**MCECS Bot**

**People detection, tracking, and following using Kinect.**

Final Report, ECE 478

12/16/2014

Peter Depeche

pdepeche@pdx.edu

1. Accomplishments.  
     
   a) Using Kinect sensor, v.2 (aka Xbox 360), robot detects people and “follows” closest to the robot person by rotation in place only, aka in "1D" plane, by rotating around z-axis only.   
   Demo video has been made.  
     
   b) 99% accomplished: Following a person in 2D plane, along x and y coordinate. Code have been written and tested in simulation, but code could not tested on actual robot. I have no access to the lab: card-key has been deactivated.

1. **KINECT**. Advanced Image Processing for Human-Robot interaction: Face recognition, KINECT specific.  
     
    Kinect's RGB (color) camera image and depth (infrared) three-dimensional images are used for people detection and tracking.  
     
   Kinect uses following algorithms:  
    Hough transform to find and establish a groundplane.  
    Viola-Jones face detection with adoption of Haar wavelets.  
    Leg detection.  
    AdaBoost (Adaptive Boost) algorithms and cascades.  
    Bayes filters.  
    Kalman filter.  
      
     
     
   a) Hough transform.  
    The Hough transform is a feature extraction technique. It finds imperfect instances of objects within a certain class of shapes by a voting procedure. (In our case, “class” is a “plane”). This voting procedure is carried out in a parameter space, from which object candidates are obtained as local maxima in a so-called accumulator space that is explicitly constructed by the algorithm for computing the Hough transform (to z = ax + by + d).  
     
     
   b) Viola-Jones face detection with adoption of Haar wavelets.  
     
    A Haar-like feature considers adjacent rectangular regions at a specific location in a detection window, sums up the pixel intensities in each region and calculates the difference between these sums. This difference is then used to categorize subsections of an image. For example, let us say we have an image database with human faces. It is a common observation that among all faces the region of the eyes is darker than the region of the cheeks. Therefore a common Haar feature for face detection is a set of two adjacent rectangles that lie above the eye and the cheek region. The position of these rectangles is defined relative to a detection window that acts like a bounding box to the target object (the face in this case).  
   http://iwillgetthatjobatgoogle.tumblr.com/post/19358751300/viola-jones-algorithm-for-face-recognition  
     
   In the detection phase of the Viola-Jones object detection framework, a window of the target size is moved over the input image, and for each subsection of the image the Haar-like feature is calculated. This difference is then compared to a learned threshold that separates non-objects from objects. Because such a Haar-like feature is only a weak learner or classifier (its detection quality is slightly better than random guessing) a large number of Haar-like features are necessary to describe an object with sufficient accuracy. In the Viola–Jones object detection framework, the Haar-like features are therefore organized in something called a classifier cascade to form a strong learner or classifier.  
     
   The key advantage of a Haar-like feature over most other features is its calculation speed. http://en.wikipedia.org/wiki/Haar-like\_features  
     
     
   c) Adaptive Boost (aka "AdaBoost").  
    Adaptive Boosting is a machine learning meta-algorithm. Boosting is a general method that can be used for improving the accuracy of a given learning algorithm. It is based on the principle that a highly accurate or ‘strong’ classiﬁer can be produced through the linear combination of many inaccurate or ‘weak’ classiﬁers. The efﬁciency of the ﬁnal classiﬁer is increased further by organizing the weak classiﬁers into a collection of cascaded layers. Such design consists of a set of layers with an increasing number of weak classiﬁers, where each layer acts as a non-body-part rejector with increasing complexity. An input image is ﬁrst passed to the simplest top layer for consideration, and is only moved to the next layer if it is classiﬁed as true by the current layer.   
   AdaBoost separately trains three different body part detectors using their respective image databases and a ground plane detector.  
   http://en.wikipedia.org/wiki/AdaBoost  
     
     
     
   d) Bayes filter  
    A Bayes filter is an algorithm used for calculating the probabilities of multiple beliefs to allow a robot to infer its position and orientation. Essentially, Bayes filters allow robots to continuously update their most likely position within a coordinate system, based on the most recently acquired sensor data. This is a recursive algorithm. It consists of two parts: prediction and innovation. If the variables are linear and normally distributed the Bayes filter becomes equal to the Kalman filter.  
    In a simple example, a robot moving throughout a grid may have several different sensors that provide it with information about its surroundings. The robot may start out with certainty that it is at position (0,0). However, as it moves farther and farther from its original position, the robot has continuously less certainty about its position; using a Bayes filter, a probability can be assigned to the robot's belief about its current position, and that probability can be continuously updated from additional sensor information.  
   e) Kalman filter.  
    Kalman filter for tracking of a detected person. Kalman filter is an algorithm that uses a series of measurements observed over time, containing noise (random variations) and other inaccuracies, and produces estimates of unknown variables that tend to be more precise than those based on a single measurement alone. In other words, the Kalman filter operates recursively on streams of noisy input data to produce a statistically optimal estimate of the underlying system state.
2. Technical documentation of the robot.
   1. Components and subsystems of Kinect's people detection and tracking.  
        
      At the time of this writing, the ROS's people detection packages, named "Perception", was not available for required version of ROS, Indigo.   
        
      Pre-requisites:   
       ROS "operating system". It is an application that is installed on Ubuntu linux.   
       Navigation Stack packages, from ROS.org.   
        
        
      In order to detect and track people by Kinect, the following software is used:  
        
      i) "open\_ptrack" package is used for detection and tracking of people.  
      https://github.com/OpenPTrack/open\_ptrack  
        
        
      ii) Avin2 drivers are used as Kinect drivers.  
      https://github.com/avin2/SensorKinect
   2. Technical difficulties and solutions.   
        
       Most of the problems were finding the working software packages. Many of them are not available for the required version of ROS, Indigo. Even though ROS Indigo claims to hav e Kinect drivers for Indigo, they do not work. OpenNI (drivers) been purchased by Apple and is guaranteed to never work again and it does not work anymore, of course, despite being advertised as "working" on ROS.org. "freenect" drivers also do not work.  
        
      One of the solutions was to find good and working drivers such as Avin2 Kinect drivers (link to the software is given below).

Ultimate solution is to use older, stable and reliable packages (software) and avoid the hype, fad, elusive and evanescent trends.

* 1. Complete “troubleshooting manual” for the next users.   
       
     i) Online, on permanent and never-disappearing link which is guaranteed to exists for the duration of PSU existence:  
     https://projects.cecs.pdx.edu/attachments/download/5621/MCECS\_bot\_manual.pdf  
       
     ii) Same manual is also available on private, editable and constantly upgraded by students link at:

https://docs.google.com/a/pdx.edu/document/d/19D7ov4Ec-Qx6J-\_wtyYpQrofYBhUFdMeBDytVWC7b4Q  
Manager of this editable copy of the manual is Richard Armstrong.  
  
  
In general, the troubleshooting in ROS and in this project is done by running **RVIZ** graphical development environment (interface) to the robot as described in ROS.org tutorials.   
Also, detailed and extensive troubleshooting procedures and methods are described in the above cited MCECS Bot's Manual.  
  
Another tool used for troubleshooting is **rqt\_graph** which is also part of ROS's graphical development interface. It allows to see the nodes and topics and how they are related to each other. A snapshot of output of rqt\_graph is very large graphic and could not be fit here. It is attached as a separate file named rosgraph.png: <rosgraph.png>

* 1. Personal advice to the next students about the project like this.  
       
     i) Work should begin by finding and installing available and working people detection and tracking packages (software), and experiencing the look and feel of recognition system, just playing with it. In the case of this author, about 50% project time was spent on finding, installing and testing the packages. Many times they were either not available or did not work.  
       
       
     ii) Where should they find information? All information about this project is avaiilable on internet.  
       
     iii) Meetings should be set up just like previous meetings which a re managed by Richard Armstrong. Richard Armstrong's usefullness, help, knowledge and pedagogic value is well above of any teaching assistant. Richard is always ahead of lectures, preparing students for what's coming, familiarizing with lecture topics, and often providing scientific knowledge which is not in the lecture slides. One is very lucky, indeed, to be given a chance to work on Richard's team.  
       
       
     iv) Useful webpages and books:  
     Lecture slides.  
     ROS's tutorials and wiki pages.   
       
       
       
       
     v) What should be added to class slides that would help in this project?  
     Class already has abundance and wealth of material.

1. Technical documentation to software of robot.   
     
   The source code is located at:

https://projects.cecs.pdx.edu/projects/roboticsclub-mcecsbot  
This link is permanent and is guaranteed to exists as long as PSU exists.  
  
Two files required to follow a closest detected person are detector\_array\_listener.cpp and kinect\_nav\_goal\_listener.cpp. The source is quoted below. Files should be placed in catkin-ized directories as described in ROS.org tutorials. Created by catkin\_create\_package catkin source directories should contain the usual CMakeLists.txt and package.xml files contents of which is quoted below.  
  
  
============== Beginning of File "detector\_markers\_array\_listener.cpp: =========================  
// detector\_markers\_array\_listener.cpp

// Subsribes to /detector/markers\_array topic which is published by open\_ptrack

// Calculates the position, x and y coords, of closest detected person.

// Publishes "x" and "y" coords of closest detected person.

// Prints to stdout, if desired, person's position coorditates x, y, z,

// and angle from center.

// 2014.12.15

#include "ros/ros.h"

#include "std\_msgs/String.h"

#include "geometry\_msgs/Twist.h"

#include <math.h>

#include "geometry\_msgs/Pose.h"

#include "geometry\_msgs/PoseStamped.h"

#include "visualization\_msgs/MarkerArray.h"

#define PRINT\_OUTPUT 0 // enables stdout debugging output if set to "1"

class DetectorMarkersArrayListener

{

public:

DetectorMarkersArrayListener();

void callBack(const visualization\_msgs::MarkerArray::ConstPtr& msg);

private:

ros::NodeHandle nh;

ros::Subscriber sub;

ros::Publisher pub;

};

DetectorMarkersArrayListener::DetectorMarkersArrayListener()

{

pub = nh.advertise<geometry\_msgs::Twist>("/detected\_person", 1);

// pub = nh.advertise<geometry\_msgs::Twist>("/cmd\_vel", 10);

sub = nh.subscribe<visualization\_msgs::MarkerArray>

("/tracker/markers\_array", 1,

&DetectorMarkersArrayListener::callBack, this);

}

void DetectorMarkersArrayListener::callBack(const visualization\_msgs::MarkerArray::ConstPtr& msg)

{

int detections\_count;

int index;

double x, y, z;

int minArrayIndex = 0;

double minGlobalVal = 999.999; // init to arbitrary large distance

double minLocalVal = 999.999;

double minXglobalVal, minYglobalVal; // coordinates of closest person

geometry\_msgs::Twist msg\_out;

geometry\_msgs::PoseStamped pose\_stamped\_msg;

double x\_out, y\_out; // goal coordinates

const double dist\_min\_const = 0.5; // meters, dist betw robot & person

const double delta\_const = 1.0; // in meters, range for which robot

// will not start approaching again.

double dist\_min\_coeff = 0.0; // intermediate coefficient

double distance; // current actual distance between

// person and robot

detections\_count = msg->markers.size();

if (detections\_count > 0) {

for( index = 0; index < detections\_count; index++)

{

// do not dereference position coords inside of ROS\_INFO macro,

// else it will seg fault, do NOT do:

// ROS\_INFO("x: %lf", msg->markers[0].pose.position.x);

x = msg->markers[index].pose.position.x;

y = msg->markers[index].pose.position.y;

z = msg->markers[index].pose.position.z;

// There can be several people in the Field Of View.

// We could pick closest to the robot.

// wiki.ros.org/geometry/CoordinateFrameConventions:

// Distances are in meters, angles are in radians:

// X: forward, Y: left, Z: up

// Pick a closest person: min(x,y), for each element of array:

if (x < y) minLocalVal = x;

else minLocalVal = y;

if ( index == 0) {

minGlobalVal = minLocalVal;

}

else {

if (minLocalVal < minGlobalVal) {

minGlobalVal = minLocalVal;

minArrayIndex = index;

}

}

if (PRINT\_OUTPUT)

ROS\_INFO("seq %i: x: % .12f y: % .12f z: % .12f",

index, x, y, z);

}

minXglobalVal = msg->markers[minArrayIndex].pose.position.x;

minYglobalVal = msg->markers[minArrayIndex].pose.position.y;

if(PRINT\_OUTPUT)

ROS\_INFO("closest seq: %i x: % .12f y: % .12f\n",

minArrayIndex, minXglobalVal, minYglobalVal);

}

if (minXglobalVal == 0.0 || minYglobalVal == 0.0)

msg\_out.angular.z = 0.0;

else

msg\_out.angular.z = atan(minYglobalVal/minXglobalVal); // yaw

if (PRINT\_OUTPUT)

ROS\_INFO("angular.z: %f", msg\_out.angular.z);

distance = sqrt( powl(minXglobalVal, 2) + powl(minYglobalVal, 2) );

// Check if robot is already within the dist\_min:

if ((minXglobalVal == 0.0 && minYglobalVal == 0.0) ||

( distance <= dist\_min\_const ))

{

msg\_out.linear.x = 0.0;

msg\_out.linear.y = 0.0;

}

else {

dist\_min\_coeff = 1 - dist\_min\_const/distance;

msg\_out.linear.x = minXglobalVal \* dist\_min\_coeff;

msg\_out.linear.y = minYglobalVal \* dist\_min\_coeff;

}

pub.publish(msg\_out);

}

int main(int argc, char\*\* argv)

{

ros::init(argc, argv, "DetectorMarkersArrayListener");

DetectorMarkersArrayListener dmal;

if (PRINT\_OUTPUT)

ROS\_INFO("Spinning node detector\_markers\_array\_listener");

ros::spin();

return 0;

}

======================== end of file "detector\_markers\_array\_listener.cpp ================

The code of "detector\_markers\_array\_listener" simply calculates the target x and y coordinates for robot to use, based on detected person's coordinates, as follows:  


Robot ‘s orientation

================= Beginning of file "kinect\_nav\_goal\_listener.cpp: ============================

// kinect\_nav\_goal\_listener.cpp

// 2014.12.16

// Subscribes to "/detected\_person" topic published

// by "detector\_markers\_array\_listener" node

// Transforms closest detected person's coordinates to "map" frame from

// "lazer" frame. For now, "lazer" frame is assumed to have same transform

// as non-existing yet "kinect" frame. Lazer and kinect are positioned at

// approximately the same x and y coordinates relative to robot's

// base\_link frame.

// Sends transformed coordinates to ActionServer of ROS's Navigation Stack.

// Navigation Stack is required to run this file.

#include <ros/ros.h>

#include <move\_base\_msgs/MoveBaseAction.h>

#include <actionlib/client/simple\_action\_client.h>

#include <actionlib/client/terminal\_state.h>

#include "geometry\_msgs/Twist.h"

#include "geometry\_msgs/Pose.h"

#include <tf/tf.h>

#include "tf/transform\_listener.h"

#include <geometry\_msgs/PoseWithCovarianceStamped.h>

#define PRINT\_OUTPUT 1

#define RUN\_WITH\_HARDWARE\_ATTACHED 1 // set to "0" for debugging

class KinectNavGoalListener

{

public:

KinectNavGoalListener();

void detectionCallBack(const geometry\_msgs::Twist::ConstPtr& msg);

void initialPoseCallback(const

geometry\_msgs::PoseWithCovarianceStamped::ConstPtr& msg);

void amclPoseCallback(const

geometry\_msgs::PoseWithCovarianceStamped::ConstPtr& msg);

void poseStampedCallBack(const

geometry\_msgs::PoseWithCovarianceStamped::ConstPtr& msg);

void runActionClient(void);

void runConcurrencyTest(void); // test

private:

ros::NodeHandle nh;

ros::Subscriber sub\_target, sub\_initialpose, sub\_amcl\_pose;

ros∷Subscriber sub\_pose\_stamped;

ros::Publisher pub\_move\_base\_simple;

geometry\_msgs::Twist saved\_msg;

volatile double saved\_lin\_x, saved\_lin\_y, saved\_ang\_z;

actionlib::SimpleActionClient<move\_base\_msgs::MoveBaseAction> \*ac;

move\_base\_msgs::MoveBaseGoal goal;

geometry\_msgs::Quaternion q\_msg\_in, q\_msg\_out; // temp vars

tf::Quaternion current\_pose\_q, delta\_z\_q;

geometry\_msgs::PoseWithCovarianceStamped robot\_pose\_estimate;

geometry\_msgs::PoseStamped msg\_to\_publish;

bool initialPosePublished;

geometry\_msgs::PointStamped laser\_point, map\_point;

tf::TransformListener tfListener;

tf::StampedTransform tfLazerToMap;

double lazerPositionX, lazerPositionY;

tf::Stamped<tf::Pose> person\_global; // transformed point into

// "global", or "map" frame

};

// This function is called when the initialpose topic is published

// (e.g. by RVIZ user input):

void KinectNavGoalListener::initialPoseCallback(

const geometry\_msgs::PoseWithCovarianceStamped::ConstPtr& msg) {

// TODO: Only checks if "initialpose" is published. We might

// need to check if a given confidence value is reached.

initialPosePublished = true;

}

// This function is called when a pose estimation is published by AMCL node:

void KinectNavGoalListener::amclPoseCallback(

const geometry\_msgs::PoseWithCovarianceStamped::ConstPtr& msg) {

// this works when viewed by 'rostopic echo /amcl\_pose':

robot\_pose\_estimate = geometry\_msgs::PoseWithCovarianceStamped(\*msg);

}

KinectNavGoalListener::KinectNavGoalListener() // constructor

{

initialPosePublished = false;

sub\_target = nh.subscribe<geometry\_msgs::Twist>

("/detected\_person", 1,

&KinectNavGoalListener::detectionCallBack, this);

sub\_initialpose = nh.subscribe<geometry\_msgs::PoseWithCovarianceStamped>

("initialpose", 1,

&KinectNavGoalListener::initialPoseCallback, this);

sub\_amcl\_pose = nh.subscribe<geometry\_msgs::PoseWithCovarianceStamped>

("amcl\_pose", 1, &KinectNavGoalListener::amclPoseCallback, this);

pub\_move\_base\_simple = nh.advertise<geometry\_msgs::PoseStamped>

("move\_base\_simple/goal", 1);

sub\_pose\_stamped = nh.subscribe<geometry\_msgs::PoseWithCovarianceStamped>

("pose\_stamped", 1,

&KinectNavGoalListener::poseStampedCallBack, this);

// MoveBase SimpleActionClient: "true/false: spin a thread by default

// "move\_base" is not a topic here, but it is a namespace used by Server

// Server "move\_base" uses "/move\_base\_simple/goal" to send

// messages to base

ac = new actionlib::SimpleActionClient<move\_base\_msgs::MoveBaseAction>

("move\_base", true);

runActionClient(); // first, run without Kinect, manually set target

}

void KinectNavGoalListener::runActionClient(void){

while( ros::ok()) {

ros::spinOnce();

if (RUN\_WITH\_HARDWARE\_ATTACHED ) {

ROS\_INFO("checking for ActionServer.");

// ActionServer will not come up if hardware is not detected

// (not attached):

while(!ac->waitForServer(ros::Duration(3.0) ) && nh.ok() ){

ROS\_INFO(

"Waiting for the move\_base action server to come up ...");

}

ROS\_INFO("\"move\_base\" action server is running.");

}

try {

tfListener.lookupTransform("map", "lazer", ros::Time(0), tfLazerToMap);

current\_pose\_q = tfLazerToMap.getRotation();

lazerPositionX = tfLazerToMap.getOrigin().x(); // current robot

// (lazer) position

lazerPositionY = tfLazerToMap.getOrigin().y();

}

catch (tf::TransformException &ex) {

ROS\_ERROR ("%s", ex.what());

}

// angle of person in current frame ("laser" or "kinect"):

delta\_z\_q = tf::createQuaternionFromYaw(saved\_msg.angular.z);

current\_pose\_q \*= delta\_z\_q; // rotate quarternion by delta

tf::Stamped<tf::Pose> person\_local( tf::Pose(current\_pose\_q,

tf::Vector3(saved\_msg.linear.x, saved\_msg.linear.y, 0.0)), ros::Time(0), "laser" );

tfListener.transformPose("map", person\_local, person\_global);

// convert "tf::Stamped<tf::Pose> person\_local" to

// "move\_base\_msgs::MoveBaseGoal goal"

tf::Quaternion person\_global\_q = person\_global.getRotation();

tf::quaternionTFToMsg(person\_global\_q, q\_msg\_out);

goal.target\_pose.pose.orientation = q\_msg\_out;

goal.target\_pose.pose.position.x = person\_global.getOrigin().x();

goal.target\_pose.pose.position.y = person\_global.getOrigin().y();

goal.target\_pose.pose.position.z = 0.0;

goal.target\_pose.header.frame\_id = "map";

goal.target\_pose.header.stamp = ros::Time::now();

ac->sendGoal(goal);

bool finished\_before\_timeout = ac->waitForResult(ros::Duration(30.0));

actionlib::SimpleClientGoalState state = ac->getState();

if (finished\_before\_timeout) {

ROS\_INFO("\t\tFINISHED: %s\n\n", state.toString().c\_str());

}

else

ROS\_INFO("\t\tFAILED: %s\n\n", state.toString().c\_str());

}

}

void KinectNavGoalListener::poseStampedCallBack(

const geometry\_msgs::PoseWithCovarianceStamped::ConstPtr& msg)

{

// robot\_pose\_estimate = geometry\_msgs::PoseWithCovarianceStamped(\*msg);

}

void KinectNavGoalListener::detectionCallBack(const geometry\_msgs::Twist::ConstPtr& msg)

{

saved\_ang\_z = saved\_msg.angular.z = msg->angular.z;

saved\_lin\_x = saved\_msg.linear.x = msg->linear.x;

saved\_lin\_y = saved\_msg.linear.y = msg->linear.y;

}

int main(int argc, char\*\* argv)

{

ros::init(argc, argv, "kinect\_nav\_goal\_listener");

KinectNavGoalListener kngl;

if (PRINT\_OUTPUT)

ROS\_INFO("spinning node \"kinect\_nav\_goal\_listener\"");

return 0;

}

============= end of file "kinect\_nav\_goal\_listener.cpp =================================

File CMakeLists.txt and package.xml of "kinect\_nav\_goal\_listener" catkin (ROS) package should contain dependencies:

roscpp

rospy

std\_msgs

move\_base\_msgs

actionlib

tf

Packages are started (run) by:

roslaunch tracker detection\_and\_tracking.launch

rosrun detector\_markers\_array\_listener detector\_markers\_array\_listener

rosrun kinect\_nav\_goal\_listener kinect\_nav\_goal\_listener

1. Relation of this work to other work in this project.  
      
    Kinect tracking and person following uses existing "jeeves" service set up by Richard Armstrong. Jeeves service uses lazer scanner to scan for obstacles, map the surroundings of the robot, and scan for obstacles. Jeeves service starts the roscore and the Navigation Stacks of ROS.   
     
   The "/detected\_person" topic is used by robot's face to move the eyes in the direction of tracked person.
2. Future plans.  
     
    Kinects's detection algorithms are not perfect. They sometimes erroneously "detect" inanimate objects that resembles people such as office plants, furniture, etc. This "noise" might cause the robot to approach the object and try to interact with it. Further noise filtering is required to eliminate such "noise". This can be done by detection of movement, temperature of a person, and other methods.
3. Step-by-step guideline such as “INSTRUCTABLES” to the robot .  
     
    Instructions on how to install the people detection and tracking packages are given in the MCECS Bot's Manual link to which is referenced previously. There will be several errors during installation. They are software bugs. The workarounds to these errors are described in Bot's Manual. They are fixed by re-running the parts of the installation scripts.  
     
     
    The robot is started up when it is connected to battery power as described by the manual referenced above.  
   Whe n Kinect's nodes are started up by ROS's "roslaunch" and "rosrun" commands as described above, the robot will detect and follow a person.  
   The detection range is between 6 tf (1.8 m ) to 40 ft (12 m). The person's height must fit into the view of Kinect. Kinect's algorithms will first try to establish a groundplane (floor). These details are described in the MCECS Bot's Manual references (links) to which are cited previously.  
     
   When "roslaunch tracking detection\_and\_tracking.launch" is started, the stdout output will produce error message in the "tracking" launch terminal window:  
     
   “[ERROR] [ros time stamp]: transform exception: “world” passed to lookupTransform argument target\_frame does not exist.”  
     
     
   This is not an error message but a bug. This message means that algorithms are looking for a groundplane or for a person. Once a person is detected, there will be messages such as "track created" and when detection or track is lost, the message "track deleted" will appear.   
     
   This is described in details in the MCECS Bot's Manual referenced previously.