Controller and Path Following Algorithm

Jose Ramirez
UC, San Diego

August 31st, 2015

Abstract

The implementation of the Slider Controller and Path following algorithm within the Quadcopters was motivated by the recent interest of unmanned exploration. The slider controller offers an effective method for the quadcopter to travel, by offering a steady and precise movement. This allows for the implementation of different algorithms, such as our path following algorithm, which allows our quadcopter to move along a predefined path. This report explains the functionality of the Slider controller and Path following algorithm.

Big Picture

Slider controller is fundamental in the development of algorithms for the the Quadcopter. Its functionality depends on two components, velocity position error. According to the computation of error between the desired estimated current position of the quadcopter, a tuned PID controller determines the velocity needed to minimize the error of an approximate goal position. By utilizing this PID controller, the team developed a path following algorithm. This algorithm will be fundamental to the teams goal of unmanned exploration.

1.1 Slider Controller

Filtering Data.

Gerardo Gonzalez and I implemented a function in the Slider controller program to better analyze data points through a moving average calculation by creating a series of averages of subsets of the full data set (10 samples per data point). This served to improve the results of the PID controller, resulting in a smoother trajectory for the quadcopter. For more information please see my contributions section.

Generating the Slider Controller Graph.

A fundamental part of the Slider controller is the implementation of its graph. As mentioned this graph consist of position error as a function of velocity. According to the error, the quadcopter will modify its velocity so that it will minimize the error between where it is and where it wants to be. The possibility of an overshoot adds complexity to form of the graph, since certain scenarios might calculate suboptimal velocities. For more information please refer to my contributions.

1.2 Path Following Algorithm

Path data points.

Aaron Ma developed a publisher in ROS that sends the coordinates of a predetermined path. Gerardo developed a subscriber that receives these points and commands the quadcopter to travel through them. In order do this, the subscriber goes through the following process. First it calculates the closest data point to the current location of the quadcopter. After this, the quadcopter will identify the next closest data point. It will then perform an interpolation between these two points. The process of interpolation benefits the quadcopters slider controller output velocity, by utilizing closer distances for the calculation of percent error.

Interpolation.

The method of interpolation we utilized was a recursive midpoint calculation between two points. First it calculates the midpoint between the current next data points of the path. After finding the midpoint, depending on the location of the quadcopter, the algorithm will choose either the left side or right side of the midpoint, and calculate a new midpoint between the previous midpoint and the closest data point. This process continues for a defined number of recursions. At the end, the quadcopter will proceed to move to this point by utilizing it in the slider controller.

My Contributions

My objective within the Slider Controller was to implement a data filtering mechanism in order to improve the quality of the results. I elaborated a function that would take up a predefined number of inputs, and it would proceed to calculate the moving average of these numbers. The moving average of these numbers was stored in an array, which was later used as the input of our PID controller. This operation is being called every time the

quadcopter utilizes the slider controller. The result of using this function is an improved velocity output. I also participated in the development of the piecewise function for the Slider controller. This function was created to resolve complications with overshooting. Our previous linear model, had a hard time whenever an significant large overshoot occur. The form of the piecewise function was mostly Cubic to better adress overshooting issues and linear near the origin for more precision with the output velocities. My next goal was to implement an interpolation mechanism for the path following algorithm. I first decided to try a linear approach, by creating a function that would find the equation of the line between two data points. This was done simply by using the information of the data points (x y coordinates), to come up with the equation y = m(x-xo)+ yo. The team decided to move with a different interpolation approach. The method of interpolation that was decided was a recursive midpoint calculation between two points. In order to use this method, I developed a distance formula function that is implemented when finding the midpoints. As explained before, the quadcopter identifies the current next closest data points. It then proceeds to calculate the midpoint between these two. Next it chooses a side between the current/next closest and the midpoint, and it proceeds to calculate once again a new midpoint. It proceeds to do this for a predefined number of iterations.

Methodology

4.1 Slider Controller.

Aaron Ma, Gerardo Gonzales, and I worked together in writing the code for the Slider controller, which is a fundamental for the development of algorithms for the quadcopters. The first step was to come up with an equation for Velocity as a function of percent error. Currently our function is a straight line, but in the future we are looking at modifying to a piece wise function. This way it will better handle scenarios like overshooting. In order to refine the data output, the team decided to integrate a moving average function, that would take the average of n inputs. For the moving average function, the following equation was used:

$$MA = \frac{(Pe + Pe + 1 + ... + Pe + n)}{n} \tag{1}$$

4.2 Path Following

The process for the path following algorithm is the following. First a series of data points are sent from a tablet running an android app developed by Aaron. The algorithm traces these data points, and depending on the last

one, labels the path as open or close. From here it deploys the distance formula, in order to find the closest data point with respect to the current position of the quadcopter.

$$Distance = \sqrt{(x - xo)^2 + (y - yo)^2} \tag{2}$$

Tips

The quadcopter division has just recently been created, so there is a lot to learn on how to best operate the quadcopters. We utilized several processes to improve our data, such as the moving average and Interpolation, since we need to account real life factors that might affect the quadcopter. The most common issue is Overshooting. Because of this, make a priority in future algorithms, the optimization of the data.

Pitfalls

The current method of interpolation has some issues with specific paths. Currently, when the path is a loop, the algorithm tends to tell the quadcopter to repeat its trajectory. This is due to the fact that our algorithm is identifying the closest point to be one that we have already passed. We have tried different methods to solve this, but so far we have not been able to resolve this issue.

Conclusions & Future Directions

The work so far has been the foundation for future algorithms that the Quadcopter division has in mind. With the Slider controller path following algorithm in optimal condition, the team is looking forward towards utilizing the camera for object recognition. This will be useful for avoiding collisions, and unmanned exploration.

Code

```
void calcMoveAvg(float newSampleX,
float newSampleY, float newSampleZ, float newSampleYaw)
{
  static bool divideSampling = false;

  if ((maIndex - 1) == 10)
  {
    maIndex = 1;
    divideSampling = true;
  }

  maArrayX[maIndex - 1] = newSampleX;
  maArrayY[maIndex - 1] = newSampleY;
  maArrayZ[maIndex - 1] = newSampleZ;
  maArrayYaw[maIndex - 1] = newSampleYaw;

  maResults[0] = 0;
  maResults[1] = 0;
  maResults[2] = 0;
  maResults[3] = 0;

  for (int i = 0; i < numSamples ; i++)
  {
    maResults[0] += maArrayX[i];
    maResults[1] += maArrayY[i];
    maResults[2] += maArrayZ[i];
    maResults[3] += maArrayYaw[i];
  }

  if (divideSampling != true)
  {
    maResults[0] = maResults[0]/maIndex;
    maResults[1] = maResults[1]/maIndex;
    maResults[2] = maResults[2]/maIndex;
    maResults[3] = maResults[3]/maIndex;
  }
  else
  {
    maResults[0] = (maResults[0]/numSamples);
    maResults[1] = (maResults[1]/numSamples);
    maResults[2] = (maResults[2]/numSamples);
    maResults[3] = (maResults[3]/numSamples);
  }

  maIndex++;
}


double distanceFormula (double x3, double x2, double y3, double y2)
{
  double c = 0;
  c = sqrt ( pow(x3 - x2, 2) + pow(y3 - y2, 2) );
  return c;
}


// This function will identify the closest point on the path to
the quadcopter
bool calcClosestPointOnPath (void)
{
  double closestDistance = 0; // distance from pose estimation
                                    to closest point on the path
  double tempClosestDistance = 0;

  for(int i = 0; i < pathPose.poses.size(); i++)
  {
    if ( (pathPose.poses[i].position.x == 0) && (pathPose.poses[i].position.y == 0) )
        {
            break;
        }
    tempClosestDistance = distanceFormula ( pathPose.poses[i].position.x, poseEst.pose.position.x,
                                        pathPose.poses[i].position.y, poseEst.pose.position.y );

    if ( (closestDistance == 0) || (closestDistance > tempClosestDistance) )
      {
        closestDistance = tempClosestDistance;
        closestPointIndex = i;
      }
  }
}


// Interpolates to find closest point on the path using the bisection method
void findClosestPointOnLine(void)
{
    double point1[2] = {0,0};
```

```
    double point2[2] = {0,0};

    point1[0] = pathPose.poses[closestPointIndex].position.x;
    point1[1] = pathPose.poses[closestPointIndex].position.y;
    point2[0] = pathPose.poses[closestPointIndex + 1].position.x;
    point2[1] = pathPose.poses[closestPointIndex + 1].position.y;
    nextPointClosest.pose.position.x = point2[0];
    nextPointClosest.pose.position.y = point2[1];

    double distance1 = 0;
    double distance2 = 0;
    for(int i=0; i<NUM_ITERATIONS; i++)
    {
        distance1 = distanceFormula(poseEst.pose.position.x, point1[0], poseEst.pose.position.y, point1[1]);
        distance2 = distanceFormula(poseEst.pose.position.x, point2[0], poseEst.pose.position.y, point2[1]);
        calculateMidPoints (point2[0], point1[0], point2[1], point1[1]);
        if(distance1 < distance2)
        {
            point2[0] = midpoints[0];
            point2[1] = midpoints[1];
        }
        else
        {
            point1[0] = midpoints[0];
            point1[1] = midpoints[1];
        }
    }

    closestPointOnLine.pose.position.x = midpoints[0];
    closestPointOnLine.pose.position.y = midpoints[1];
}


}
```