Player/Stage controlador e simulador de robôs móveis

Instalação e utilização

Laboratório de Robótica Móvel LRM - ICMC/USP
www.icmc.usp.br/~lrm

V2.1.3a

1 – Introdução

O Player é um sistema para controle de robôs móveis amplamente utilizado por universidades, institutos de pesquisa e empresas em diversos países. Seu desenvolvimento foi iniciado em 2000 por pesquisadores da *University of Southern Califórnia* – EUA para suprir a demanda de um controlador/simulador de robôs móveis, que fosse eficiente, de grande compatibilidade e independente de arquitetura de programação do robô. Atualmente o desenvolvimento do Player conta com a colaboração de diversos pesquisadores das mais diversas instituições. Por ser um sistema de código aberto e de livre distribuição, o Player está em constante desenvolvimento para se adequar a um número cada vez maior de plataformas robóticas e sensores comercias. Da mesma forma que um sistema operacional cria uma interface de alto nível para facilitar o acesso de usuários/programadores ao hardware de um computador, o Player apresenta uma interface de acesso ao hardware de robôs móveis e sensores, tornando simples sua programação e utilização.

A estrutura do Player é baseada no modelo cliente/servidor. O servidor faz a interface com o robô e com outros sensores, obtendo dados, enviando-os para o cliente e recebendo instruções do cliente para o controle do robô e dos sensores. O cliente é o programa que controla efetivamente o robô (aplicação). O cliente é responsável por obter os dados do servidor, interpretá-los e enviar instruções para o servidor (robô) para a execução de determinada tarefa. A arquitetura cliente/servidor permite grande versatilidade no controle robôs e sensores, de forma que um cliente (programa de controle) pode controlar diversos servidores e diferentes clientes podem controlar diferentes sensores de um mesmo robô. Toda a comunicação entre cliente e servidor é realizada através de TCP/IP. O cliente Player foi projetado para ser compatível com diversas linguagens. Atualmente, existem bibliotecas disponíveis para clientes em C, C++, Java, Python, Tcl, entre outras. Para que o cliente possa se comunicar com o servidor é necessário que o código fonte do programa de controle inclua uma biblioteca fornecida com o Player.

O Stage é um simulador de robôs e sensores para ambientes bidimensionais compatível com o Player (Figura 1). Múltiplos robôs e sensores podem ser simulados simultaneamente, controlados por um ou mais clientes. O Stage normalmente é usado para o desenvolvimento inicial de código, até que o mesmo se encontre confiável o suficiente para ser testado em robôs reais. Outras aplicações do Stage envolvem a utilização de sensores e robôs em quantidade não disponível na prática, mas que podem ser validadas através de simulação. Como toda a comunicação cliente/servidor é realizada através da rede (TCP/IP), é totalmente transparente para o programa cliente (aplicação) se o mesmo está conectado a um robô simulado ou a um robô real.

Normalmente, a única modificação necessária para se executar um código desenvolvido para controlar um robô simulado em um robô real é a mudança do IP do robô simulado (computador que executa o simulador) para o IP do robô real.

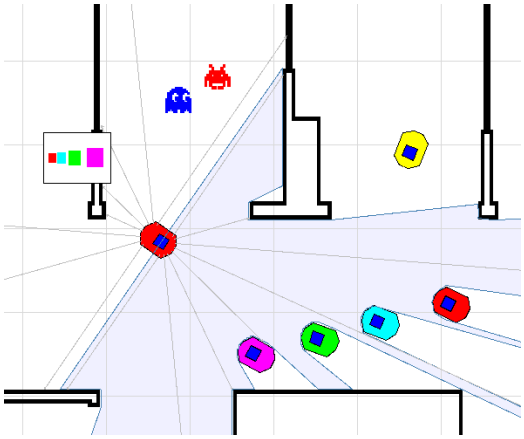
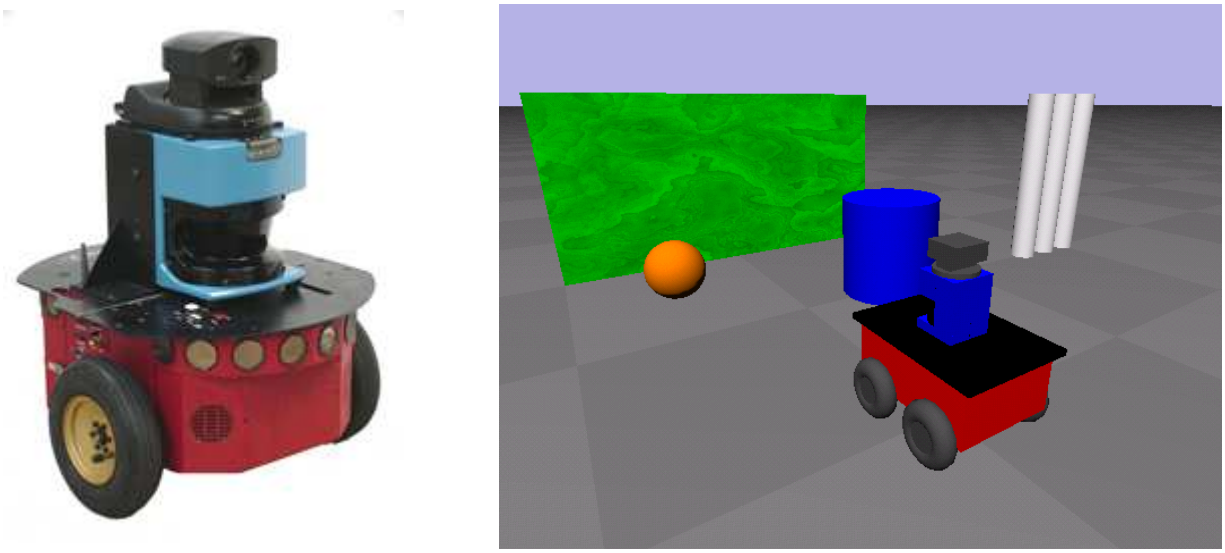


Figura 1: Simulador Multi-robôs Stage

Além do Stage, o Player também é compatível com o simulador Gazebo. O Gazebo é um simulador de robôs móveis em 3D que permite a simulação realista de ambientes complexos. Através do uso de bibliotecas de modelagem física, o Gazebo permite uma simulação extremamente fiel do comportamento e da interação física de robôs e objetos do ambiente (Figura 2). Por ser computacionalmente mais complexo que o Stage, o Gazebo requer mais recursos computacionais.



Pioneer DX2 Simulador Gazebo

Figura 2: Robô real e robô simulado

O Gazebo normalmente é utilizado em situações onde a simulação bidimensional do Stage não é fiel o suficiente. Por exemplo em ambientes externos, onde o solo é irregular e isso compromete substancialmente o funcionamento dos robôs e sensores. Ou quando o robô deve mapear ambientes externos, incluindo árvores e carros. Nesse caso, não é possível modelar esses objetos no Stage.

2- Instalação

O código fonte do Player/Stage/Gazebo, bem como os manuais originais desses softwares, podem ser obtidos gratuitamente em http://playerstage.sourceforge.net. A instalação do Player é simples, porém exige conhecimentos básicos do sistema operacional Linux.

Os passos para a instalação da versão 2.1.3 são:

```
./configure
make
make install
```

Se nenhuma outra opção for especificada, o Player é instalado no diretório /usr/local, que exige a permissão do root. Após a instalação, os executáveis do Player estarão em /usr/local/bin. As bibliotecas estarão em /usr/local/lib e os arquivos de header estarão em /usr/local/include. Mais opções de instalação podem ser obtidas através do comando:

```
./configure –help
```

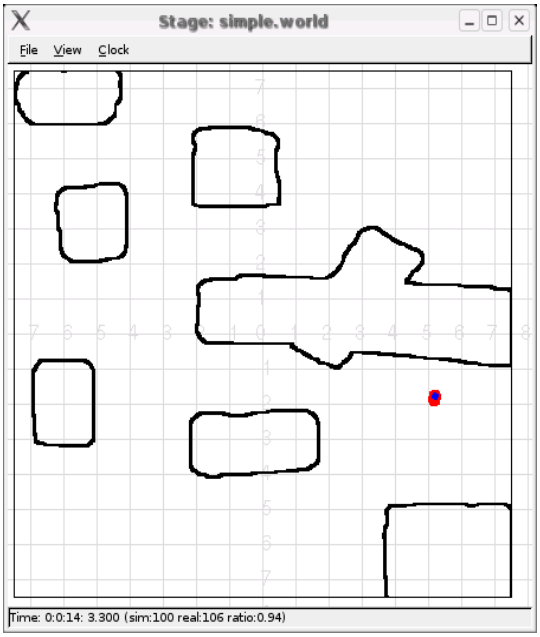
O simulador Stage deve ser instalado utilizando-se a mesma seqüência de comandos descrita na instalação do Player. Depois de instalado o Stage pode ser testado da seguinte forma:

```
player    /usr/local/share/stage/worlds/simple.cfg
```

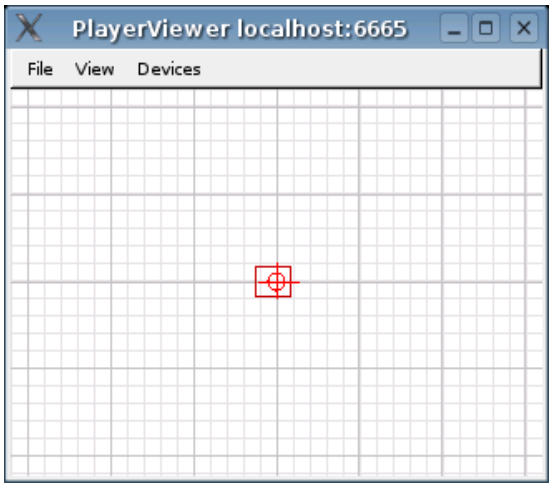
Após executada o comando acima, uma janela do Stage deve abrir com um robô no canto inferior esquerdo da tela. Para testar o controle do robô pode-se utilizar a ferramenta 'playerv' da seguinte forma:

```
playerv
```

Utilize as općões Devices->position2d->Subscribe, Devices->position2d->Command e Devices->laser->Subscribe. É possível fazer o robô se mover clicando no centro do robô no playerv e movendo a seta do mouse para frente. Também é possível visualizar as informações obtidas pelo laser através do playerv.



Stage (simple.cfg)



Playerv

Um erro comum na hora de executar o stage é a falta do arquivo “rbg.txt”:

```
Registering driver
Player v.2.1.3

* Part of the Player/Stage/Gazebo Project
[http://playerstage.sourceforge.net].
* Copyright (C) 2000 - 2006 Brian Gerkey, Richard Vaughan, Andrew Howard,
* Nate Koenig, and contributors. Released under the GNU General Public
License.
* Player comes with ABSOLUTELY NO WARRANTY.  This is free software, and you
* are welcome to redistribute it under certain conditions; see COPYING
* for details.

invoking player_driver_init()...
 Stage driver plugin init

 ** Stage plugin v2.1.1 **
 * Part of the Player/Stage Project [http://playerstage.sourceforge.net]
 * Copyright 2000-2006 Richard Vaughan, Andrew Howard, Brian Gerkey
 * and contributors. Released under the GNU General Public License v2.
success
  Stage driver creating 1 device
   6665.31.0 is a Stage world [Loading
/usr/local/share/stage/worlds/simple.world][Include
pioneer.inc][Include map.inc][Include sick.inc]
err: unable to open color database: No such file or directory (stage.c
stg_lookup_color)
```

Neste caso, é necessário copiar o arquivo rgb.txt para a pasta /usr/X11R6/lib/X11.

3 – Biblioteca Player C (cliente)

3.1 - Client

O cliente Player é uma bilbioteca de funções que permite a comunicação entre programa que controla o robô e o servidor (robô real ou simulado). O cliente Player é utilizado para se enviar instruções e obter dados do servidor, por exemplo: posição do robô, leitura do laser, leitura dos sonares e etc. As principais funções do cliente são:

playerc_client_t * playerc_client_create (playerc_mclient_t *mclient, const char *host, int port)

Cria um objeto do tipo cliente para ser conectado ao servidor no endereço “host” e na porta “port”.

void playerc_client_destroy (playerc_client_t *client)

Destroi o objeto cliente.

int playerc_client_connect (playerc_client_t *client)

Conecta o cliente ao servidor.

int playerc_client_disconnect (playerc_client_t *client)

Desconecta o cliente do servidor.

void * playerc_client_read (playerc_client_t *client)

Solicita dados do robô e dos sensors para o servidor.

3.2 - Position2d

O position2d é o dispositivo utilizado para se obter informações sobre a posição do robô (baseado no seu odômetro interno) e para mover o robô. A estrutura principal do position2d é:

```
struct  playerc_position2d_t {
        double px, py          ; //posição (X, Y) do robô em m.
        double pa; //ângulo (direção) do robô em radianos.
        double vx, vy, va; //velocidade do robô em m/s e radianos/s.
        int stall; // estado dos motores. 1 para motores operando, 0 para motores pausados
        }
```

Estrutura auxiliar:

```
struct  player_pose_t {
        double px, py, pa     ; //posição (X, Y) do robô em m e orientação em radianos.
        }
```

Principais funções para a operação do position2d:

playerc_position2d_t * playerc_position2d_create (playerc_client_t *client, int index)

Cria um objeto do tipo position2d, dados o cliente player e o índice (normalmente 0).

void playerc_position2d_destroy (playerc_position2d_t *device)

Destroi um objeto do tipo position2d.

int playerc_position2d_subscribe (playerc_position2d_t *device, int access)

Conecta ao objeto position2d. O acesso deve ser 'PLAYERC_OPEN_MODE'.

int playerc_position2d_unsubscribe (playerc_position2d_t *device)

Desconecta o objeto position2d do robô.

int playerc_position2d_enable (playerc_position2d_t *device, int enable)

Ativa os motores do robô para a operação, quando enable = 1.

int playerc_position2d_set_cmd_vel (playerc_position2d_t *device, double vx, double vy, double va, int state)

Movimenta o robô em uma velocidade definida, onde vx corresponde a velocidade com que o robô se move para a frente e, va a velocidade rotacional, (vy não é utilizado na maioria dos robôs) e state deve ser 1.

int playerc_position2d_set_odom (playerc_position2d_t *device, double px, double py, double pa)

Altera as informações do odômetro.

int playerc_position2d_set_cmd_pose (playerc_position2d_t *device, px, py, pa, int state)

Define uma posição a ser alcançada pelo robô.

int playerc_position2d_set_cmd_pose_with_vel (playerc_position2d_t *device, player_pose_t pos, player_pose_t vel, int state)

Define uma posição a ser alcançada pelo robô e a velocidade a ser utilizada. É importante ressaltar que as duas funções anteriores apenas tentam conduzir o robô até o ponto determinado, sem desviar de obstáculos no caminho. Para incluir o desvio automático de obstáculos, é necessário usar position2d e o laser (ou o sonar) através do driver "vfh", criando um novo position2d. O vfh deve ser definido no arquivo de configuração do Player.

Para se controlar o robô simulado através de um programa, a biblioteca cliente do Player deve ser incluída. Segue o exemplo de um programa que move o robô.

```
#include <stdio.h>
#include <libplayerc/playerc.h>

int main(int argc, const char **argv)
{
  int i;

  //Cliente para conexão com o servidor
  playerc_client_t *client;

  //Objeto para conexão com o odômetro
  playerc_position2d_t *position2d;

  //Conecta ao servidor no endereço “localhost”, na porta 6665 (default)

  client = playerc_client_create(NULL, "localhost", 6665);
  if (playerc_client_connect(client) != 0)
  {
    fprintf(stderr, "error: %s\n", playerc_error_str());
    return -1;
  }

  //Conecta ao odômetro
  position2d = playerc_position2d_create(client, 0);
  if (playerc_position2d_subscribe(position2d, PLAYERC_OPEN_MODE) != 0)
  {
    fprintf(stderr, "error: %s\n", playerc_error_str());
    return -1;
  }

  //Ativa os motores do robô
  playerc_position2d_enable(position2d, 1);

  //Comando para o robô se mover para frente a 0,2m/s e rotacionar a 0.4rad/s
  playerc_position2d_set_cmd_vel(position2d, 0.2, 0, 0.4, 1);


  for (i = 0; i < 400; i++)
  {

    //Recebe dados do servidor
    playerc_client_read(client);

    //Mostra a posição do robô
    printf("position : %f %f %f\n", position2d->px, position2d->py, \
    position2d->pa);
  }

  //Desconecta do odômetro e do servidor
  playerc_position2d_unsubscribe(position2d);
  playerc_position2d_destroy(position2d);
  playerc_client_disconnect(client);
  playerc_client_destroy(client);

  return 0;
}
```

Para compilar o código acima, basta salvá-lo como test.c e executar o seguinte comando:

```
gcc test.c -o test -I/usr/local/include/player-2.1 -L/usr/local/lib -lplayerc -lm -lplayerxdr -lplayererror
```

Para executar o programa compilado (e fazer o robô se mover):

```
./test
```

3.3 - Laser

O LASER é um sensor que mede a distância entre o robô e os obstáculos ao seu redor. O laser cobre um campo 180 graus, com 180 ou 360 leituras (leituras a cada 1 grau ou a cada meio grau), dependendo da configuração utilizada no servidor.

A distância máxima que pode ser lida do laser é, normalmente, 8m em ambientes fechados. O laser pode ser configurado para a distância máxima de 80m para a utilização em ambiente externos.

A estrutura principal do laser é:

```
struct  playerc_laser_t {
    int scan_count; //número de leituras do laser.
    double scan[scan_count][2]; // [0] distância até obstáculo em m e [1] ângulo da
                                   leitura em radianos.
 }
```

Principais funções para a operação do laser:

```
playerc_laser_t * playerc_laser _create (playerc_client_t *client, int index)
```

Cria um objeto do tipo laser, dados o cliente player e o índice (normalmente 0).

```
void playerc_laser _destroy (playerc_laser _t *device)
```

Destroi um objeto do tipo laser.

```
int playerc_laser _subscribe (playerc_laser _t *device, int access)
```

Conecta ao objeto laser. O acesso deve ser ‘PLAYERC_OPEN_MODE’.

```
int playerc_ laser _unsubscribe (playerc_laser _t *device)
```

Desconecta o objeto laser do robô.

O exemplo a seguir ilustra a utilização do laser através da leitura dos sensores apontados

para a esquerda, direita e para frente (configurado para 360 leituras).

```
#include <stdio.h>
#include <libplayerc/playerc.h>

int main(int argc, const char **argv)
{
  int i;
  playerc_client_t *client;
  playerc_laser_t *laser;

  client = playerc_client_create(NULL, "localhost", 6665);
  if (playerc_client_connect(client) != 0)
    return -1;

  laser = playerc_laser_create(client, 0);
  if (playerc_laser_subscribe(laser, PLAYERC_OPEN_MODE))
    return -1;




  for (i = 0; i < 10000; i++)
  {
    playerc_client_read(client);

    //Mostra a leitura do laser, sensores 0 (esquerda),
    //180 (frente) e 360 (direita)
    printf("Laser: direita:%.3fm\n       esquerda: %.3fm\n       frente:  \
    %.3fm\n\n", laser->scan[0][0], laser->scan[360][0], laser->scan[180][0]);
  }

  playerc_laser_unsubscribe(laser);
  playerc_laser_destroy(laser);
  playerc_client_disconnect(client);
  playerc_client_destroy(client);

  return 0;
}
```

3.3 - Sonar

O sonar é um sensor que mede a distância entre o robô e os obstáculos ao seu redor. Comparado ao laser, o sonar apresenta uma precisão menor, além de cobrir uma área menor. O número de sonares varia, dependendo do modelo do robô. Normalmente são utilizados 8 ou 16 sonares apontados para direções diferentes. A distância máxima que pode ser lida do sonar é de 5m.

A estrutura principal do sonar é:

struct playerc_sonar_t {

int scan_count; //número de sensors (sonares).

double scan[scan_count]; //distância até obstáculo em m.

poses[scan_count][3]; //posição e direção de cada um dos sensores (x, y, dir)
em metros e graus.

}

Principais funções para a operação do sonar:

playerc_sonar_t * playerc_sonar_create (playerc_client_t *client, int index)

Cria um objeto do tipo sonar, dados o cliente player e o índice (normalmente 0).

void playerc_sonar_destroy (playerc_sonar_t *device)

Destroi um objeto do tipo sonar.

int playerc_sonar_subscribe (playerc_sonar_t *device, int access)

Conecta ao objeto sonar. O acesso deve ser 'PLAYERC_OPEN_MODE'.

int playerc_sonar_unsubscribe (playerc_sonar_t *device)

Desconecta o objeto sonar do robô.

As informações do sonar são obtidas através do cliente Player. O exemplo a seguir ilustra a utilização do sonar através da leitura dos sensores apontados para a esquerda, direita e para frente.

Para se testar o sonar, **é necessário** carregar o ambiente **everything.cfg** no stage, uma vez que o robô do ambiente simple.cfg não possui sonar.

```
#include <stdio.h>
#include <libplayerc/playerc.h>

int main(int argc, const char **argv)
{
  int i;
  playerc_client_t *client;
        playerc_sonar_t *sonar;

  client = playerc_client_create(NULL, "localhost", 6665);
  if (playerc_client_connect(client) != 0)
    return -1;

  sonar = playerc_sonar_create(client, 0);
  if (playerc_sonar_subscribe(sonar, PLAYER_OPEN_MODE))
    return -1;

  for (i = 0; i < 1000; i++)
  {
    playerc_client_read(client);

    //Mostra a leitura do sonar, sensores 0 (esquerda), 4 (frente) e 7 (direita)
```

```
    printf("Sonar: esquerda:%.3fm\n       frente: %.3fm\n       direita: \
    %.3fm\n\n", sonar->scan[0], sonar->scan[4], sonar->scan[7]);
  }

  playerc_sonar_unsubscribe(sonar);
  playerc_sonar_destroy(sonar);
  playerc_client_disconnect(client);
  playerc_client_destroy(client);

  return 0;
}
```

3.4 - Blobfinder

O blobfinder é um dispositivo virtual que utiliza imagens de uma câmera de vídeo para localizar objetos de cores específicas (blobs), tornando mais simples tarefas como identificar ou seguir objetos de determinada cor.

A estrutura de dados principal do blobfinder é:

```
struct  playerc_blobfinder_t {
        unsigned int width; //largura da imagem em pixels.
        unsigned int height; // altura da imagem em pixels.
        unsigned int blobs_count; // número de blobs localizados na imagem
        playerc_blobfinder_blob_t blobs[MAX_BLOBS]; //detalhes dos blobs detectados
        }
```

Estrutura auxiliar:

```
playerc_blobfinder_blob_t {
        unsigned int id; //identificação do blob.
        unsigned int color; //cor do blob (0x00RRGGBB).
        unsigned int area; //área do blob na tela, em pixels.
        unsigned int x; //centro do blob na tela (eixo x), em pixels.
        unsigned int y; //centro do blob na tela (eixo y), em pixels.
        unsigned int left, right, bottom, top; //limites do blob na imagem, em pixels.
        float range; //distância aproximada do centro do blob em mm.
        }
```

Principais funções para a operação do blobfinder:

playerc_blobfinder_t * playerc_blobfinder_create (playerc_client_t *client, int index)

Cria um objeto do tipo blobfinder, dados o cliente player e o índice (normalmente 0).

void playerc_blobfinder_destroy (playerc_blobfinder_t *device)

Destroi um objeto do tipo blobfinder.

int playerc_blobfinder_subscribe (playerc_blobfinder_t *device, int access)

Conecta ao objeto blobfinder. O acesso deve ser 'PLAYERC_OPEN_MODE'.

int playerc_blobfinder_unsubscribe (playerc_blobfinder_t *device)

Desconecta o objeto blobfinder do robô.

O exemplo a seguir mostra quantos blobs são detectados em um dado momento e a cor, tamanho e posição (x, y) do blob na tela. Também é necessário carregar o ambiente **everything.cfg** para testar o blobfinder.

```
#include <stdio.h>
#include <libplayerc/playerc.h>

int main(int argc, const char **argv)
{
  int i, k;
  playerc_client_t *client;
        playerc_blobfinder_t *bf;

  client = playerc_client_create(NULL, "localhost", 6665);
  if (playerc_client_connect(client) != 0)
    return -1;

  bf = playerc_blobfinder_create(client, 0);
  if (playerc_blobfinder_subscribe(bf, PLAYER_OPEN_MODE))
    return -1;

  for (i = 0; i < 1000; i++)
  {
    playerc_client_read(client);

                printf("\nNumber of blobs:%u\n",bf->blobs_count);

                for(k = 0; k < bf->blobs_count; k++)
                        printf("Blob[%u] color:%8u, area:%4u, x:%2u, y:%2u\n", \
    bf->blobs[k].id, bf->blobs[k].color, bf->blobs[k].area, bf->blobs[k].x, \
    bf->blobs[k].y);
  }

  playerc_blobfinder_unsubscribe(bf);
  playerc_blobfinder_destroy(bf);
  playerc_client_disconnect(client);
  playerc_client_destroy(client);

  return 0;
}
```

3.5 - Gripper

O gripper é um conjunto de garras mecânicas que pode ser usado para suspender e carregar pequenos objetos. A estrutura de dados principal do blobfinder é:

```
struct  playerc_gripper_t {
        int paddles_open; //indica se as garras estão abertas.
        int paddles_closed; //indica se as garras estão fechadas.
        int paddles_moving; //indica se as garras estão se movendo.
        int lift_up; //indica se as garras suspensas.
        int lift_down; //indica se as garras abaixadas.
        int lift_moving; //indica se as garras se movendo.
        int lift_error; //indica se há algum erro ao mover as garras.
        int gripper_error; //indica se há algum erro com as garras.
       }
```

Principais funções para a operação do gripper:

playerc_gripper_t * playerc_gripper_create (playerc_client_t *client, int index)

Cria um objeto do tipo gripper, dados o cliente player e o índice (normalmente 0).

void playerc_gripper_destroy (playerc_gripper_t *device)

Destroi um objeto do tipo gripper.

int playerc_gripper_subscribe (playerc_gripper_t *device, int access)

Conecta ao objeto gripper. O acesso deve ser ‘PLAYERC_OPEN_MODE’.

int playerc_gripper_unsubscribe (playerc_gripper_t *device)

Desconecta o objeto gripper do robô.

int playerc_gripper set_cmd (playerc_gripper_t *device, cmd, arg)

Os comandos que podem ser usados como argumento (cmd) para a funçao anterior são:

GRIPopen //abre as garras

GRIPclose //fecha as garras

GRIPstop //para de movimentar as garras

LIFTup //eleva as garras

LIFTdown //abaixa as garras

LIFTstop //para de movimentar as garras

O 3º argumento (arg) deve ser 0. O exemplo a seguir mostra o robô abrindo as garras, fechando as garras (caso houver algum objeto entre as garras, ele será apanhado), movendo o robô e novamente abrindo as garras (e soltando o objeto).

```
#include <stdio.h>
#include <libplayerc/playerc.h>

int main(int argc, const char **argv)
{
  int i, k;
  playerc_client_t *client;
        playerc_position2d_t *position2d;
        playerc_gripper_t *gripper;


  client = playerc_client_create(NULL, "localhost", 6665);
  if (playerc_client_connect(client) != 0)
    return -1;

  position2d = playerc_position2d_create(client, 0);
  if (playerc_position2d_subscribe(position2d, PLAYERC_OPEN_MODE) != 0)
  {
    fprintf(stderr, "error: %s\n", playerc_error_str());
    return -1;
  }

  gripper = playerc_gripper_create(client, 0);
  if (playerc_gripper_subscribe(gripper, PLAYER_OPEN_MODE))
    return -1;

        playerc_position2d_enable(position2d, 1);

        playerc_gripper_set_cmd(gripper, GRIPopen, 0);
        playerc_gripper_set_cmd(gripper, GRIPclose, 0);
        sleep(2);

  playerc_position2d_set_cmd_vel(position2d, 0.2, 0, 0, 1);

  for (i = 0; i < 20; i++)
  {
    playerc_client_read(client);
  }

  playerc_position2d_set_cmd_vel(position2d, 0, 0, 0, 1);

        playerc_gripper_set_cmd(gripper, GRIPopen, 0);
        sleep(2);

  playerc_gripper_unsubscribe(gripper);
  playerc_gripper_destroy(gripper);
  playerc_position2d_unsubscribe(position2d);
  playerc_position2d_destroy(position2d);
  playerc_client_disconnect(client);
  playerc_client_destroy(client);

  return 0;
}
```