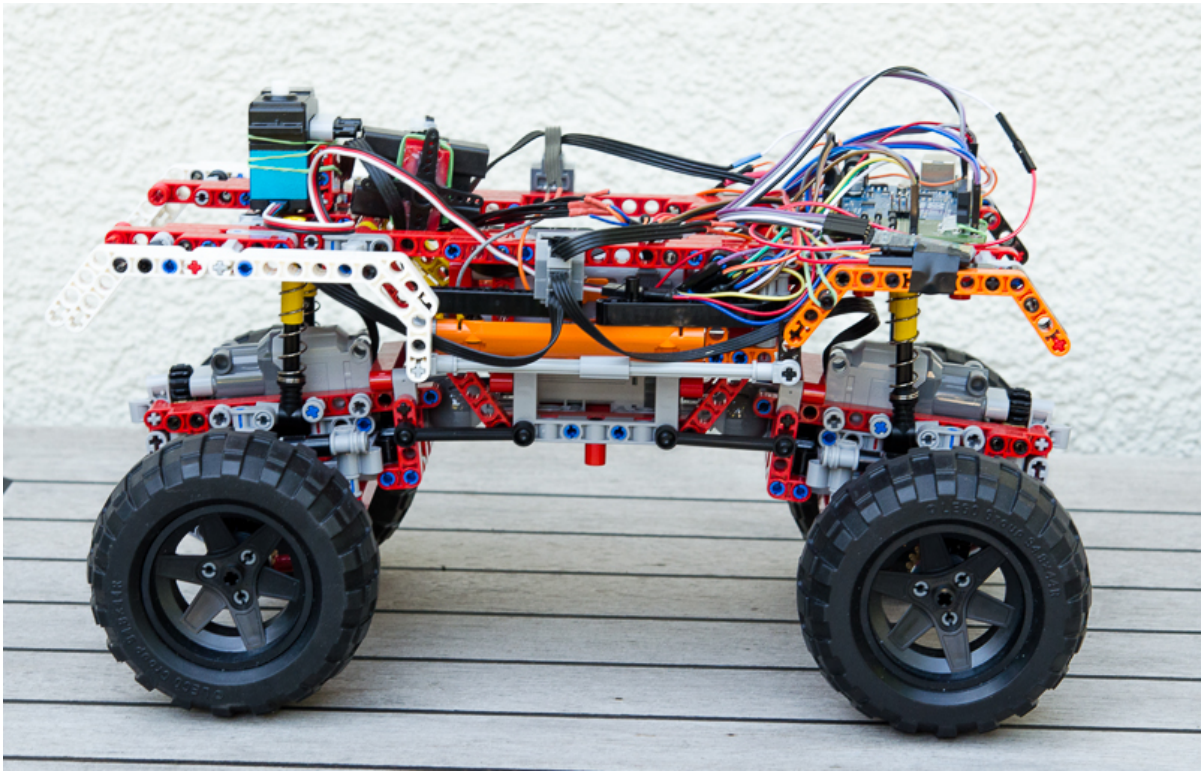
TECHNISCHE UNIVERSITÄT MÜNCHEN

FAKULTÄT FUR INFORMATIK
DER TECHNISCHEN UNIVERSITAT MÜNCHEN

HW/SW Co-Design with a LEGO car

Final Project Documentation



Team: Hannes Schulz, Sebastian Fischer, Lars Kipferl, John Chiotellis

1. Objective

The objective was to build an autonomously driving LEGO car by using an FPGA board and various sensors and actuators of our own choice. We were given the LEGO car and a budget of 100€ for hardware parts as well as some introduction into parallel computing, scheduling and other relevant topics. The design of the hardware and software was completely up to the students.
Because of that freedom, we decided to omit the original proposal of using the FPGA Board and switched to a design incorporating a Raspberry PI as a computing unit and an Arduino Uno Board (further denoted as Arduino) to communicate with sensors and actuators.

2. Overall Design

Our goal was to build a car that would be able to drive autonomously through any given environment and draw a map of that environment without bumping into obstacles. Furthermore, we wanted to add an object following behaviour to be able to follow other dynamic objects such as other LEGO cars. Due to the late arrival of hardware parts and the overall complexity of incorporating many different pieces of hardware into one working system, we had to make some adjustments to our original plans and ended up to design a simple, but extendable obstacle avoidance behaviour while driving around.

Our hardware design consists of three parts. First, we build our own “poor mans laser” sensor out of two infrared distance sensors (further denoted as IR sensor) mounted on top of a servo motor to be able to scan the environment for possible obstacles. As for the actuators, we used the original LEGO motors and servo motors to drive and steer respectively. Second, the Arduino is the single controller for the driving motors connected to a H-bridge as well as the servo motor that is responsible for steering. It also provides the values from the IR distance measure sensors. Last but not least, the Raspberry PI is connected over an USB cable to the Arduino and is responsible for the calculations regarding the driving behaviour using the values it gets from the IR sensors. As it sends the commands to control the car it constantly calculates its environment and avoids obstacles.
On the software side we decided to use the Robot Operating System (ROS) to be able to set up the cars driving logic as well as having an easy way of handling the sensor and actuator data of the whole system. Figure 1 shows the whole hardware design. The algorithm for driving autonomously is based on the potential field method for path planning and obstacle avoidance.

As for the car itself, we sticked with the original blueprints from LEGO, except that we did not use the upper chassis, since we needed space for all the cables and hardware parts (as you can see on the title page). This enables our car to drive over small, flat obstacles, which is quite

useful as we discuss later on.

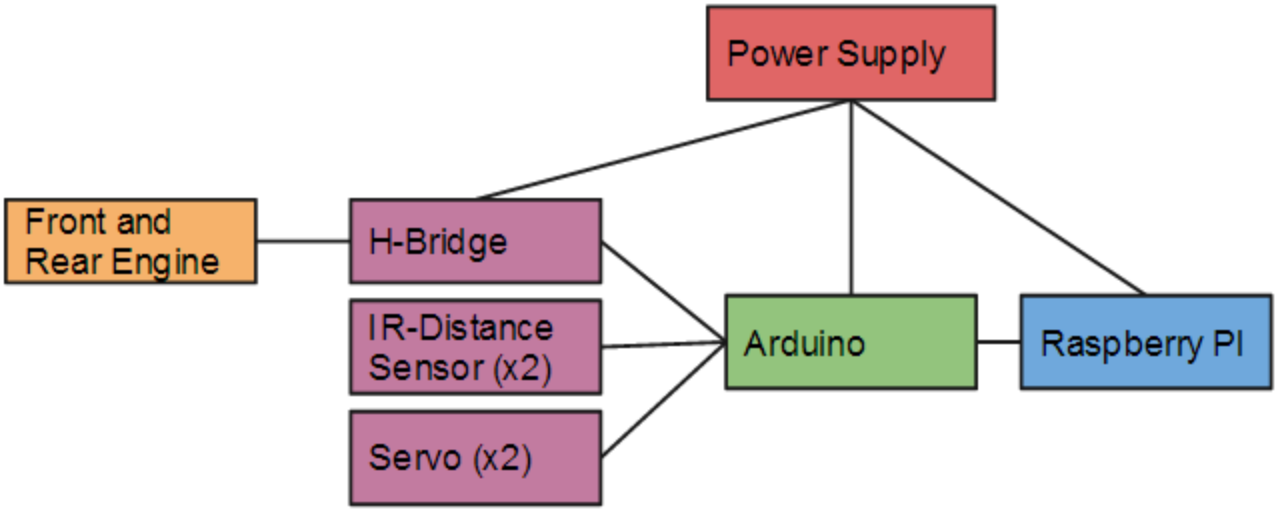


Figure 1

3. Hardware

We had two different demands on our hardware. First we needed a small but relatively powerful computer that is capable of doing the calculations for the potential field. And second a programmable microcontroller that in the best case is combined with an Analog-Digital-Converter (ADC) to interact with the IR sensors.
As the heart of our car we chose a Raspberry PI. The Raspberry PI is a very small and low cost ARM based computer, which has sufficient processing power to fulfill our needs.
Because of the IR sensors, that generate analogue values as well as the necessity to generate PWM signals to communicate with the servos and the H-Bridge we chose an Arduino microcontroller. This way we overcame the restrictions of the Raspberry PI: it is not capable of reading analogue inputs and can only produce one PWM output, but for our design we needed more. Also this platform is widely used and has a huge community, so a library for our IR sensors was already available. Another advantage of the Arduino is, that it can be programmed directly in C. Additionally, code can just be programmed over the USB port without the need for a special device for flashing the code onto the microcontroller. That makes the integration of the Arduino in our software framework easier.
As power supply, the battery pack that came with the provided LEGO car is used to power all components as well as the Raspberry PI. Therefore we needed a 5V output, but the power provided by the batteries was too much. So we built a voltage transformer according to the circuit diagram depicted in figure 2.

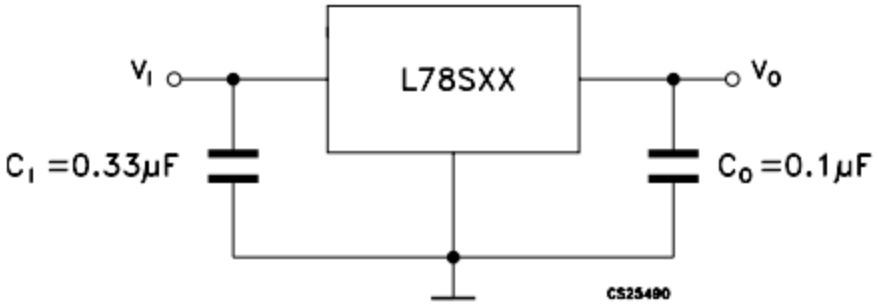


Figure 2

As a sensor to scan the environment we decided to build our own “poor mans laser” scanner consisting of a servo motor and two infrared range finders with a range of 20cm to 150cm, both mounted on top of the servo motor that rotates 180 degrees and back. The servo motor sweeps both rangefinders in steps of 10 degrees, giving us a 2 x 18 range scans of the environment to work with. The Problem with this setup is, that it scans the environment quite coarse and slow, leaving very small objects undetected and making path planning more difficult. Furthermore, we had to mount the IR sensors on top of the car, meaning that all obstacles smaller than the height of our IR sensor are a threat to our car, since they could be too big to drive over them but are not detectable by our setup even though the original car has a very solid 4 by 4 gear that is capable of driving over obstacles that are around the size of half the diameter of the wheels. All in all, it is very difficult to build a good distance sensor with limited resources at hand. But considering the fact that professional laser sensors cost several thousand euros it was the best solution for the given budget.

The complete circuit diagram with all components is depicted in figure 3. The Raspberry PI is not shown in this image, because it is connected to the Arduino over USB.

Initially it was also planned to use a webcam for object recognition and object following, but due to the lack of time and non-available Linux drivers we omitted this feature.

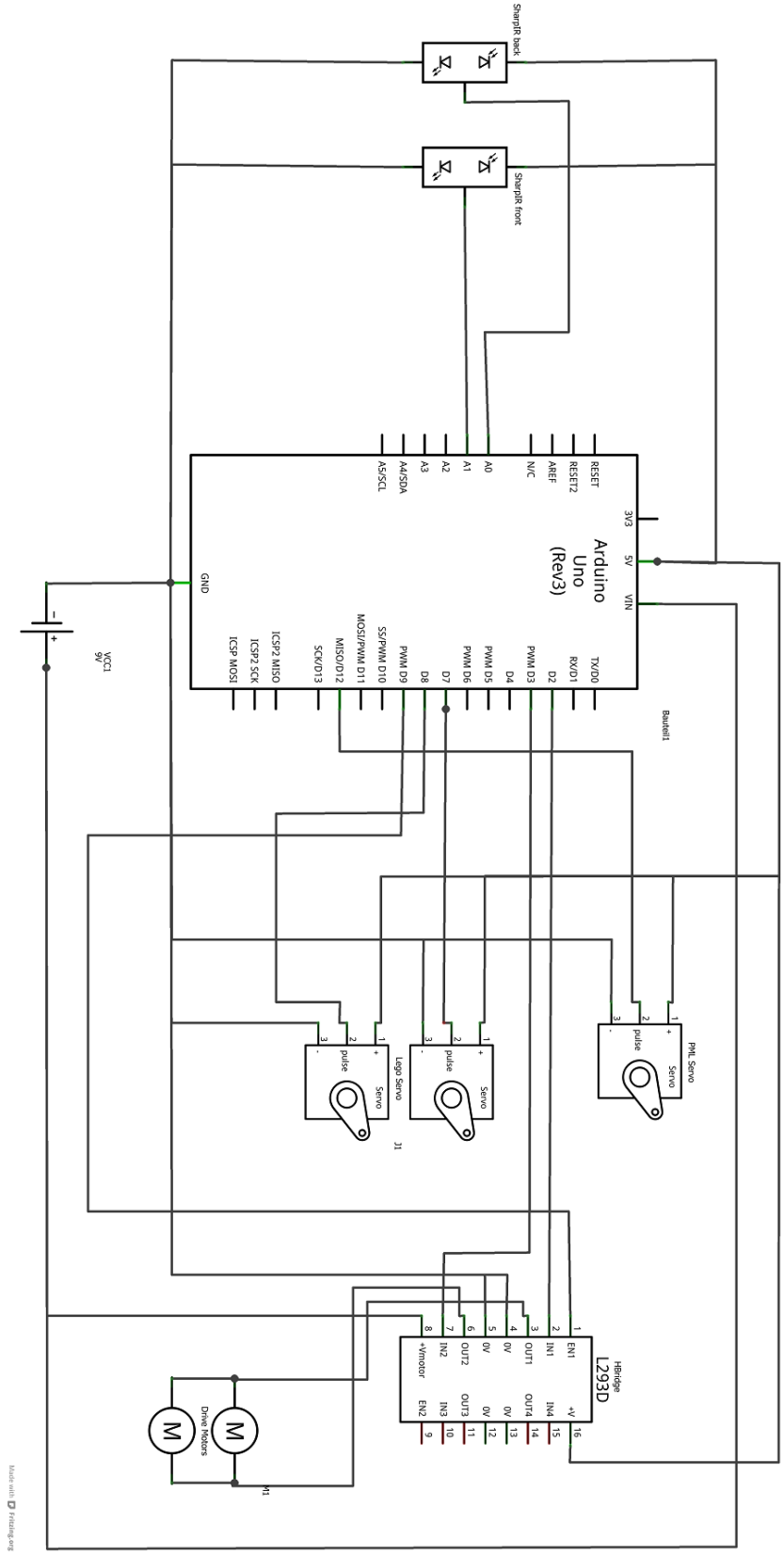

SharpIR back
SharpIR front
Arduino
Uno
(Rev3)
RESET
RESET2
AREF
N/C
A0
A1
A2
A3
A4/SDA
A5/SCL
3V3
5V
VIN
GND
TX/D0
RX/D1
PWM D3
D2
PWM D5
D4
PWM D6
D7
PWM D9
D8
SS/PWM D10
MOSI/PWM D11
MISO/D12
SCK/D13
ICSP2 MISO
ICSP2 SCK
ICSP MOSI
VCC1
9V
BauteilI1
pulse
Servo
PML Servo
Lego Servo
J1
HBridge
L293D
EN1
IN1
OUT1
0V
0V
OUT2
IN2
+Vmotor
EN2
IN3
OUT3
0V
0V
IN4
OUT4
+V
Drive Motors
M1

Figure 3

4. Software

On the Raspberry PI we installed Raspbian Wheezy which is a derivative of Debian and the Robot Operating System (ROS). ROS is a software framework for robot software development. It provides functionalities like message passing between different processes and hardware components, making communication of control commands and sensor callbacks a lot easier to handle. ROS is based on so called “nodes”, that publish or subscribe messages to or from topics. For our case we made 5 topics: radar_angle, radar_front, radar_back, directions_drive and directions_steer. The radar topics contain the values provided from the IR sensors and the angle in which they are pointing. The directions topics contain the desired action of the car, which is if the car should move for- or backward and if it should steer in any direction.
That makes the whole software design very transparent, since it is possible to see what is going on in the system, using the various visualization tools provided by the ROS framework.
There is also a library available - called rosserial - that allows us to integrate the Arduino over a serial connection (USB) into the system, as it takes care of the communication. Rosserial is a protocol we have to run a separate node on the ROS system that communicates with the Arduino. This node is written in python and is shipped with the rosserial library. Besides the integration of the Arduino, ROS provides us with a series of predefined functions and libraries that make it a bit easier to implement complex algorithms.

For the Arduino a few libraries are available. To read the values from the Sharp IR distance sensors we use an existing library that returns us the readings from the sensors in centimeters. The Servo that rotates the distance sensors are controlled by the standard libraries that come with the Arduino IDE. The Lego servo motor from the powerfunction set is not controllable with this standard library because it runs on a custom PWM signal. Therefore we had to write our own library that depends on the Arduinos clock registers. Due to lack of documentation we had to try different frequencies to determine the right one. We did this with a simple for-loop that runs through the different values. Once we found the right frequency we still had to try how the servo reacts to different inputs. The rest of the Arduino programming is very straightforward self explaining together with the comments in our code.

So all in all, the software setup of our projects consists of three ROS nodes, one representing the Arduino, which provides the data of our IR sensor and receives commands to drive and steer the car. This node publishes the current angle of the servo motor, the distance measurement of both IR sensors and receives two values, one for the motor and the other one for the servo for steering. These two values are provided by the node running on the Raspberry PI, which takes the IR sensor data to calculate where to drive based on the implemented logic, which is discussed in the algorithm section. To enable communicating between the Raspberry Pi and the Arduino the rosserial node is needed, that uses the USB connection to transfer data from one processor to the other. See figure 4 for the node setup.

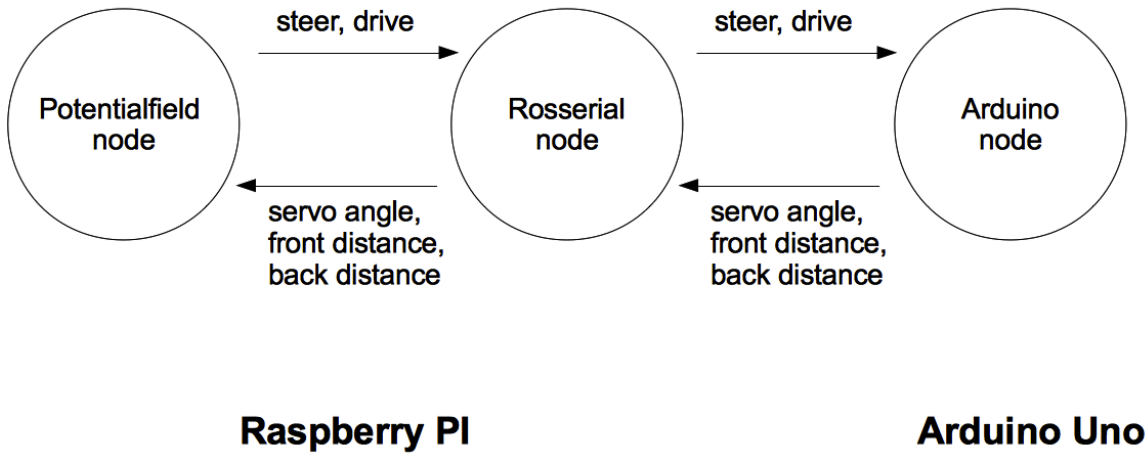


Figure 4

5. Algorithm

Our algorithm for autonomous driving is based on the potential field method, which is a combination of path planning and obstacle avoidance. Due to the bad sensor information we omitted some of the original features of the potential field method.

The potential field method basically guides the robot safely from one point in space to another by treading the environment where the robot moves as a differentiable function and the robot as a point in this space, thus converting the problem of moving the robot into a gradient descent problem. This is done by building a virtual force that drives the robot towards its goal and away from any obstacles. The virtual force consists of an attractive force, which is the distance between the current position of the robot and its goal. Since we have not implemented any kind of self-localization or mapping it is difficult to define a goal in space. That is why we defined our attractive force as a point directly in front of the robot, meaning the attractive force is pushing the robot straight forward all the time:

$$F_{att}(q) = (1|0) \tag{1}$$

The second part of the virtual force is the repulsive force, which originates from any obstacle scanned by our IR sensor while driving. The repulsive force is calculated by the derivative of a repulsive field equation that becomes weaker the further the robot is away from an obstacle, meaning that the distance to an obstacle has a direct influence on the strength of the repulsive force:

$$F_{rep}(q) = k_{rep}\left(\frac{1}{p(q)} - \frac{1}{p_0}\right)\frac{1}{p^2(q)}\frac{q - q_{obstacle}}{p(q)} \quad if\ p(q) \leq p_0 \tag{2}$$

where k_{rep} is a scaling factor, $p(q)$ is the distance to the obstacle, p_0 is the maximum distance of influence of objects, q is the position of the robot in space (cartesian coordinates) and $q_{obstacle}$ is the position of the obstacle in space (also cartesian coordinates). Since we have no map, we only see our environment locally and therefore the robots position is always (0|0).
Since there could be more than one obstacle influencing the virtual force driving the robot, we decided to only use the closest obstacle to form the repulsive force to make it easier to decide where to drive.
The virtual force is the sum of the attractive and the repulsive force and should guide the robot safely through the environment:

$$F(q) = F_{att}(q) + F_{rep}(q) \tag{3}$$

A major problem for the potential field method are local minima. As the robot approaches concave structures, such as a corner in room, the robot gets several equidistant measurements from the range sensors, leaving the system trapped, because it will oscillate between different steer movements leaving the system trapped. To avoid this kind of behaviour, we implemented a backward movement, that If the robot approaches an obstacle and comes too close, it drives backwards and steers in the opposite direction, in the hope of getting out of the local minimum.

Another problem in our implementation is the way we handled the attractive force. Since we just want the robot to drive straight forward, the attractive force is always the same, which makes it more difficult to find a feasible set of parameters for the algorithm that works fine with the actual system. Furthermore, that also makes room for problems with small obstacles that lie directly in front of the robot. Since the robot is just pushed forward, a small obstacle in front of the robot that is measured only by the range scan that points in front of the robot would result in oscillating back and forth movement, since no force would push the robot sideways. This behaviour should also be avoided by the above described backward movement.

Another feature we added was that we do not look at obstacles that are over 40° or under -40° to the left and right sight respectively, because those obstacles do not threaten the robot while it is driving forward.

Also, to get a full scan of the environment our laser sensor needs several seconds, making real-time calculations of the repulsive forces somewhat difficult. Therefore our algorithm waits a full scan before driving and then drives for the period of one full scan, repeating that behaviour over and over again. This ensures a more robust obstacle avoidance, since we give our sensors enough time to collect the data we need for our calculations.

Despite the relative simplicity of the potential field method, it is quite difficult to find a good set of parameters that works well on the actual hardware system with all possible state space configurations the robot could encounter while driving in a real world environment. Even if our system does fail to avoid obstacles in some cases, it can manage quite a few situations without bumping into anything, making the drive logic of our system quite good.

6. Problems during the project

We lost a lot of time because of the missing documentation for the Lego powerfunction servo motor that runs at a custom PWM signal which is different from standard servo motors and therefore could not be used with the Arduinos standard libraries. We had no oszilloskop and even if we had one it would have taken us as much time to figure out how to use it as it took us to find the frequency by try-and-error.
Separating the work was also a hard task. Because we only had one car to test our functions on we had to coordinate numerous meetings and even then sometimes everybody wanted to test something on the car at the same time.
Unfortunately, the rosserial is still in an experimental state for architectures like the Raspberry PI. Despite trying several ROS distributions and hacking into every file, we were unable to get the rosserial node working on the Raspberry PI, so that we could not use it for presentation purposes. But since the rosserial package is still worked on actively, there should be fully working versions for the Raspberry PI in the near future. Due to the late delivery of the Raspberry PI, we encountered this problem far too late and were not able to write our own communication protocol. As we got it, we already centered our project around ROS and it would have been too much trouble to change this.

7. Conclusion

During our work on this project we had a few problems to solve. The issue with the powerfunctions servo for example forced us to engage in hardware level programming. But we learned a lot while we experiment with timers and hardware interrupts.
The, unfortunately unsolved, problem of rosserial showed us how complicated the integration of different systems. Future work could deal with this problem. Perhaps it is possible to connect the PI and the Arduino with I2C or SPI.
Besides this we have built a working car that is capable of recognizing and avoiding obstacles. As mentioned above there is also space for improvements and future development, for example regarding the algorithm. Improved sensor integration could result in the car being able to drive continuously without having to stop and wait for new data.

It is also possible to change the layout of the deployed sensors. More sensors could be placed in different heights or angles on the car or a camera could scan the whole environment around the car. The camera could also follow an object or a beacon.
In the end our car was able to drive autonomously using the potential field algorithm. The behavior of the car could be refined by mathematically estimating the optimal parameters for the algorithm (like the scaling factor). Furthermore, one could use/implement another communication protocol than rosserial and thus improve the data transmission speed and consequently the behavior of the car (detect obstacles faster). The whole project could be extended in many ways. One idea would be to create a map of the environment using the information about the obstacles positions from the infrared sensors (possibly SLAM). The map could then be transmitted to the cars of the other teams. These could explore the world, collect different kinds of data - depending on their sensors - and implement other algorithms (like object recognition) while following collision-free paths.

8. Appendix

Code running on the Arduino in C++:

```
/*********************************************
 * Arduino code that handles communication with
 * all sensors, servos and motors.
 *
 * TUM - SS 2013 - Hardware/Software Co-Design
 * Lab Course (Supervisor: Kai Huang)
 *
 * AUTHORS: Sebastian Fischer, Hannes Schulz,
 *             Lars Kipferl, John Chiotellis
 *********************************************/


#include <ros.h>
#include <std_msgs/String.h>
#include <std_msgs/Int32.h>
#include <ServoLego.h>
#include <Servo.h>
#include <SharpIR.h>

// ros related variables
// nodehandle takes care of all the sending and receiving
ros::NodeHandle nh;
// we are sending integer messages
std_msgs::Int32 angle_msg;
std_msgs::Int32 frontDist_msg;
std_msgs::Int32 backDist_msg;
// define publishers that publish the distance values with the topic // "radar"
ros::Publisher angle_pub("radar_angle", &angle_msg);
ros::Publisher frontDist_pub("radar_front", &frontDist_msg);
ros::Publisher backDist_pub("radar_back", &backDist_msg);

Servo pmlServo;              // Servo for our "laser radar"
ServoLego steerServo;        // Lego Servo for steering
const int motorIN1 = 2;      // H-bridge leg 1 (pin IN1, dir1 PinA)
const int motorIN2 = 3;      // H-bridge leg 2 (pin IN2, dir2 PinA)
const int speedMotorA = 9;           //  H-bridge EN_A
int angle;   // Fetch data from "laser radar" for ros_msg to publish
int frontDist;
int backDist;

// Create an instance of SharpIR, constructor is SharpIR(type,
// analog_pin)
SharpIR backIR = SharpIR(GP2Y0A02YK,0); // back
SharpIR frontIR = SharpIR(GP2Y0A02YK,1); // front

/*********************************************
 * this method sets the steering.
 * 0 - no steering
```

```
 * 1 - left
 * 2 - right
 ************************************************/
void steer(int i) {
  if(i == 1)
       steerServo.left();
  else if(i == 2)
       steerServo.right();
  else
       steerServo.none();
}

/*****************************************************************
 * set the direction of the motor movement over the H-bridge
 * 0 - no driving
 * 1 - forward
 * 2 - backward
 *****************************************************************/
void drive(int i) {
  if(i == 2) {
       digitalWrite(motorIN1, LOW);
       digitalWrite(motorIN2, HIGH);
  }
  else if(i == 1) {
       digitalWrite(motorIN1, HIGH);
       digitalWrite(motorIN2, LOW);
  }
  else {
       digitalWrite(motorIN1, LOW);
       digitalWrite(motorIN2, LOW);
  }
}

/*********************************************************
 * callback functions for the messages we have subscribed
 * this method is called from nh.spinOnce()
 *********************************************************/
// callback functions for subscribers
int drive_value;
void driveCallback(const std_msgs::Int32& drive_msg) {
 drive_value = drive_msg.data;
 drive(drive_value);
}

int steer_value;
void steerCallback(const std_msgs::Int32& steer_msg) {
  steer_value = steer_msg.data;
  steer(steer_value);
}

// now we can instantiate our subscribers
ros::Subscriber<std_msgs::Int32> drive_sub("directions_drive", &driveCallback);
ros::Subscriber<std_msgs::Int32> steer_sub("directions_steer", &steerCallback);
```

```
void setup(){
  //Serial.begin(57600);
  steerServo.attach(7,8);
  pmlServo.attach(12);
  // Motor A
  pinMode(motorIN1, OUTPUT);
  pinMode(motorIN2, OUTPUT);
  pinMode(speedMotorA, OUTPUT);
  // set motor to full speed
  digitalWrite(speedMotorA, HIGH);
  // initialise the nodehandler and
  nh.initNode();
  // advertise our messages
  nh.advertise(angle_pub);
  nh.advertise(frontDist_pub);
  nh.advertise(backDist_pub);
  // subscribe to directions topic
  nh.subscribe(drive_sub);
  nh.subscribe(steer_sub);
}

/************************************************************
 * This method is used to get the next step of our radar.
 * The Radar turns in steps of 10 degrees.
 * If 180 degrees are reached we jump back to 0 degrees.
 ************************************************************/
// store the current position
int currentPos = 0;

void sensorNext(){
  if( currentPos < 180 )  {
       currentPos+=10;
  }
  else {
       currentPos = 0;
  }
  pmlServo.write(currentPos);

  // prepare data and write to message
  angle = currentPos;
  frontDist = frontIR.getData();
  backDist = backIR.getData();
}


void loop(){
  // let our radar trun
  sensorNext();
  // get message for publisher
  angle_msg.data = angle;
  frontDist_msg.data = frontDist;
  backDist_msg.data = backDist;
  //Serial.println(angle_msg.data);
```

```
  Serial.println(frontDist_msg.data);
  //Serial.println(backDist_msg.data);
  // now publish the data
  angle_pub.publish(&angle_msg);
  frontDist_pub.publish(&frontDist_msg);
  backDist_pub.publish(&backDist_msg);
  // this handles all the communication callbacks
  nh.spinOnce();
  // we hae nothing to do for the subscriber because
  // the defined callback is called from the nh.spinOnce()
  //wait 200ms
  delay(200);
}
```

Self written LegoServo header file

```
#ifndef ServoLego_h
#define ServoLego_h

#include <inttypes.h>

// TICK_SIZE is the time between interrupts. This is essentially the // resolution
// of the PWM we're able to create. This is also an average value,
// because other
// interrupts will affect the frequency of ours.
#define PWM_TICK_SIZE 50 // uS
// PWM_PERIOD is the period of the PWM we want to have
#define PWM_PERIOD   13000 // uS (20ms)
// This calculation is used to determine the 'reset' number for the
// counter
#define PWM_TICK_PERIOD (int) (PWM_PERIOD / PWM_TICK_SIZE)

class ServoLego
{
public:
  ServoLego();
  void attach(int pin1, int pin2);  // attach the given pin to the
                                                    // next free channel, sets
                                                    // pinMode, returns channel
                                                    // number or 0 if failure
  void left();
  void right();
  void none();
};

#endif


Corresponding C file that implements the ServoLego class

#include <avr/interrupt.h>
#include <avr/io.h>
```

```
#include <Arduino.h>
#include <ServoLego.h>

// Interrupt variables have to be defined as 'volatile' so that the
// CPU always
// fetches them from RAM and doesn't get an incorrect value from cache
volatile unsigned int timer2_counter = 0;
volatile unsigned int timer2_trigger_low = 500;
int level = 0;
int PWM_PIN1;
int PWM_PIN2;
// Timer2 overflow interrupt vector handler
// This is called every time timer 2 overflows (around every 50ms)
// This routine needs to be as small and short as possible so that we
// don't
// impact the rest of the Arduino by executing lots of code in the background
ISR(TIMER2_OVF_vect) {
  timer2_counter++;
  if(timer2_counter == timer2_trigger_low) {
    level = 0;
  } else if(timer2_counter >= PWM_TICK_PERIOD) {
    timer2_counter = 0;
    level = 1;
  }
  // Reset the timer2 counter so that it can count and overflow once
  // again
  TCNT2 = 0;
}

ServoLego::ServoLego(){
       TIMSK2 = 1<<TOIE2;  // Timer 2 overflow interrupt enable
       TCNT2 = 0;
}

void ServoLego::attach(int pin1, int pin2){

       PWM_PIN1 = pin1;
       PWM_PIN2 = pin2;
       pinMode(PWM_PIN1, OUTPUT);
       pinMode(PWM_PIN2, OUTPUT);
}

void ServoLego::left(){

       digitalWrite(PWM_PIN1, level);
    digitalWrite(PWM_PIN2, LOW);
}

void ServoLego::right(){

       digitalWrite(PWM_PIN2, level);
    digitalWrite(PWM_PIN1, LOW);
}
```

```
void ServoLego::none(){

       digitalWrite(PWM_PIN1, LOW);
    digitalWrite(PWM_PIN2, LOW);
}
```

Code running on the Raspberry PI (obstacle avoidance):

```
/*
  Code for autonomous driving of LEGO Car. This Piece of code implements
  a basic obstacle avoidance behaviour while trying to drive forward.

  TUM - SS 2013 - Hardware/Software Co-Design Lab Course
  (Supervisor: Kai Huang)

  AUTHORS: Lars Kipferl, Hannes Schulz, Sebastian Fischer, John Chiotellis

*/

#include "ros/ros.h"
#include "std_msgs/Int32.h"
#include "math.h"
#include "float.h"

/// GLOBAL VARIABLES ///

int laser_angles[19] = {0};  // variables to store and manage
                                        // laser data
int laser_frontDist[19] = {320};
int laser_backDist[19] = {320};
int arraySize = 19;
int angle_index = 0;
int front_index = 0;
int back_index = 0;

double attForce[2];   // variable for attracting force
double repForce[2];   // variable for repulsive force
double resForce[2];          // variable for resulting force

bool drive_mode = false;     // switches between scanning and driving

int drive = 0;        // Integers for sending drive commands to arduino
int steer = 0;

double maxInf = 80.0;        // maximum distance (values over 80cm are
                                // ommited)

int scalFact = 27000; // scaling factor for calculating forces

double steerToRight = -0.01;         // parameter which decides when to
                                        // steer right
```

```
double steerToLeft = 0.01;   // parameter which decides when to steer
                                    // to left

// Calculates the attractive force (currently set to fixed value)
void calcAttForce() {
  attForce[0] = 1.0;
  attForce[1] = 0.0;
}

// Calculates the repulsive force by using laser_data
void calcRepForce() {
  bool go_on = false;        // bool to decide wether other
                                    // calculations are necessary

  repForce[0] = 0;
  repForce[1] = 0;
  // Drive forward if no obstacles are found in a 40 to -40 degree
  // radius

  for(int i=0;i<arraySize;i++) {
    if(laser_angles[i] > -40 && laser_angles[i] < 40) {
      if(laser_frontDist[i] < maxInf) { // found obstacle? ...
        go_on = true;        // ... then we need to calculate the
                                    // repulsive force of it

        break;
      }
    }
  }
  if(go_on) {         // obstacle found in 40 to -40 degree radius? ->
                      // calculate repForce

    // Search for nearest obstacle ...
    double minDist = DBL_MAX;
    int angle_ind;
    for(int i=4;i<arraySize-4;i++) { // ... in 40 to -40 radius
      if(laser_frontDist[i] < minDist) {
        minDist = laser_frontDist[i];
        angle_ind = i;
      }
    }
    // now calculate the repulsive force of the obstacle
    double rad = laser_angles[angle_ind]*(M_PI/180); // need rad for
                                                                    // cos() and
                                                                    // sin()
                                                                    // function
    double x = minDist*cos(rad);    // cartesian coordinates of the
                                              // obstacle

    double y = minDist*sin(rad);
    if(minDist < maxInf && minDist != 0) {
      repForce[0] = scalFact*((1/minDist)-(1/maxInf))*(1/powf(minDist,2))*(-x/minDist);
      repForce[1] = scalFact*((1/minDist)-(1/maxInf))*(1/powf(minDist,2))*(-y/minDist);
    }
```

```
  }
}

// Calculates the resulting force by adding all attracting and
// repulsive forces
void calcResForce() {
  resForce[0] = attForce[0] + repForce[0];
  resForce[1] = attForce[1] + repForce[1];
}

// Decide which motor controls to send
void discretize() {
  if(resForce[1] <= steerToRight) { // if certain treshold is
                                                // surpassed, steer right ...
    steer = 2;
  } else if(resForce[1] > steerToLeft) {    // ... same for the left
                                                          // side ...
    steer = 1;
  } else { // ... or just drive straight when nothing is happening
    steer = 0;
  }
  if(resForce[0] <= 0.0) { // if obstacles are too near, drive
                                      // backwards ...
    drive = 2;
    switch(steer) { // ... and steer in the opposite direction
      case(2): steer = 1;
      break;
      case(1): steer = 2;
      break;
      case(0): steer = 2;
    }
  } else {
    drive = 1; // if the path ahead is clear, drive forward
  }
  ROS_INFO("steer: %d  drive: %d", steer, drive); // outputs on
                                                                        // console which
                                                                        // driving commands
                                                                        // are sent
}

// Callback for servo motor (angle) subscriber
void laserAngleCallback(const std_msgs::Int32::ConstPtr& laserAngle_msg)
{
  // Fill laser data with one full scan, then begin overwriting old
  // data
  if(angle_index >= 19) {
    angle_index = 0;
    drive_mode = !drive_mode; // alternate between driving and
                                        // scanning for the duration of one
                                        // scan each
  }
  laser_angles[angle_index] = laserAngle_msg->data;
  laser_angles[angle_index] -= 90; // angles should be represented in
                                                // right-hand-coordinate system
```

```
  angle_index++;
}

// Callback for front IR subscriber
void laserFrontDistCallback(const std_msgs::Int32::ConstPtr& laserFrontDist_msg)
{
  if(front_index >= 19) {
    front_index = 0;
  }
  laser_frontDist[front_index] = laserFrontDist_msg->data;
  front_index++;
}

// Callback for back IR subscriber
void laserBackDistCallback(const std_msgs::Int32::ConstPtr& laserBackDist_msg)
{
  if(back_index >= 19) {
    back_index = 0;
  }
  laser_backDist[back_index] = laserBackDist_msg->data;
  back_index++;
}

int main(int argc, char **argv)
{
  // initialize node
  ros::init(argc, argv, "potentialfield");

  // nodehandler keeps track of wether the node is still running or
  // not
  ros::NodeHandle n;

  // set up subscriber to get data from arduino
  ros::Subscriber angle_sub = n.subscribe("radar_angle", 1, laserAngleCallback);
  ros::Subscriber frontDist_sub = n.subscribe("radar_front", 1, laserFrontDistCallback);
  ros::Subscriber backDist_sub = n.subscribe("radar_back", 1, laserBackDistCallback);

  // set up publisher to send data to arduino
  ros::Publisher drive_pub = n.advertise<std_msgs::Int32>("directions_drive", 1);
  ros::Publisher steer_pub = n.advertise<std_msgs::Int32>("directions_steer", 1);
  // loop_rate determines how often processes are called in hertz
  ros::Rate loop_rate(10);

  int count = 0; // variable to count loop iterations (used
                      // internally by ros)
  while(ros::ok())
  {
    // messages to be published by publisher
    std_msgs::Int32 drive_msg;
    std_msgs::Int32 steer_msg;

    // scan if not driving, drive when not scanning
    if(!drive_mode) {
      calcAttForce();
```

```
      calcRepForce();
      calcResForce();
      discretize();
      drive_msg.data = 0; // sets the message data
    } else {
      drive_msg.data = drive;
    }
    steer_msg.data = steer;

    // publish messages to topic
    drive_pub.publish(drive_msg);
    steer_pub.publish(steer_msg);

    // Pump Callbacks
    ros::spinOnce();

    // Wait (Depends on loop_rate)
    loop_rate.sleep();
    ++count;
  }
  return 0;
}
```