% Draw image

drawImageFromPix(segmentsPix,xLimPix,yLimPix,Z\_i)

| function drawImageFromPix(segmentsPix,xLimPix,yLimPix,Z\_i)  nSegments = length(segmentsPix);  % Define whiteboard limits  load WhiteboardLimits.mat xLim yLim  % Convert pixel coordinates to physical distances and then to encoder counts  fraction = 0.7;  segmentsMeters = transformPixelsToMeters(segmentsPix,xLim,yLim,xLimPix,yLimPix,fraction);   | function segmentsMeters = transformPixelsToMeters(segmentsPix,xLim,yLim,xLimPix,yLimPix,fraction)  % This function computes the scale factor for converting pixels to meters, then converts all the segments in segmentsPix to meters.  % First, create variables to represent the ranges xRange, yRange, xRangePix, and yRangePix.  xMinM = xLim(1);  yMinM = yLim(1);  xRangeM = diff(xLim);  yRangeM = diff(yLim);  % Determine the range of the coordinates to draw  xMinPix = xLimPix(1);  yMinPix = yLimPix(1);  xRangePix = diff(xLimPix);  yRangePix = diff(yLimPix);  % Calculate the two possible scale factors. The fraction factor indicates what percentage of the available space should be used.  % Scale from pixels to real world units (meters)  xScaleFactor = fraction\*xRangeM/xRangePix;  yScaleFactor = fraction\*yRangeM/xRangePix;  % Pick the smaller scale factor. If both are NaN, pick 0.  pix2M = min(xScaleFactor,yScaleFactor);  if isnan(pix2M)  pix2M = 0;  end  % Based on the chosen scaling factor and range values, identify the new origin for the scaled drawing so that the entire drawing will be centered at the center of the drawable area.  % Identify the origin of the scaled drawing  centerMeters = [xMinM yMinM] + [xRangeM yRangeM]/2;  drawingOriginM = centerMeters - pix2M\*[xRangePix yRangePix]/2;  % Loop through all segments and transform pixel values to meters for all coordinates. Subtract the pixel origin, multiply by the scaling factor, and add the drawing origin.  segmentsMeters = cell(size(segmentsPix));  nSegments = length(segmentsPix);  for ii = 1:nSegments  % Scale all segments by the computed scaling factor  coordsPix = segmentsPix{ii};  coordsPix = fliplr(coordsPix);  %Convert from row,col to x,y  coordsMeters = pix2M\*(coordsPix-[xMinPix yMinPix]) + drawingOriginM;  segmentsMeters{ii} = coordsMeters;  end | | --- |   % Reduce size of each segment  radius = 0.002;  %Max distance between points to draw (meters)  for ii = 1:nSegments  segmentsMeters{ii} = reduceSegment(segmentsMeters{ii},radius);  end   | function reducedSegment = reduceSegment(segment, radius)  % First, create variables that will be used to decide which points to keep in the reducedSegment  nPoints = size(segment,1);  keepPoint = true(nPoints,1);  reference = 1;  test = 2;  % Loop through the segment and decide which points to remove. By default, all points will be included in the output. Select points to keep and remove such that each consecutive point remaining is separated by a minimum distance from the previous point.  while test < nPoints  % Check if the distance between the reference point and the current test point is less than the specified minimum radius. If it is, mark the test point for removal and test the next point in the segment.  if norm(segment(test,:) - segment(reference,:)) < radius  % If so, mark test point for removal and test the next point  keepPoint(test) = false;  test = test + 1;  % If the distance between the two points exceeds the specified radius, make the test point the new reference point, and begin testing the next point in the segment.  else  % If not, update reference and test points  reference = test;  test = reference + 1;  end  end  % Create the output by keeping only the points that are not marked for removal.  reducedSegment = segment(keepPoint,:); | | --- |   load RobotGeometry.mat Base  segmentsTheta = cell(size(segmentsMeters));  for ii = 1:nSegments  segmentsTheta{ii} = xyToRadians(segmentsMeters{ii},Z\_i,Base);  end   | function theta = xyToRadians(xy,Z\_i,Base)  % Define constants  r\_spool = 0.0045;  % Convert x and y to Z1 and Z2  x = xy(:,1);  y = xy(:,2);  Z(:,1) = sqrt(x.^2 + y.^2);  Z(:,2) = sqrt((Base-x).^2 + y.^2);  % Subtract initial Z to get change in Z  dZ = Z - Z\_i;  % Convert change in Z to change in string length  dStringLength = 2\*dZ;  % Compute change in string length to angle  theta = dStringLength/r\_spool; | | --- |   % Connect to hardware  a = arduino;carrier = motorCarrier(a);  s = servo(carrier,3);  pidML = pidMotor(carrier,2,'position',3,[0.18 0.0 0.01]);  % Modify the PID gains [Kp Ki Kd] as per your requirements  pidMR = pidMotor(carrier,1,'position',3,[0.18 0.0 0.01]);  % Define up and down positions for servo motor  load ServoPositions.mat LeftMarker NoMarker  % Draw image on whiteboard  writePosition(s,NoMarker)  for ii = 1:nSegments  % Get theta for current segment  thetaList = segmentsTheta{ii};  % Move to first position and lower marker  moveToRadians(thetaList(1,:),pidML,pidMR)  writePosition(s,LeftMarker)  % Move to all positions of current segment  moveToRadians(thetaList,pidML,pidMR)  % Raise marker  writePosition(s,NoMarker)  end  end   | function moveToRadians(theta,pidML,pidMR)  % Define the gear ratio constant`  gearRatio = 100;  % Convert the motor shaft rotations to the actual motor rotations  thetaL = theta(:,1)\*gearRatio;  thetaR = theta(:,2)\*gearRatio;  % Implement an nth order delay 'nD' to store the past angular position values  nD = 3;  % Initializing delay vectors of length 'nD' for storing the past angular position values for both the motors  thetaDelayL = zeros(nD,1);  thetaDelayR = zeros(nD,1);  % Size of the Input Sequence  N = size(theta,1);  % Set up a For-Loop to drive the motor through a set of angular position values consecutively  for i = 1:N  % Command the motors to move to the target angular position values  writeAngularPosition(pidML,thetaL(i),'abs');  writeAngularPosition(pidMR,thetaR(i),'abs');  % At each iteration step, check if the motor has completed its travel to the set angular position value by comparing its current position theta(n) with a past value theta(n-nD).  % If the motor has achieved the target position, then it will stop rotating, and the difference value will be zero.  % At the next iteration step, move on to the next angular position value and continue until all the points have been traversed.  diffL = abs(readAngularPosition(pidML) - thetaDelayL(1));  diffR = abs(readAngularPosition(pidMR) - thetaDelayR(1));  j = 1;  % Use a while loop to keep the motors running until they achieve the target position values`  while ~(diffL==0 && diffR==0)  if j > nD  % The variable 'k' increments from '1' upto 'nD' and then resets back to 1.  % This variable helps recall the past 'nD’ angular position values to compute the difference value.  k = 1 + mod(j-1,nD);  diffL = abs(readAngularPosition(pidML) - thetaDelayL(k));  diffR = abs(readAngularPosition(pidMR) - thetaDelayR(k));  % Overwrite the past values with current values once the difference value has been computed  thetaDelayL(k) = readAngularPosition(pidML);  thetaDelayR(k) = readAngularPosition(pidMR);  else  thetaDelayL(j) = readAngularPosition(pidML);  thetaDelayR(j) = readAngularPosition(pidMR);  end  j = j+1;  end  % Reset the delay vectors after every iteration step  thetaDelayL(:) = 0;  thetaDelayR(:) = 0;  end | | --- | |
| --- | --- | --- | --- | --- |