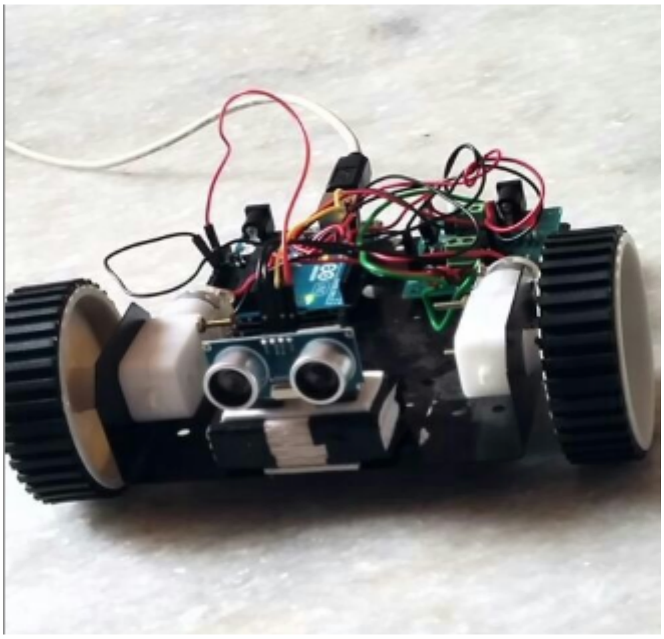
MASTER-FOLLOWER

(AUTONOMOUS DIFFERENTIAL DRIVE)



Team-05

Skanda S Bharadwaj
Sumukha B N
Chandan R Kumar
Sughosh H K

Abstract:

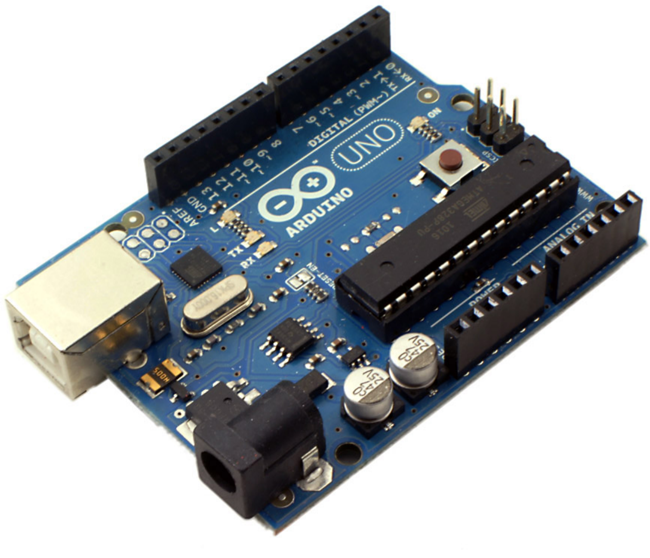
Master-follower is a autonomous differential drive bot that will follow the master bot. Distance between the master and the follower is observed using a range sensor and the data is processed using kalman filter. Position, velocity and acceleration of the bot is acquired from the kalman filter and using a controller the actuators are guided to follow the master bot.

Introduction:

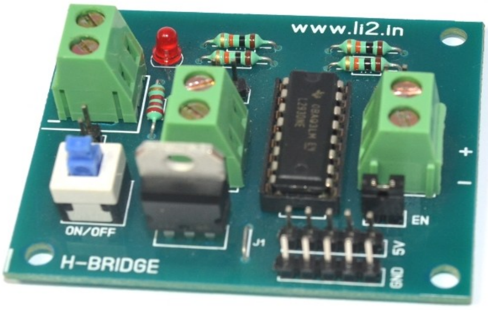
The Master follower robot uses a range sensor to acquire the distance from the master robot. This sensor acts as the observer. The observer input is fed into the kalman filter. Kalman filter uses the observer input and predicts the state variables along with predefined process noise and covariance. The output of the kalman filter is the acceleration of master depending on which the follower has to vary its velocity to catch up the master. The kalman filtering is done in python. The actuators of the follower bot is controlled using the arduino-Uno, a single board micro-controller based on atmega-328. The controller and the python are interfaced in order to serve the purpose.

Component list:

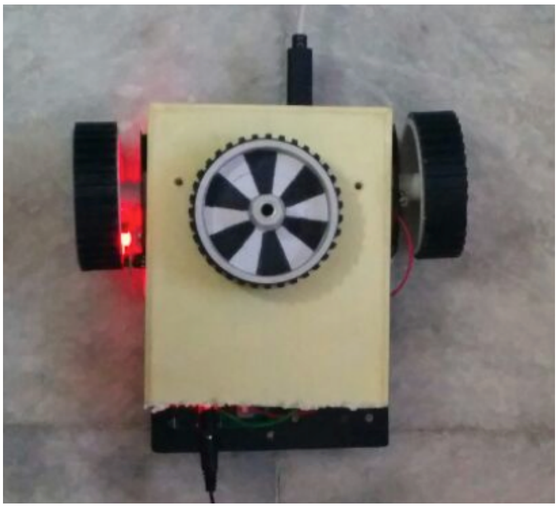
1. Arduino-UNO: A single board micro-controller based on ATMEGA328



2. H-Bridge: motor driving circuit with L293D with input ranging form 5-36v.

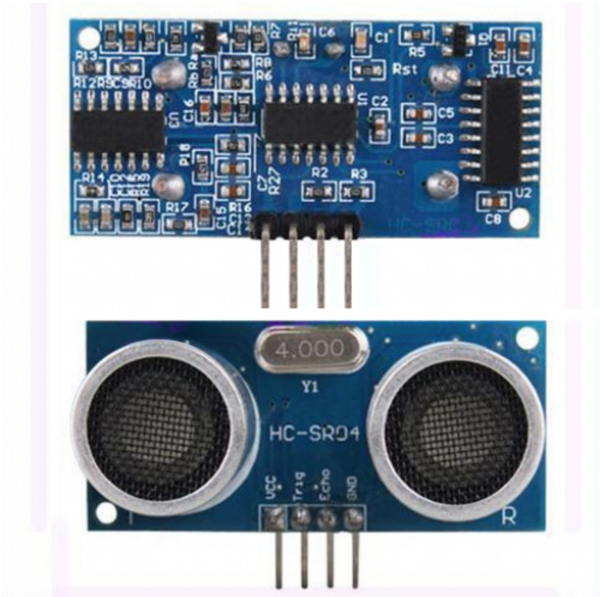


3. Encoded wheels- Regular wheel but encoded using strips of black and white



4. IR-Sensor

5. Ultrasound Range Sensor :



Working details of the follower:

The follower observes the master using a range sensor HC-SR04. This gives the distance between the master and the follower at every instant. This is serially communicated to the kalman filter. The kalman filter accepts the distance from the sensor. The initial conditions are assumed and fed to the kalman filter. The mathematical model of the follower is given below

state = [relative position, relative velocity, master acceleration]

rel_pos = rel_pos + rel_vel*timeslice

rel_vel = rel_vel + master_accel*timeslice - follower_accel*timeslice

master_accel = master_accel

Kalman Filter:

Kalman filter basically has to steps into which the algorithm can split into. The first is the prediction step and the second is the updating step. In the prediction step, the Kalman filter produces estimates of the current state variables, along with their uncertainties. Once the outcome of the next measurement (necessarily corrupted with some amount of error, including random noise) is observed, these

estimates are updated using a weighted average, with more weight being given to estimates with higher certainty. Because of the algorithm's recursive nature, it can run in real time using only the present input measurements and the previously calculated state and its uncertainty matrix; no additional past information is required.
The filter equations are

#Prediction step
predicted_state_estimate
$X\_p = AX + Bu$

#predicted probability estimate
predicted_prob_estimate
$V\_p = ACA^T + W$

#Observation step
$IC = H * V\_p * H^T + Q$

#Kalman_gain K
$K = V\_p * H^T * (IC)^{-1}$

$X = X\_p + K * (z - H * X\_p)$
$C = V\_p - K * H * V\_p$
$K = K$
$V = V\_p$

Controller code:

The filter outputs are predicted distance (relative position), relative velocity and master acceleration. The master acceleration is printed in the terminal and is read into the arduino. In the controller code a reference velocity is set for the follower bot. This is done using wheel encoder by calculating the resolution of the bot. Now the acceleration of the master which is continuously being monitored is used to manipulate the velocity the follower. Product of time and acceleration gives velocity. If the master accelerates then correspondingly velocity is added to the reference velocity of the follower and if the master decelerates the follower velocity is also decreased.

Arduino code:

```
#define trigPin 13
#define echoPin 12
long distance;

void setup()
{
  Serial.begin(9600);
  pinMode(11,OUTPUT);
  pinMode(8,OUTPUT);
  pinMode(trigPin, OUTPUT);
  pinMode(echoPin, INPUT);
}

void loop()
{
  sensor();
  if(Serial.available())
   {
     char acc1=Serial.read();
     String s="";
     s.concat(acc1);
     delay(50);
     acc1=Serial.read();
     s.concat(acc1);
     int acc=s.toInt();
     vel=100;//has been set as reference vel using wheel encoder.
     if(acc==0)
      mov(100);
     else
      vel=vel+acc*0.05;
     vel=map(vel,-10,10,0,255);
     mov(vel);
    }
  }
```

```
void sensor()
{
  long duration;
  digitalWrite(trigPin, LOW);
  delayMicroseconds(2);
  digitalWrite(trigPin, HIGH);
  delayMicroseconds(10);
  digitalWrite(trigPin, LOW);
  duration = pulseIn(echoPin, HIGH);
  distance = (duration/2) / 29.1;
  Serial.println(distance);
  Serial.flush();
 // return distance;
}

void mov(int x)
{
    analogWrite(9,x);
    analogWrite(10,x);
    digitalWrite(11,LOW);
    digitalWrite(8,LOW);
}
```

Python code:

```
# Master Follower Robot
# Implements a multi-variable linear Kalman filter.
# This code is based on a larger tutorial "Kalman Filters for
Undergrads"
# located at http://greg.czerniak.info/node/5.

import serial
from numpy import matrix
```

```
from numpy import eye as identity
from numpy import transpose
from numpy import zeros
from numpy.linalg  import inv as inverse

ser = serial.Serial("/dev/tty.usbmodem1411",9600)

# How many seconds should elapse per iteration?
timeslice = 0.02#(0.02)
# accumulated error (for integral control)
errorAcc = 0
errorOld = 0

# Implements a linear Kalman filter.
class KalmanFilterLinear:
  def __init__(self,_A, _B, _H, _x, _C, _W, _Q):
    self.A = _A         # State transition matrix.
    self.B = _B         # Control matrix.
    self.H = _H         # Observation matrix.
    self.X  = _x       # current_state_estimate: Initial state estimate.
    self.C = _C       # current_prob_estimate: Initial covariance
estimate.
    self.W = _W         # Estimated error in process.
    self.Q = _Q        # Estimated error in measurements.
    self.AT = transpose(self.A)
    self.K = identity(2)
    self.V = identity(2)

  def GetCurrentState(self):
     return self.X

  def GetCurrentProb(self):
     return self.V

  def GetCurrentGain(self):
     return self.K

  def GetCurrentVar(self):
```

```
        return self.C

    def Step(self, u, z):   # u=control_vector, z=measurement_vector

        # observation_matrix
        self.HT = transpose(self.H)

        # Prediction step
        # predicted_state_estimate X_p = AX + Bu
        X_p = self.A * self.X + self.B * u
        # print "predicted", X_p
        # predicted_prob_estimate V_p = ACA_t + W
        V_p = self.A * self.C* self.AT + self.W

        # Observation step
        IC = self.H * V_p * self.HT + self.Q
        # kalman_gain K
        K = V_p * self.HT * inverse(IC)
        self.X = X_p + K * (z - self.H * X_p)
        self.C = V_p - K * self.H * V_p
        self.K = K
        self.V = V_p

def getMeasurement(s):
        while(1):
                try:
                        a = []
                        for s in ser.readline():
                                if not s.isdigit(): break
                                a.append(s)
                        if a: return int(''.join(a))
                except:
                        continue

tt = 0.5*timeslice*timeslice

## state = [relative position, relative velocity, master acceleration]
## rel_pos = rel_pos + rel_vel*timeslice
```

```
## rel_vel = rel_vel + master_accel*timeslice -
follower_accel*timeslice
## master_accel = master_accel
state_transition = matrix([[1, timeslice, 0],[0, 1, timeslice],[0, 0, 1]])

## control = [ follower_accel ]
control_matrix = [[0],[timeslice],[0]]

## measurement = [ relative position ]
observation_matrix = [[1, 0, 0]]

# This is our guess of the initial state.
initial_state = matrix([[10.0],[1.0],[0.000]])
initial_probability = identity(3)
process_covariance = [[0.01,0,0],[0,0.01,0],[0,0,0.01]] #0.01
measurement_covariance = [[10*10]]

#Follower acceleration
follower_accel = 0.01

kf = KalmanFilterLinear(state_transition, control_matrix,
observation_matrix, initial_state, initial_probability,
process_covariance, measurement_covariance)

i = 0
while(1):

        dist = getMeasurement(ser.readline())
        #print dist
        # print dist

        control_vector = matrix([[follower_accel]])
        measurement_vector = matrix([[dist]])
        kf.Step(control_vector, measurement_vector)

        # predicted distance (relative position)
        px = kf.GetCurrentState()[0,0]
        # predicted relative velocity
```

```
pxdot = kf.GetCurrentState()[1,0]
# predicted (master) acceleration
px2dot = kf.GetCurrentState()[2,0]

# controller
kp = 3#3
kd = 0.0001
ki = 0
errorNew = dist - px
errorAcc += errorNew
follower_pos = (kp*errorNew) + (kd*((errorOld-
errorNew)/timeslice)) + (ki*errorAcc)

niter = 30#
time = timeslice * niter
# acceleration needed to catch up (dist) in 'niter' steps
a1 = (follower_pos/time/time -pxdot/time) + px2dot
niter = 5
time = timeslice * niter

# acceleration to slow down (speed)
a2 = -pxdot/time + px2dot

# equal weightage to catching up and slowing down
k1 = k2 = 1
follower_accel = (k1*a1+k2*a2)/(k1+k2)
errorOld = errorNew


control_string = str(int(pxdot))
print control_string
ser.write(control_string)
ser.write("\n")
```