# IAR Task 1 Report

Angus Pearson – s1311631
Jevgenij Zubovskij – s1346981

October 18, 2016

## Abstract

We have implemented a control program for the Khepera robot, utilising eight IR distance sensors to reactively avoid colliding with obstacles, follow the perimeter of an object or wall and explore the environment. Multiple avenues of control were investigated and compared. The notion of the robot becoming bored affords interesting behaviours and an enhanced ability to explore an environment. The prominent limitations are the robot's sensing suite, with a small number of sensors and artefacts arising from the physics of IR sensing.

## 1 Introduction

The first assignment entails utilising infra-red distance sensors on the Khepera robot to navigate autonomously around an environment, "without hitting obstacles or getting stuck in corners or dead-ends. Second, the Robot should tend to follow long walls, keeping a consistent distance away from the wall." The Robot is controlled remotely by a computer over serial. We have elected to use Python as the implementation language for this practical.

As per the Task description, we split the development into two goals, one being obstacle avoidance and the other being a wall-following behaviour. Our approach is influenced by the demonstrable abilities of reaction-based control in robotics and the BUG algorithms [1].

## 2 Exploring Methods of Not Hitting Things

### 2.1 Experimenting with PID

An 8-dimensional PID controller was implemented over the sensor data, yielding an error with respect to the distance from an obstacle per sensor – Too far, the error motivates a movement towards the object; Too close and the error motivates a move away.

The approach had some promising behaviours: The robot was able to avoid hitting an object. However, wall following would either oscillate towards the wall, bumping against it or depart after following for a short distance. This is due in part to the narrow field of view afforded by solely using the forward-facing sensor pair.

However the controller was very sensitive to it's tuned gains $K_p$, $K_i$ & $K_d$ and tended to perform best when $K_p$ was much larger than the other gains. The control is neither continuous

nor low-latency, as we used discrete thresholds to react to the error vector PID produced. Thus, we discard PID in favour of a simpler reactive solution.

### 2.2 Force-Vector Control

We consider treating each distance sensor reading as a force vector pushing the robot along the axis of it's sensor. Summing these forces gives us an 'Intended Direction' resultant vector, which is used to correct the initial direction vector away from the obstacle.

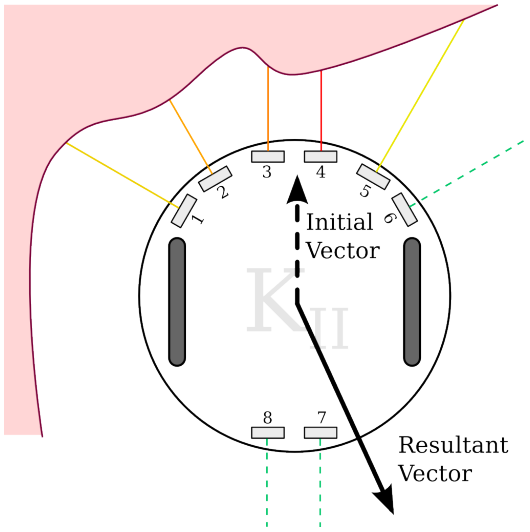


Figure 1: Showing the derivation of Force Vector from distance readings to the obstacle; A shorter distance to the obstacle yields a stronger push in the opposite direction. Distances over a threshold (shown dashed) are ignored and have magnitude 0.

Experimentation showed that the lack of calibration between sensors and direct mapping to forces introduced a 'wobble', with no exhibition of wall-following as the error vectors push in direct opposition to the wall, resulting in the robot basically bouncing off. Attempts to smooth or condition the sensor signals to prevent wobbling gave the robot poor performance in any environment with sharp angles.

The method is somewhat more elegant than PID & rule based control insofar as the complexity of behaviour that emerges from a simple rule set. Nonetheless, we discard Force-Vector control in favour of Reaction-Based control using some methods learned from the PID Experiment.

## 3 Developing Object-Following Behaviour

It was decided to build upon threshold based avoidance control. However, that would require an algorithm that accounts for the robot not being intrinsically able to keep a *at a constant distance* from a wall.

Thus, final obstacle-avoidance method reactionally adjusts our trajectory away from a collision using a composition of multiple distance sensors, using different groups to decide first whether an obstacle is too close, then deciding which way to turn using the sensors to choose a direction that presents a larger open-space to move into. Discrete thresholds are used to make decisions

about closeness. When we have made a decision to rotate left or right we stick with it until the forward direction is clear; this prevents an indecisive oscillation and the robot becoming stuck. We make no distinction between a wall and another shape.

### 3.1 Sensor Use

Different scaling is applied to sensors on the side of the robot than the front, as we can afford to be closer to an obstacle when travelling parallel to it rather than when perpendicularly approaching (Given the robot always moves forwards). Please see the code for more detailed implementation explanation. The outline of sensor use in the control loop is as follows:

1. Sensors (0, 1, 2, 3, 4, 5) used to determine if robot can no longer proceed in the forward direction and which way we have more space to move and, hence, should unstuck towards.
2. Sensors (1, 2, 3, 4) to detect followable shape and what side of the robot (left / right) to follow it on.
3. Sensors (0, 5) used to ensure we follow the shape within a consitent range of values
4. The back sensors (6, 7) were not used due to our robot only moving in the forward direction and 6 sensors allowing better control than 2

### 3.2 Boredom, a useful concept

The notion that the robot may become 'bored' with a wall that it is pursuing gives rise to some useful behaviours: An intrinsic property of becoming bored with a wall is that we will not forever circle an obstacle thinking it is just a really long wall. This also gives the robot a much higher chance of fully exploring it's environment given enough time, as random changes in course will continuously occur.

We implemented a boredom algorithm, counting how long a wall has been followed for without interruption (being stuck).

## 4 Algorithm

Every 20 ms a control loop iteration starts by taking new IR measurements. The 20 ms delay is used in order for the IR sensors to update between iterations. The IR data is then used determine further actions in descending order of priority:

1. If the robot is "bored" it begins a set of cycles to turn away from the followed shape
2. If robot is finished turning away from "boring" wall, it drives forward until it becomes stuck when the loop resumes from 3.
3. If the robot is stuck and cannot continue moving in the frontal direction, then it turns in the direction away from the previously followed shape or towards where the sensors detected more space until it detects it is no longer stuck.

4. If the robot encounters (or was previously following) a shape it can follow, it follows it on either left or right side (depending to which one is closer) at a consistent distance perpendicular to the shape's edge.

5. If robot has just initialized or has nothing else to do, it drives forwards until 3 or 4 occur.

The following is a representation of the algorithm in a state diagram form:

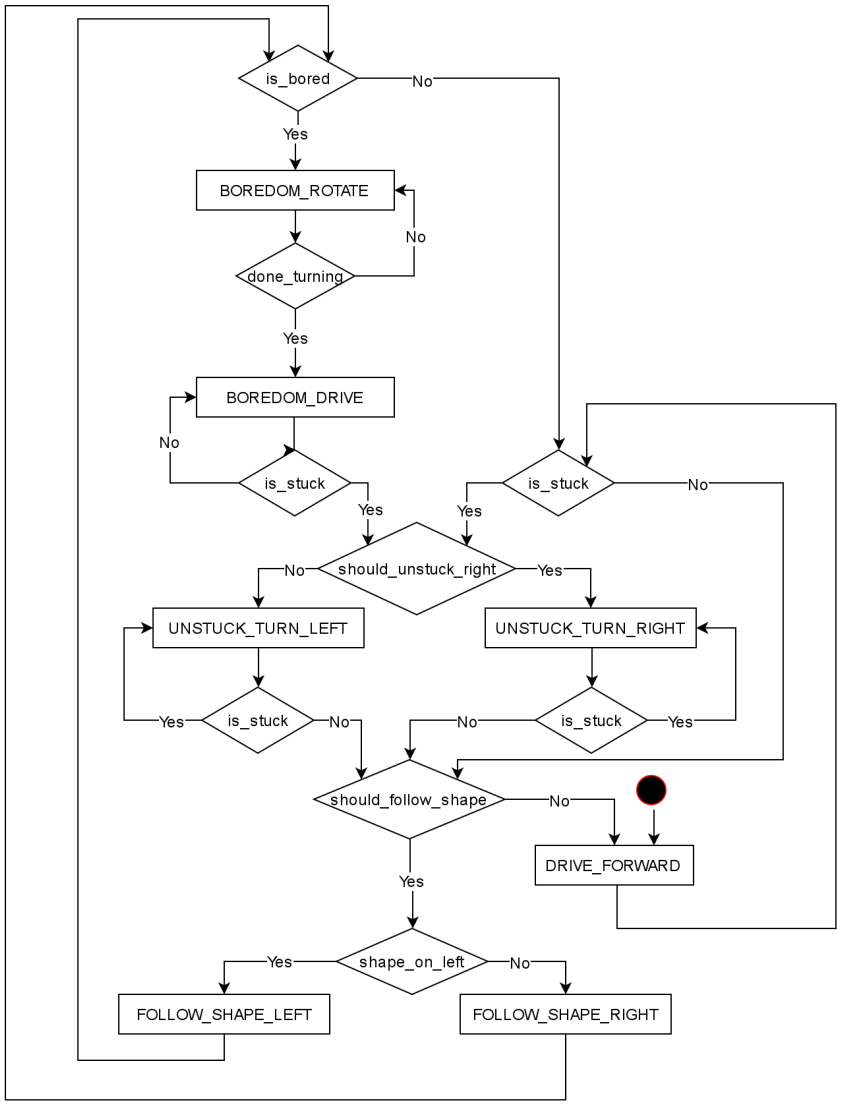


Figure 2: Illustration of the algorithm in a state diagram form.

*Source Code is included in the Appendix.*

# 5 Results

The robot is able to follow any shape it encounters at a close distances and avoid getting stuck among obstacles, dead ends and mazes. Moreover, it exhibits exploratory behavior due to the boredom concept implemented. However, it approached the outlines of darker objects noticeably closer than that of lighter ones even after several rounds of calibrating the thresholds.

# 6 Discussion & Possible Improvements

The chosen method and algorithm proved successful. The system works as per requirements and has an emergent, seemingly intelligent behavior by navigating and exploring the environment. However, physical capabilities were not fully utilized as back sensors are not used and movement is only in the forward direction.

Darker objects absorb more light, reaching the threshold IR value at a considerably closer distance than lighter ones. The algortihm does not discriminate, meaning darker objects are followed at a closer distance than lighter ones as same threshold value is used. Thus, a method allowing differentiation would allow equal distances to be held from all objects regardless of their absorbant properties.

# 7 Physical Limitations and Possible Improvements

The Khepera's distance sensors are spaced to prioritise forward motion, which is beneficial for exploring a space though does make reversing out of a dead-end much harder than turning on the spot then driving out forwards. Evenly spaced sensors in all directions would allow better estimation of object positions.

Moreover, a bump (or other touch) sensor would allow easier detection of collisions and obstacles in case the IR sensor based algorithms fails to detect or predict an object.

# Appendix

## .1 Code Listings

../../main.py

```
#!/usr/bin/env python

#
#     - - - -  I A R  - - - -
#
#  s1311631         Angus Pearson
#  s1346981    Jevgenij Zubovskij
#

from __future__     import print_function
from comms      import Comms

from odometry_algorithm   import Odometry_Algorithm
from odometry_state     import Odometry_State
```

```
from navigation_state     import Navigation_State
from navigation_algorithm   import Navigation_Algorithm

from bug_algorithm    import Bug_Algorithm
from bug_state      import Bug_State



import constants
import sys
import getopt      # CLI Option Parsing
import whiptail    # Simplest kinda-GUI thing
import time
import math
import matplotlib.pyplot as plt
from data import DataStore

namebadge = " -- IAR C&C -- "
helptext = str(sys.argv[0]) + ' -p <serial port> -b <baud rate> -t <timeout> -s <server hostname>
    '

wt = whiptail.Whiptail(title=namebadge)



def main():


    try:
        #flashy to see if robot works
        comms.blinkyblink()

        odo = Odometry_Algorithm()
        odo_state = Odometry_State()
  nav_state = Navigation_State()
  nav = Navigation_Algorithm()
  bug = Bug_Algorithm()
  bug_state = Bug_State()


        # varaibles to not resend speeds during wall following
        speed_l = 0
        speed_r = 0

        # reset odometry for this robot run
        comms.reset_odo()


  #begin control loop
        while True:


      odo_state = odo.new_state(odo_state, comms.get_odo())
        nav_state.dist = comms.get_ir()

      #check reactive first, then bug
      nav_state = nav.new_state(nav_state, odo_state, bug_state)
      #if have free movement, use the bug algorithm
      if bug_state.algorithm_activated and bug_state.in_control == True:
    nav_state = bug.new_state(nav_state, odo_state, bug_state)
    if bug_state.done:
      print("DONE")
      comms.drive(0, 0)
```

```
      break

      #only send stuff over serial if new values
      if not( speed_l == nav_state.speed_l and speed_r == nav_state.speed_r):
        comms.drive(nav_state.speed_l, nav_state.speed_r)

            speed_l = nav_state.speed_l
            speed_r = nav_state.speed_r

            ds.push(odo_state, nav_state.dist)
      # do not attempt to instantly read sensors again
            time.sleep(constants.MEASUREMENT_PERIOD_S)

    except TypeError as e:
        comms.drive(0,0)
        raise(e)

# ######################
# Init & CLI gubbins...
# ######################

if __name__ == "__main__":
    # Ignore 1st member, which is the name
    # the program was invoked with
    args = sys.argv[1:]

    # Read & Parse command line options
    try:
        optlist, args = getopt.getopt(args, 'hp:t:b:s:',
                                      ['help','port=','server=','baud=','timeout='])
    except getopt.GetoptError:
        print("Invalid Option, correct usage:")
        print(helptext)
        sys.exit(2)

    # Our defaults, may be different
    # from the ones built into the class'
    # __init__(self) constructor
    port = "/dev/ttyUSB0"
    timeout = 1
    baud = 9600

    server = "localhost"

    for opt, arg in optlist:
        if opt in ('-h', '--help'):
            print(namebadge)
            print(helptext)
            sys.exit(0)
        elif opt in ('-p', '--port'):
            # change serial port to use
            port = str(arg)

        elif opt in ('-t','--timeout'):
            # change blocking timeout for reading
            timeout = float(arg)

        elif opt in ('-b', '--baud'):
            # change baud rate
            baud = int(arg)

        elif opt in ('-s', '--server'):
            server = str(arg)
```

```
            print("Connecting to Redis server at " + str(server))

    # Initialise a serial class, or
    try:
        comms = Comms(port, baud, timeout)
    except Exception as e:
        if wt.confirm("Can't initialise serial, exit?\n\n"+str(e)):
            sys.exit(1)
        raise(e)

    print(namebadge)

    ds = DataStore(host=server)

    try:
        main()
    except KeyboardInterrupt as e:
        comms.drive(0,0)
        ds.save()
        print("Stopping and Quitting...")
        raise e

else:
    # if *not* running as __main__
    # invoke the class with defaults
    comms = Comms()
    ds = DataStore()
```

../../comms.py

```
#
#     - - - -  I A R  - - - -
#
#  s1311631         Angus Pearson
#  s1346981    Jevgenij Zubovskij
#

from __future__ import print_function
import serial # Documentation: http://pyserial.readthedocs.io/en/latest/index.html
import sys
from serial.tools import list_ports as list_ports
import time

class CommsReadException(Exception):
  pass

class Comms:

    port = serial.Serial()

    # Initialise the class, trying to open the Serial Port
    # with multiple levels of graceful failure.
    def __init__(self, port="/dev/ttyUSB0", baud=9600, timeout=1):

        self.port.baud = baud
        self.port.timeout = timeout

        try:
            self.port.port = port
            self.port.open()

        except serial.serialutil.SerialException as e1:
            print(e1)
            print("Trying alternate port...")
```

```
        try:
            self.port.port = "/dev/ttyUSB1"
            self.port.open()
            pass

        except serial.serialutil.SerialException as e2:
            print(e2)
            print("\n!!! CAN'T OPEN PORT !!!")

            # We can't open a port, enumerate them for the user
            print("\nAvailable Serial Ports:")
            for possible in list_ports.comports():
                print(possible)
            print("")

        raise(e1)
    self.clear_port()


def __del__(self):
    if self.port.is_open:
        self.port.close()

def _parse_sensor(self, string):
    data = string.strip('\n\r').split(",")
    data_ints = [int(d) for d in data[1:]]
    return data_ints

def clear_port(self):
    self.port.reset_input_buffer()

# directly control motor speeds
def drive(self, lspeed, rspeed):
    cmd = "D," + str(int(lspeed)) + "," + str(int(rspeed)) + "\n"
    print(cmd, end="")
    self.port.write(cmd)
    #print(self.port.readline(), end="")
    self.port.readline()

# self-explanatory
def stop(self):
    self.drive(0,0)

# Return odometry (wheel rotation) data
def get_odo(self):
    self.port.write("H\n")
    odo = self._parse_sensor(self.port.readline())

    if len(odo) != 2:
        raise CommsReadException("Odometry wrong length")

    return odo

# Reset the robot's wheel counts to 0
def reset_odo(self):
    self.port.write("G,0,0\n")
    self.port.readline()

# Return IR Distance measurements
def get_ir(self, sensor_no=None):
    self.port.write("N\n")
    dist = self._parse_sensor(self.port.readline())
```

```
  if len(dist) is not 8:
      raise CommsReadException("IR wrong length")

        if sensor_no is None:
            return dist
        else:
            return dist[sensor_no]


    # Return IR Ambient Light Measurements
    def get_ambient(self, sensor_no=None):
        self.port.write("O\n")
        amb = self._parse_sensor(self.port.readline())

  if len(amb) is not 8:
      raise CommsReadException("IR wrong length")
  return amb

    # control status LEDs on the robot
    def led(self, led_num=None, state=1):
        if state not in [0,1]:
            state = 0

        if led_num is None or led_num == 0:
            self.port.write("L,0," + str(state) + "\n")
        else:
            self.port.write("L,1," + str(state) + "\n")


    def blinkyblink(self):
        self.led(0,1)
        self.led(1,0)
        time.sleep(0.1)
        self.led(0,0)
        self.led(1,1)
        time.sleep(0.1)
        self.led(0,1)
        self.led(1,0)
        time.sleep(0.1)
        self.led(0,0)
        self.led(1,1)
        time.sleep(0.1)
        self.led(0,1)
        self.led(1,0)
        time.sleep(0.1)
        self.led(0,0)
        self.led(1,1)
        time.sleep(0.1)
        self.led(0,0)
        self.led(1,0)
        time.sleep(0.1)
        self.clear_port()
```

../../constants.py

```
#
#     - - - -  I A R  - - - -
#
#  s1311631         Angus Pearson
#  s1346981    Jevgenij Zubovskij
#

#REACTIVE THRESHOLDS
```

```
CONST_SPEED = 8
CONST_WALL_DIST = 200

#BOREDOM CONSTANTS

CONST_WALL_BORED_MAX = 100000
CONST_BORED_TURN_MAX = 20

#TURN SCALING

TURN_LESS = 0.2
TURN_MORE = 1.0

#REACTIVE STATES

STATE_DRIVE_FORWARD = 0
STATE_DRIVE_BACKWARD = 1
STATE_STUCK_LEFT = 2
STATE_STUCK_RIGHT = 3
STATE_LEFT_FOLLOW = 4
STATE_RIGHT_FOLLOW   = 5
STATE_BOREDOM_ROTATE = 6
STATE_BOREDOM_DRIVE  = 7


STATE_BUG_180 = 8


#ENCODER_CONSTANTS

TICKS_PER_MM = 12.0    # encoder ticks per milimeter
TICKS_PER_M = TICKS_PER_MM * 1000.0 # encoder ticks per meter

# PHYSICAL CONSTANTS

WHEEL_BASE_MM = 56.0 # mm
WHEEL_BASE_M = WHEEL_BASE_MM / 1000.0 # m



MEASUREMENT_PERIOD_S = 0.05 # s
EXPLORATION_CYCLES = 1.0 / MEASUREMENT_PERIOD_S


# SENSOR CONSTANTS

DIST_CUTOFF = 0.0


M_DISTANCE = 20 # mm
M_N_ANGLE = 5 # degrees
```

# References


[1] Principles of Robot Motion: Theory, Algorithms, and Implementation
*Howie Choset*