“Навигация на робот в динамична среда с използване на платформата Player и симулаторa Stage”

проект

по “Вградени Автономни Системи“

на  Антон Николаев Джонджоров,

магистърска програма „Изкуствен Интелект“

 фн М24055

02 Февруати 2013

ФМИ

СУ „Св. Климент Охридски“

Съдържание

1. Резюме ---------------------------------------------------------------------------- 3

2. Дизайн ---------------------------------------------------------------------------- 3

 2.1 Архитектура на робот при използване на платформата Player ------- 3

 2.2 Симулатор на роботи Stage --------------------------------------------------- 4

3. Симулация ----------------------------------------------------------------------- 7
 3.1 Общи характеристики ---------------------------------------------------------- 7
 3.2 Постановка ------------------------------------------------------------------------ 9
 3.3 Избягване на колизии ---------------------------------------------------------- 10

 3.4 Търсене на целта --------------------------------------------------------------- 10

4. Референции --------------------------------------------------------------------- 12
5. Използвана литература------------------------------------------------------ 13
6. Приложения---------------------------------------------------------------------- 14
 6.1 Конфигурационен .cfg файл--------------------------------------------------- 14
 6.2 Конфигурационен .world файл ----------------------------------------------- 15
 6.3 C++ код на приложението ----------------------------------------------------- 18

1.Резюме

Проектът представя движение и навигация на робот в динамична среда, съдържаща други движещи се обекти. Крайната задача на робота е да достигне до дадена цел, като при придвижването си роботът се стреми да избягва статични и динамични препядствия. Архитектурата на роботът се състои от различни сензори и акуратори, с които той да взаимодейства с външната среда. За решаването на този проблем са използвани алгоритми за избягване на препядствията и  експлоративно търсене. Проектът е изграден като модул на езикът C++  върху платформата Player[1], която е middleware система за управление на мобилни учебни роботи. За визуализация и симулация на процесите е използван 2D симулаторът  Stage[2] - plugin модул за Player.

2. Дизайн

2.1 Архитектура на робот при използване на платформата Player

Моделът на робота и постановката на задачата са изградени върху платформата Player. Платформата е създадена за управление на учебни роботи в университета на Южна Калифорния.

Player съдържа в себе си вградени драйвери за различни хардуерни устройста: сонари, камери, лазери и други, както и за различни прозиводители и марки на устройствата. Драйверите са HAW(Hardware Abstraction Layer) и комуникират директно с физическите компоненти на робота. Всеки драйвер комуникира с платформата чрез определен набор от интерфейси, също вградени в Player. Интерфейсите определят формата на данните, които ще бъдат изпращани и получавани между Player и драйверите. Физическото устройство и неговия драйвер със съответния интерфейс се разглеждат като едно устройство от програмна гледна точка.

Вътрешно Player представлява многонишков сървър за устройствата. Той използва собствен протокол, чрез който реализира комуникационния модел клиент-сървър. За транспорта на по-ниско ниво се използват TCP сокети, но е възможно да  използва и други протоколи като JINI[3]. Клиентския код се свързва директно към сървъра и получава сървиси, чрез които да комуникира с устройствата. Основната нишка на сървъра слуша за свързване на клиент и когато това се случи сървърът пуска нишки към отделните устройства. Устройствата имат собствени буфери за данни и команди и понеже Player не поддържа заключващ механизъм за достъп до тези буфери е възможно презаписване при множество клиенти. Трансфера на данните може да работи по PUSH и PULL методите[4].

 
Фиг. 1 Нива на абстракции в платформата Player

За свързване на клиента и комуникация със сървъра се използват специални класове от Player - проксита(proxies)[5]. Всеки интерфейс си има определено proxy. Клиентския код заедно с библиотеките на Player с които работи представлява функционалната подсистема на робота.

2.2 Симулатор на роботи Stage

 Като plugin към Player могат да се използват два симулатора :
- Stage - 2D симулатор
- Gazebo[6] -  3D симулатор

В този проект е използван първия. Stage се използва директно с Player, като имитира комуникацията на палтформата с физическите устройства. По този начин клиентски код, който работи на симулатора Stage може без особени промени да се пренесе и да работи на физически роботи.

 

Фиг. 2 Схема на взаимодействие между кода и физически устройства


Фиг. 3 Схема на взаимодействие между кода и симулатора Stage

В по-новите версии Stage поддържа 2.5D симулация[7] и по тази причина всички обекти има 3 измерения за размерите си.

Stage предоставя възможност за изграждене и симулация на среда и различни обекти. Клиентските файлове за симулиране се състоят от файлове с код, конфигурационни файлове, както и ресурси - най-често bmp модели на обекти от симулацията. Player/Stage поддържа множество езици за програмиране : C/C++,  Python, Ruby, Java и др. В кофигурационните файлове се задават харектеристика на модели - обекти, сензор, среди, както и се дефинират конкретни инстанции на тези модели. Задават се различни характеристики като форма, координати, цвят, размер, а също така и специфични за определеното устройство свойства. Също така  има конфигурационни файлове, при които се задават адреси и портове,  чрез които ще се осъществява комуникацията между клиентския код и различните обекти и сензори. За локална симулация това са най- често “localhost” и 6665. В началото на изпълняването се създава клиент, който се свързва към определен адрес- адресът на Player сървъра(localhost за локални приложения). След това се инстанцират proxy класове, като се посочва портът на който се свързват към определените устройства. Тези проксита се използват в по-късен етап за извличане на данните и изпращане на команди към устройствата. В кодът след това най-често се изпълнява безкраен цикъл със следната последователност от действия:

1. Четене на данните от сензорите
2. Вътрешна обработка на данните
3. Обновяване на данните на акураторите
4. Заспиване на нишката за определено време



Фиг. 4 Схема на цикъла на изпълнение

3. Симулация

3.1 Общи характеристики

Приложението(клиентския код) се сестои от 3 файла :
 *vas.cfg* - конфигурационен файл описващ адреси и портове на устройства, както и драйвери (в нашия случай Stage plugin-a)
 *vas.world* - конфигурационен файл описващ моделите в средата както и тяхните характеристики. В този файл се съдържа и информация за вида на средата и се задават конкретните индивиди от различните модели, които ще бъдат симулирани, заедно с координатите им в средата
 *vas.cc* - файл с код на C++, съдържащ изпълнимия код, който управлява робота посредством клиента и прокси класовете

Светът на симулацията съдържа карта - статичния терен, в който са поставени агентите.



Фиг.4 2D изглед от симулацията



Фиг.5 2.5D изглед от симулацията

Моделът на всеки обект съдържа неговата геометрия, зададена чрез файл представящ изображение(.bmp) или като блок от точки и размерите в тримерното пространство. Специален вид модел е позиция (position) описва обектите, които се движат. На тези обекти могат да се монтират различни сензори : камери, сонари и други. На всеки обект от тип position се задават акуратори за придвижване. Поддържат се два варината diff(диференциал)[8] - дава контрол на скоростта и завъртането на колелата и omni[9] - позволяват независимо движение във всички посоки. Също така има два модела на локализация - gps - глобално позициониране в средата и odom - одометрична система базирана на пъровначалната позиция на робота използвана при неизвстна среда.

Друг вид модели са сензорите. В Stage се поддържат следните видове сензори:

*ranger* - представя сензори които изследват контурите на средата в определен периметър. В зависимост от параметърите за зрителното поле, ъглите и броя и ъглите на лъчите може да представя различни видове сонари, лазери, инфрачервени сензори и други.

*camera* - представя камера с определена резолюция, обхват и ъгъл на заснемане

*blobfinder[10]*  - специална камера, която засича формата на предмети  определени цветове. Всеки цвят постъпва през различен канал и се вижда пикселизиран на различна  матрица, чиято големина зависи от резолюцията на камерата

Освен сензорите и акураторите, осигуряващи предвижването в средата има и още 1 акуратор:

*gripper[11]* -  държач, който позволява други обекти да бъдат повдигнати и преместени

Повечето сензори имат възможност за задаване на отклонения от измерванетo, което позволява по-реална симулация. Player/Stage платрформата позволява голяма разширяемост и е възможно добавяне на нови сензори и акуратори, чрез написване на драйвери и интерфейси.

3.2 Постановка

 Средата в която се движи роботът е динамична- средата се изменя и без намеса на роботът, стохастична- роботът не може да определи как ще изглежда средата след неговите действия и неизвестна – той не е наясно с това как изглежда.
За карта е използван bmp файл съдържащ изображение, кoйто автоматично е разпознат от Player/Stage. Характеристиките на преградите като височина на стените и размери на картата се четат от конфигурационния .world файл. Приложението има определена карта, но е възможно да работи с всякакви, стига да се сменят първоначалните координати на обектите.

Използван е модел на робот на име finder, който има цилиндрична форма и син цвят . За придвижване използва системата diff -диференциал, описана в миналата точка. Роботът не притежава gps и използва за координатна одометър- строи координатната си система по първоначалната си позиция и първоначално не притежава никакви знания за вида на средата. Той е снабден с лазерен сензор с поле 180 градуса и обсег от 2 метра. Лазерният сензор отчита неравностите в дадения обсег през няколко градуса и използва интерполация за да изчисли междинните точки. Роботът има и камера, която различава предмети със зелен цвят, за да може да разпознае целта. Камерата е с резолюция 120x80, спектър от 90 градуса и обхват от 2 метра. Когато камерата засече обект от съответния цвят  през прокси класа може да се достъпи масив с размери 120x80 съдържащ пикселизирана форма и размер на обекта.

Освен статичните прегради от картата съществуват и динамични препядствия (obstacles). Те представляват обекти, снабдени с лазери. Движението на препядствията е случайно като използват лазертните сензори за да не се сблъскват в преградите постоянно. Препядствията имат цилиндрична форма подобно на finder, но са с червен цвят за да бъдат различавани от наблюдателите на симулацията. Лазерните сензори на динамичните препядствия са разположени на горната им част и те не могат да засекат роботът finder( иначе биха го избягвали и не биха представлявали истински препядствия). Те също не могат да засичат и други препядствия като тях, за това са възможни колизии, които не са перманентни поради рандомизираното им пведение.

Всички действия и данни от сензорите са определени и съвършенни - не е зададено никакво ниво на шум или отклонения на устройството. Това отдалечава симулацията от реалността, но помага за  наблюденията на модела.

3.3 Избягване на колизии

Основна задача при навигацията на робота е да се придвижват в средата без да се сблъскват със статични или динамични препядствия. Данните  за околната среда постъпват чрез сензорите  - от лазерния сензор и камерата разпознаваща зелен цвят.

Роботът може да разграничи  данни за разстоянието му до препядствия на максимум 2 метра, защото толкова е обсегът на  лазера. За избягване на препяствията роботът използва  почти изцяло рефлексивна стратегия - директно съответствие между моментните данни получени от сензора и съответните команди изпратени към колелата на робота.  Роботът има зададен минимален праг на близост и когато дистанцията до обект стане по-малка от този праг  роботът завива под определен ъгъл, така че да избегне колизията. При фронтално  препядствие(засичане на обект по-близо от определения праг около 90 градусавия лъч) роботът  се премества на заден ход, за да избегне обекта. Не се пази карта на светът, защото се приема, че той е динамичен и е възможно да се изменя по неизвестен начин с течение на времето.
 Този подход за избягване на  колизии има няколко недостатъка. Отсъствието на сензори в задната част на роботът може да доведе до колизия със статичнен или динамичен обект - лазерът покрива поле с диапазон от 180 градуса.  Роботът finder се предвижва с определена скорост, която е и по-бавна при маневри близо до препядствия. Скоростта на динамичните роботи е случайна в някакви интервали. Това позволява понякога  при по висока скорост на препядствия, времето нужно да се обновят данните на finder и да се изпрати команда към акураторите за избягване да е недостатъчно. Друг недостатък е начина по-който работи лазерът( или на негово място равномерно разпределени сонари или други подобни устройства). Данните които се получават са през определен интервал от градуси, а останалите стойности се запълват чрез интерполация на известните стойности. Finder използва краен брой дискретни стойности от ъгли на лъчи, за които да засича опасност от сблъскване. Поърхнини с остри ръбове(независимо дали са динамични или не) е възможно да не бъдат засечени и роботът да се блъсне в тях.

3.4 Търсене на целта

Разпознаването на целта става благодарение на камерата. Невъзможно е да се различи целта, чрез лазерния сензор, защото той отчита единствено разстояние до обекти. Също не е възможно да се навигира до определени координати, тъй като в текущата формулировка на проблема роботът е в неизвестна среда. Камерата различава зеления цвят - и единствения предмет с зелен цвят е целта.

За намиране на целта се използва техника от reinforcement learning[11]. Поради неизвестността на средата е невъзможно да се планира път до целта. Също така е трудно да се построи вътрешно представяне на средата - модел на света, защото тя се мени непрекъснато. За намиране на целта finder използва експлоративно обхождане. Светът се състои от непрекъснато пространство, което трябва да се представи в ограничена памет. За да се справи с този проблем пространството се дискретизира. Определен радиус около някаква точка в двумерното пространство се приема за едно и също състояние. Придвижването на робота е също така непрекъснато и при него се налага дискретизация на възможните ротации. Приема се, че роботът има 4 възможни посоки - горе, долу, ляво и дясно и всяко негово завиване под някакъв ъгъл се дискретизира до тези стойнсоти на базата на интервала, в който е ъгъла на ротация. На всеки цикъл от обновяване на данните роботът запомня своето състояни и записва към него в коя посока е поел. Ако след командата пак е в същото състояние избира посока поемана най –малко пъти и поема в нея. В противен случай(когато сменя състоянието) роботът продължава да се движи в същата посока. Този алгоритъм осигурява избягването на цикли при близост с препядствия и кара роботът да предпочита неизбирани посоки от дадено състояние. По този начин той изследва средата. Включена е и рандомизация при избор на еднакво подходящи ротации в различни еквивалентни места от логиката.

4. Референции

[1] Player <http://playerstage.sourceforge.net/index.php?src=player>

[2] Stage <http://playerstage.sourceforge.net/index.php?src=stage>

[3] JINI <http://en.wikipedia.org/wiki/Jini>

[4] Push/Pull методи <http://www.vijaymukhi.com/vmis/pushpull.htm>

[5] Прокси(proxy) <http://en.wikipedia.org/wiki/Proxy_server>

[6] Gazebo <http://gazebosim.org/>

[7] 2.5D симулация <http://en.wikipedia.org/wiki/2.5D>

[8] Differential <http://ocw.mit.edu/courses/mechanical-engineering/2-12-introduction-to-robotics-fall-2005/lecture-notes/chapter5.pdf>

[9] Omni <http://www.societyofrobots.com/robot_omni_wheel.shtml>

[10] Blobfinder [http://www.cmpbjournal.com/article/S0169-2607(08)00207-1/abstract](http://www.cmpbjournal.com/article/S0169-2607%2808%2900207-1/abstract)

[11] Gripper <http://www.robots.com/articles/viewing/grippers-for-robots>

[12] Reinforcement learning <http://www.scholarpedia.org/article/Reinforcement_learning>

5. Използвана литература

<http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-186-mobile-autonomous-systems-laboratory-january-iap-2005/lecture-notes/>

<http://www.ee.nmt.edu/~wedeward/papers/2004ITEA.pdf>

[http://www-users.cs.york.ac.uk/jowen/player/play](http://www-users.cs.york.ac.uk/jowen/player/playerstage-tutorial-manual.pdf)
[erstage-tutorial-manual.pdf](http://www-users.cs.york.ac.uk/jowen/player/playerstage-tutorial-manual.pdf)

<http://yorkroboticist.blogspot.com/2011/12/installing-playerstage-in-ubuntu-1110.html>

<http://www.societyofrobots.com/programming_wavefront.shtml>

<http://www-users.cs.york.ac.uk/jowen/player/playerstage-tutorial-manual-2.1.pdf>

<http://en.wikipedia.org/wiki/Robotics_middleware#Player_Project>

<http://psurobotics.org/wiki/index.php?title=Player/Stage_Drivers>

<http://image.diku.dk/mediawiki/images/9/9d/Playerstagereport.pdf>

6. Приложения

6.1 Конфигурационен .cfg файл

driver

(

 name "stage"

 plugin "stageplugin"

 provides ["simulation:0"]

 # load the named file into the simulator

 worldfile "vas.world"

)

driver

(

 name "stage"

 provides ["6665:position2d:0" "6665:ranger:0" "6665:blobfinder:0"]

 model "finder"

)

driver

(

 name "stage"

 provides ["6665:position2d:1" "6665:ranger:1"]

 model "obstacle1"

)

driver

(

 name "stage"

 provides ["6665:position2d:2" "6665:ranger:2"]

 model "obstacle2"

)

driver

(

 name "stage"

 provides ["6665:position2d:3" "6665:ranger:3"]

 model "obstacle3"

)

driver

(

 name "stage"

 provides ["6665:position2d:4" "6665:ranger:4"]

 model "obstacle4"

)

driver

(

 name "stage"

 provides ["6665:position2d:5" "6665:ranger:5"]

 model "obstacle5"

)

driver

(

 name "stage"

 provides ["6665:position2d:6" "6665:ranger:6"]

 model "obstacle6"

)

6.2 Конфигурационен .world файл

include "map.inc"

define robot position

(

 size [0.5 0.5 0.5]

 bitmap "bitmaps/circle.png"

 drive "diff"

 laser()

 camera()

 obstacle\_return 1

 blob\_return 1

 ranger\_return 1

)

#laser

define laser ranger

(

 sensor(

 range\_max 2

 fov 180

 samples 180

 size [0.025 0.025 0.025]

 color "gray10"

 )

)

#camera

define camera blobfinder

(

 colors\_count 1

 colors ["green"]

 image [160 120] #resolution

 range 2

 fov 90

)

define obstacle position

(

 size [0.5 0.5 1]

 bitmap "bitmaps/circle.png"

 #block

 #(

 # points 4

 # point[3] [0 1]

 # point[2] [1 1]

 # point[1] [1 0]

 # point[0] [0 0]

 # z [0 1]

 #)

 drive "diff"

 obsLaser()

 obstacle\_return 1

 ranger\_return 1

)

#obstacle laser

define obsLaser ranger

(

 sensor(

 range\_max 1

 fov 180

 samples 180

 size [0.025 0.025 0.25]

 sview [0.025 0.025 -0.25]

 color "red10"

 )

)

define goal model

(

 bitmap "bitmaps/circle.png"

 size [0.25 0.25 1]

 color "green"

 blob\_return 1

)

#window

window

(

 size [500.000 500.000]

 scale 30

 show\_data 1

)

#environment

floorplan

(

 bitmap "./bitmaps/cave.png"

 size [15 15 1.5]

)

robot

(

 name "finder"

 pose [-5 -6 0 0]

 #pose [6 -3 0 0]

 color "blue"

)

obstacle

(

 name "obstacle1"

 pose [-2 1 0 -120]

 color "red"

)

obstacle

(

 name "obstacle2"

 pose [-3 5 0 0]

 color "red"

)

obstacle

(

 name "obstacle3"

 pose [-7 -5 0 0]

 #pose [6 2 0 0]

 color "red"

)

obstacle

(

 name "obstacle4"

 pose [-6 2 0 -90]

 color "red"

)

obstacle

(

 name "obstacle5"

 pose [5 5 0 0]

 color "red"

)

obstacle

(

 name "obstacle6"

 pose [6 -3 0 0]

 color "red"

)

goal

(

 name "goal"

 pose [-5 1 0 0]

)

6.3 C++ код на приложението

#include <stdio.h>

#include <libplayerc++/playerc++.h>

#include <time.h>

#include <libplayerinterface/player.h>

#define NUMOBSTACALES 6

#define STATES 1000

/\*need to do this line in c++ only\*/

using namespace PlayerCc;

const double PI = 3.14;

const double EPS = 0.0001;

enum Orientation {UP=0, DOWN=1, RIGHT=2, LEFT=3};

struct pose

{

 double x, y;

 int times[4];

};

pose model[STATES];

int poseCount = 0;

double stateRadius = 0.4;

double avoidDistance = 1.5;

int avoidTurn = 60;

pose previousState;

void PrintModel()

{

 for(int i=0;i<poseCount;i++)

 printf("%d (%f, %f) %d %d %d %d \n",i,model[i].x,model[i].y,model[i].times[0],model[i].times[1],model[i].times[2],model[i].times[3]);

}

Orientation GetOrientaionByMinTimes(int\* times)

{

 int min =100;

 int minIndex = 0;

 for(int i=0;i<4;i++)

 {

 if(times[i]<min)

 {

 min = times[i];

 minIndex = i;

 }

 }

 return minIndex;

}

double GetRandomAngle(double l, double h)

{

 return l + (double)rand() / ((double)RAND\_MAX / (h-l));

}

double GetAngle(Orientation o)

{

 if(o == UP)

 {

 return GetRandomAngle(PI/2 - EPS, PI/2 + EPS);//pi/2;

 }

 if(o == RIGHT)

 {

 return GetRandomAngle(0-EPS, 0+EPS);

 }

 if(o == DOWN)

 {

 return GetRandomAngle(-PI/2 - EPS, -PI/2 + EPS);

 }

 return GetRandomAngle(-PI-EPS, -PI+EPS);

}

Orientation GetOrientation(double a)

{

 if(a>=PI/4 && a<3\*PI/4)

 {

 return UP;

 }

 if(a>=-PI/4 && a<PI/4)

 {

 return RIGHT;

 }

 if(a>-3\*PI/4 && a<-PI/4)

 {

 return DOWN;

 }

 return LEFT;

}

double ProcedePose(Position2dProxy& pos)

{

 double x=pos.GetXPos();

 double y=pos.GetYPos();

 double a=pos.GetYaw();

 pose currentState;

 for(int i=0;i<poseCount;i++)

 {

 if((model[i].x - x)\*(model[i].x - x) + (model[i].y - y)\*(model[i].y - y) < stateRadius\*stateRadius)

 {

 Orientation o;

 if((model[i].x - previousState.x)\*(model[i].x - previousState.x) + (model[i].y - previousState.y)\*(model[i].y - previousState.y) < stateRadius\*stateRadius)

 {

 o = GetOrientaionByMinTimes(model[i].times);

 model[i].times[o]++;

 }

 else

 {

 return a;

 }

 previousState = model[i];

 return GetAngle(o);

 }

 }

 model[poseCount].x=x;

 model[poseCount].y=y;

 for(int i=0;i<4;i++)

 {

 model[poseCount].times[i]=0;

 }

 model[poseCount].times[RIGHT]=1;

 previousState = model[poseCount];

 poseCount++;

 return a;

}

void SavePose(Position2dProxy& pos)

{

 double x=pos.GetXPos();

 double y=pos.GetYPos();

 double a=pos.GetYaw();

 Orientation o = GetOrientation(a);

 for(int i=0;i<poseCount;i++)

 {

 if((model[i].x - x)\*(model[i].x - x) + (model[i].y - y)\*(model[i].y - y) < stateRadius\*stateRadius)

 {

 if((model[i].x - previousState.x)\*(model[i].x - previousState.x) + (model[i].y - previousState.y)\*(model[i].y - previousState.y) < stateRadius\*stateRadius)

 {

 model[i].times[o]++;

 }

 previousState = model[i];

 return;

 }

 }

 model[poseCount].x = x;

 model[poseCount].y = y;

 for(int i=0;i<4;i++)

 {

 model[poseCount].times[i] = 0;

 }

 model[poseCount].times[o] = 1;

 previousState = model[poseCount];

 poseCount++;

}

void MoveObs(double \*speed, double \*turn, RangerProxy& laser, Position2dProxy& pos)

{

 \*speed = (double)(rand()%7)/(double)10-0.2;

 \*turn = rand()%121-60;

 if(laser[70] < avoidDistance)

 {

 \*speed = 0;

 \*turn = avoidTurn;

 return;

 }

 else if(laser[120] < avoidDistance)

 {

 \*speed = 0;

 \*turn = (-1)\*avoidTurn;

 return;

 }

 else if( (laser[85] < avoidDistance) && (laser[95] < avoidDistance))

 {

 \*speed = -0.2;

 \*turn = (rand()%2 ? 1 : -1)\*avoidTurn;

 return;

 }

}

void Move(double \*speed, double \*turn, RangerProxy& laser, Position2dProxy& pos, BlobfinderProxy& cam)

{

 srand ( time(NULL) );

 \*speed = 0.5;

 PrintModel();

 if( (laser[75] < avoidDistance) || ((laser[75] < avoidDistance) && (laser[105] < avoidDistance)))

 {

 \*speed = -0.5;

 \*turn = (rand()%2 ? 1 : -1)\*avoidTurn;

 pos.SetSpeed(\*speed, dtor(\*turn));

 SavePose(pos);

 return;

 }

 else if(laser[60] < avoidDistance)

 {

 \*turn = avoidTurn;

 pos.SetSpeed(\*speed, dtor(\*turn));

 SavePose(pos);

 return;

 }

 else if(laser[120] < avoidDistance)

 {

 \*turn = (-1)\*avoidTurn;

 pos.SetSpeed(\*speed, dtor(\*turn));

 SavePose(pos);

 return;

 }

 else if(laser[45] < avoidDistance)

 {

 \*turn = avoidTurn/2;

 pos.SetSpeed(\*speed, dtor(\*turn));

 SavePose(pos);

 return;

 }

 else if(laser[135] < avoidDistance)

 {

 \*turn = (-1)\*avoidTurn/2;

 pos.SetSpeed(\*speed, dtor(\*turn));

 SavePose(pos);

 return;

 }

 else if(laser[30] < avoidDistance)

 {

 \*turn = avoidTurn/12;

 pos.SetSpeed(\*speed, dtor(\*turn));

 SavePose(pos);

 return;

 }

 else if(laser[150] < avoidDistance)

 {

 \*turn = (-1)\*avoidTurn/12;

 pos.SetSpeed(\*speed, dtor(\*turn));

 SavePose(pos);

 return;

 }

 \*turn =-pos.GetYaw()+ProcedePose(pos);

 pos.SetSpeed(\*speed, \*turn);

}

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

{

 PlayerClient client("localhost", 6665);

 Position2dProxy position(&client, 0);

 RangerProxy laser(&client, 0);

 BlobfinderProxy camera(&client, 0);

 position.RequestGeom();

 laser.RequestGeom();

 Position2dProxy \*obstaclesPositions[NUMOBSTACALES];

 RangerProxy \*obstaclesLasers[NUMOBSTACALES];

 for(int i=1;i<=NUMOBSTACALES;i++)

 {

 obstaclesPositions[i-1]=new Position2dProxy(&client, i);

 obstaclesLasers[i-1]=new RangerProxy(&client, i);

 obstaclesPositions[i-1]->RequestGeom();

 obstaclesLasers[i-1]->RequestGeom();

 }

 previousState.x=0;

 previousState.y=0;

 double speed, turn;

 double obsSpeed[NUMOBSTACALES], obsTurn[NUMOBSTACALES];

 //enable motors

 position.SetMotorEnable(true);

 while(true)

 {

 client.Read();

 if(camera.GetCount()>0)

 {

 position.SetSpeed(0,0);

 for(int i=1;i<=NUMOBSTACALES;i++)

 {

 obstaclesPositions[i-1]->SetSpeed(0,0);

 }

 sleep(1);

 continue;

 }

 Move(&speed, &turn, laser, position, camera);

 srand ( time(NULL) );

 for(int i=1;i<=NUMOBSTACALES;i++)

 {

 MoveObs(&obsSpeed[i-1], &obsTurn[i-1], \*obstaclesLasers[i-1], \*obstaclesPositions[i-1]);

 obstaclesPositions[i-1]->SetSpeed(obsSpeed[i-1],dtor(obsTurn[i-1]));

 }

 sleep(1);

 }

 return 0;

}