              }
              ROS_INFO("on interpolation loop OPEN\n");
              findClosestPointOnLine();
              closestPointOnLine.pose.orientation = tf::createQuaternionMsgFromYaw(0);
              calcConstVelTerm();
              std::cout << "Goal pose:\n" << closestPointOnLine << "\n\n";
              std::cout << "Constant vel:\n" << constVelTerm << "\n\n";
              velPub.publish(constVelTerm);
              goalPub.publish(closestPointOnLine);
              //pathPose.poses[closestPointIndex].position.x = 0;
              //pathPose.poses[closestPointIndex].position.y = 0;
              calcClosestPointOnPath();
              ros::spinOnce();
              loop_rate.sleep();
            }
            goalPose.pose = (pathPose.poses)[lastPointOnPathIndex]; // publish final point on path
            goalPose.pose.orientation = tf::createQuaternionMsgFromYaw(0);
            goalPub.publish(goalPose);
            //pathPose.poses[closestPointIndex].position.x = 0;
            //pathPose.poses[closestPointIndex].position.y = 0;
                        constVelTerm.linear.x = 0;
                        constVelTerm.linear.y = 0;
                        velPub.publish(constVelTerm);
                        // FIXME: reset path variables
                        //prevClosestPointIndex = 0;
          }
          else // path given is CLOSED
          {
            newPath = false; // reset flag and set stopping condition for while loop
            calcClosestPointOnPath();
            sortPathArray(); // array now starts at the closest point index
```

```
            closestPointIndex = 0;
            while( !newPath || ros::ok() ) // while no new path has been published
            {
               ROS_INFO("on interpolation loop CLOSED");
               findClosestPointOnLine();
               closestPointOnLine.pose.orientation = tf::createQuaternionMsgFromYaw(0);
               calcConstVelTerm();
               std::cout << "Goal pose:\n" << closestPointOnLine << "\n\n";
               std::cout << "Constant vel:\n" << constVelTerm << "\n\n";
               velPub.publish(constVelTerm);
               goalPub.publish(closestPointOnLine);
               calcClosestPointOnPath();
               if (closestPointIndex == lastPointOnPathIndex)
               {
                  closestPointIndex = 0;
               }
               ros::spinOnce();
               loop_rate.sleep();
            }
            constVelTerm.linear.x = 0;
                        constVelTerm.linear.y = 0;
                        velPub.publish(constVelTerm);
                        ROS_INFO("finished CLOSED loop");
         }
      }

      loop_rate.sleep();
   }
}
```