Universidad de Granada

decsai.ugr.es

PRÁCTICAS TSI - 2015

Subgrupo 1.4

Pedro Antonio Ruiz Cuesta Eva Almansa Aranega
Jose Hernandez Casado Ignacio Martín Requena

DECSAI

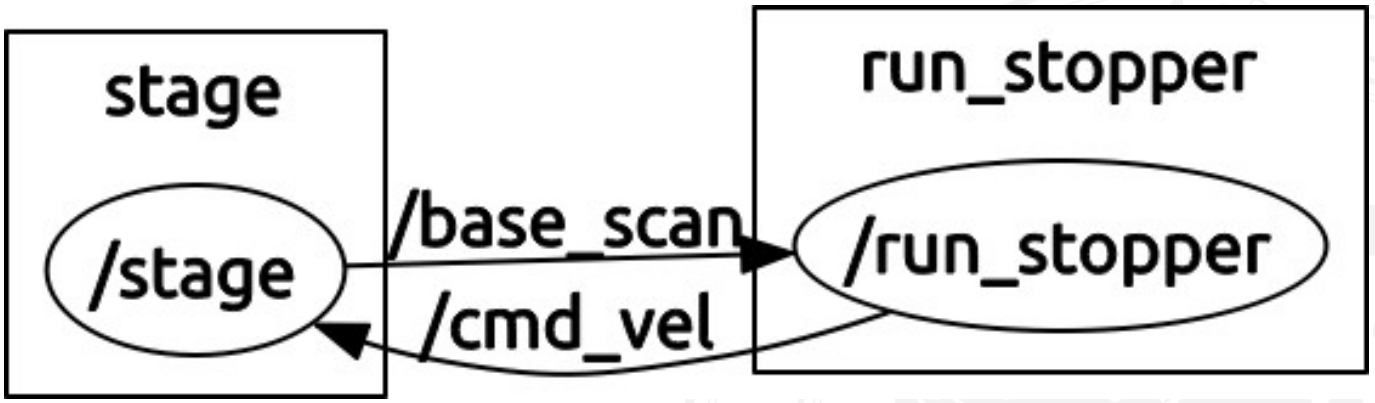
Departamento de Ciencias de la Computación e Inteligencia Artificial

1. Introducción
2. Solución a entrega 1
3. Solución a entrega 2
4. Solución a entrega 3

Los tipos de problemas que se han resuelto en estas prácticas han sido:

- Deambulación por un entorno evitando obstáculos
- Evitación de obstáculos mediante el cálculo de potenciales
- Cálculo de una ruta óptima para llegar a un objetivo mediante potenciales
- Cálculo de la ruta óptima para llegar a un objetivo y evitar obstáculos mediante el algoritmo A*

Esquema de la arquitectura de Módulos/Nodos



Aspectos a destacar sobre la implementación

- Momento en el que girar:

```
ROS_INFO_STREAM("Closest range: " << closestRange);
if (closestRange < MIN_PROXIMITY_RANGE_M) {
    ROS_INFO("Giro!");
    giro();
}
```

- Función girar:

```
void Stopper::giro()
{
    geometry_msgs::Twist twist;
    srand(time(NULL));
    bool direccion = rand()%2;
    twist.angular.x = 0; twist.angular.y = 0; twist.angular.z = direccion?rand():-rand();
    commandPub.publish(twist);
}
```

Implementación opcional del launch

- La implementación para pasar directamente la distancia a la que girar a la hora de llamar al launch ha sido la siguiente:

```
<launch>
    <node name="stage" pkg="stage_ros" type="stageros" args="$(find stage_ros)/world/willow-erratic.
    world"/>
    <arg name="dist" default="0.5"/>
    <node name="run_stopper" pkg="random_walk" type="run_stopper" args="$(arg dist)"/>
</launch>
```

- Por defecto dist = 0.5

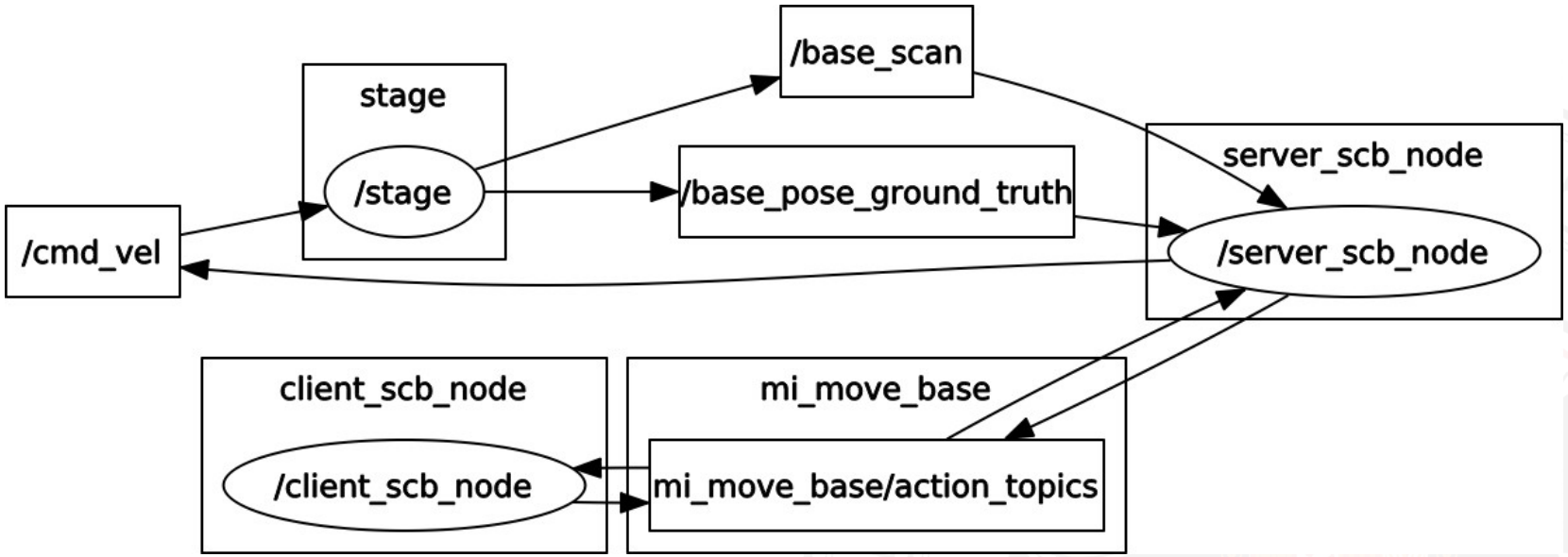
1. Introducción
2. Solución a entrega 1
3. Solución a entrega 2
4. Solución a entrega 3

Explicación de la solución y esquema de la arquitectura de módulos/nodos

- La solución de esta práctica consiste en calcular los potenciales de atracción y repulsión para que nuestro robot evite obstáculos y sea atraído por el objetivo

- Para ello tenemos un cliente que envía la información a realizar, un servidor que a partir de lo solicitado por el cliente actúa de una manera, y un planificador que implementa un planificador local que realiza el cálculo de potenciales y velocidades.

Esquema de la arquitectura de módulos/nodos



- Nuestra solución ha consistido en la implementación de las funciones de cálculo de componentes atractiva y repulsiva, en concreto:
  - Función getOneDeltaRepulsivo:

```
void LocalPlanner::getOneDeltaRepulsivo(Tupla posObst, Tupla &deltaO){
// recibe una posición de un obstáculo y calcula el componente repulsivo para ese obstáculo.
// Devuelve los valores en deltaO.x y deltaO.y
    double d = distancia(posObst, pos);
    double theta = atan2(posObst.y - pos.y, posObst.x - pos.x);

    if (d < CAMPOREP.radius){
        deltaO.x = -9999*cos(theta);
    deltaO.y = -9999*sin(theta);
        return;
        }
    if ( (CAMPOREP.radius <= d) and (d <= (CAMPOREP.spread + CAMPOREP.radius ))){
        deltaO.x = -CAMPOREP.intens*(CAMPOREP.spread + CAMPOREP.radius - d)*cos(theta);
        deltaO.y = -CAMPOREP.intens*(CAMPOREP.spread + CAMPOREP.radius - d)*sin(theta);
        return;
    }
    if (d > (CAMPOREP.spread + CAMPOREP.radius)){
        deltaO.x = deltaO.y = 0;
        return;
    }

}
```

- Nuestra solución ha consistido en la implementación de las funciones de cálculo de componentes atractiva y repulsiva, en concreto:
  - Función setTotalRepulsivo:

```
void LocalPlanner::setTotalRepulsivo(){
//Calcula la componente total repulsiva como suma de las componentes repulsivas para cada obstáculo.
        deltaObst.x = deltaObst.y = 0;
    int tam = posObs.size();
    for (int i=0; i<tam; i++) {
        Tupla d3lta;
        getOneDeltaRepulsivo(posObs[i], d3lta);
        deltaObst.x += d3lta.x;
        deltaObst.y += d3lta.y;
    }
}
```

En esta práctica nuestros principales problemas han sido:

- Cálculo para el potencial repulsivo de un obstáculo:
  - En concreto, en la función getOneDeltaRepulsivo y setDeltaAtractivo nos ha costado ajustar los parámetros para que funcionara correctamente
  - La solución no ha sido otra que la de intentar comprender el problema y el funcionamiento de todo el programa para modificarlo

En esta práctica nuestros principales problemas han sido:

- Implementación correcta de ServerLite.cpp y myPlannerLite:
  - En concreto en la mejora del navegador local que implementaba la velocidad angular y linear para el robot así como la asignación de los parámetros correctos de velocidades l ángulos
  - La solución ha consistido en ajustar los atributos de myPlannerLite correctamente, como por ejemplo aumentando la velocidad linear en 10 veces la de por defecto para que el robot girara más rápidamente.

```
void LocalPlanner::setv_Lineal(){
//calcula la velocidad lineal
    v_lineal =  10*sqrt(delta.x*delta.x + delta.y*delta.y);
    }
```

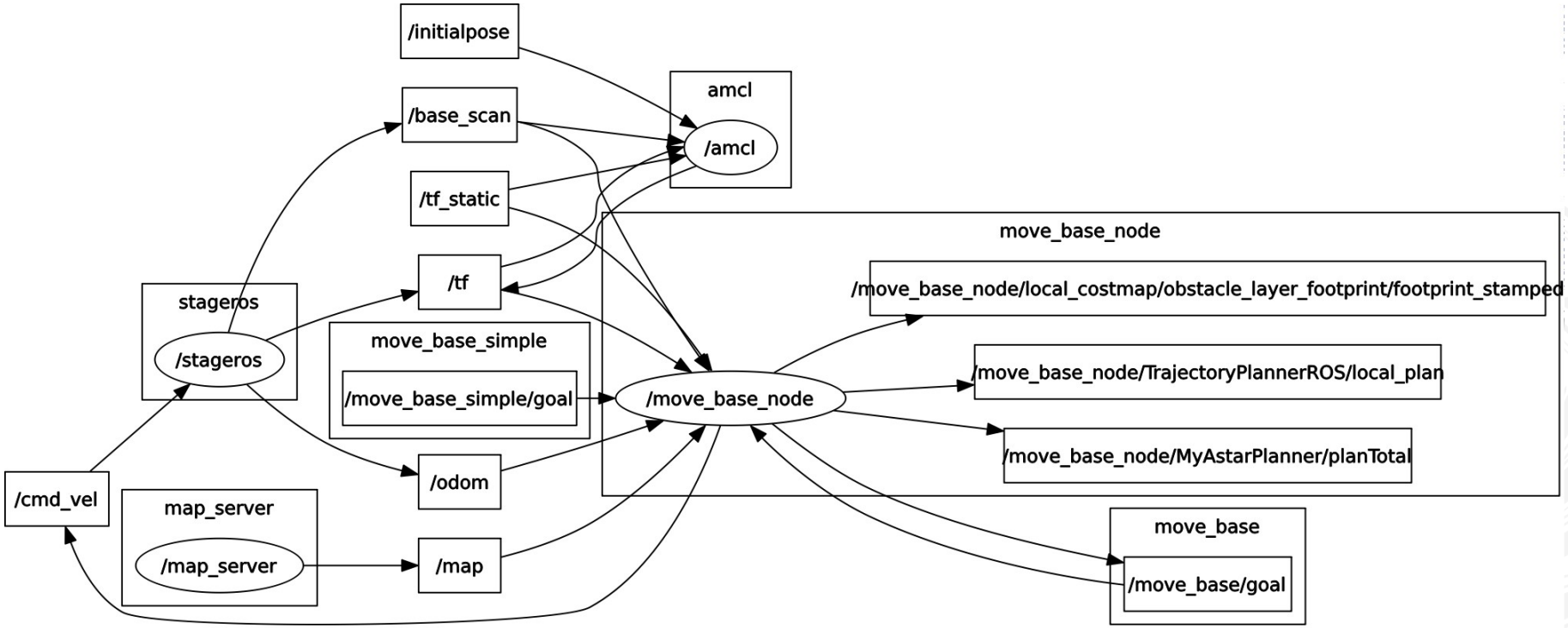
  - Otra solución ha sido la de modificar la distancia a partir de la cual el robot detectaba el obstáculo y giraba, en nuestro caso reduciéndola.

1. Introducción
2. Solución a entrega 1
3. Solución a entrega 2
4. Solución a entrega 3

Explicación de la solución y esquema de la arquitectura de módulos/nodos

- El objetivo de esta práctica es obtener un planificador global que encuentre el mejor camino, sin obstáculos, hacia un objetivo.
- Para ello, se ha utilizado la estructura facilitada del algoritmo A*. De la cual, se ha completado la función principal que maneja las distintas listas.
- Se ha añadido un control sobre el coste del valor “G”, el coste conocido hasta el momento por el planificador global.
- Además, se ha añadido un parámetro interno para cuánta seguridad se tiene en cuenta en cada búsqueda del camino.

Esquema de la arquitectura de módulos/nodos



Las listas de abiertos y cerrados

- La lista de abiertos contiene aquellos nodos por explorar.
- La lista de cerrados contiene aquellos nodos que ya han sido explorados y que no se ha conseguido el objetivo.

- Se comienza con el nodo de la posición del robot.
- Se elige de la lista de abiertos el que tiene menor coste en “f” y se comprueba si es el objetivo, si no lo es, se generan sus vecinos que no sean obstáculos en el mapa y no se encuentren en la lista de abiertos.
- El nodo ya comprobado, se inserta en la lista de cerrados.
- Los vecinos son evaluados, tanto con su “h”, coste estimado, como su “g”, coste conocido. El coste total es la suma de ambos costes, lo que resulta en “f”.
- En el caso de que alguno de los vecinos, sea mejor que algún nodo que se encuentre en la lista de cerrados, lo reemplaza. Manteniendo así sólo la información del mejor.

Acceso al costmap

- Se ha hecho uso de la clase costmap para obtener información del mapa del mundo.

Funciones utilizadas:

- WorldToMap → Convertir las coordenadas del mundo en coordenadas del mapa, celdas.
- IndexToCells → Convertir el índice en coordenadas del mapa, celdas.
- MapToWorld → Convertir coordenadas de celdas en coordenadas del mundo.
- GetCost → Función con dos parámetros, celdas x e y, que devuelve el coste de la celda.

Constantes que devuelve getCost():

- FREE_SPACE = 0; INSCRIBED_INFLATED_OBSTACLE = 253; LETHAL_OBSTACLE = 254; NO_INFORMATION = 255

Heurísticas

- H → Distancia Euclídea entre el punto del camino a evaluar y la posición del robot en ese instante.
- G → Coste del padre más el coste de cada movimiento más el control de seguridad (si procede).
- F → La suma de G y H.

```
//Description: It is used to add the neighbor cells to the open list
/*****************************************************************************/
void MyastarPlanner::addNeighborCellsToOpenList(list<coupleOfCells> & OPL, vector <unsigned int> neighborCells, unsigned int parent,
{
        vector <coupleOfCells> neighborsCellsOrdered;
        unsigned int mx, my, theta = 0.0; //theta --> Proporción de reduccion de la seguridad
        double wx, wy;

        //ROS_INFO("Total vecinos: %d", neighborCells.size());
        for(uint i=0; i< neighborCells.size(); i++)
        {
          coupleOfCells CP;
          CP.index=neighborCells[i]; //insert the neighbor cell
          CP.parent=parent; //insert the parent cell

          //CALCULAR el gCost ----------------------------------------------------------------
          CP.gCost=gCostParent+1;//getMoveCost(parent,neighborCells[i]); --> esto solo calcula la distancia euclidea, que es una esti
          //para calcularlo, primero se ha de pasar el indice a celda
          costmap_->indexToCells(CP.index, mx, my);
          //Pasar de celda a coordenadas del mundo
          costmap_->mapToWorld(mx, my, wx, wy);
          //Acumular coste segun donde se encuentre la celda
          CP.gCost += footprintCost(wx, wy, theta);

          //calculate the hCost: Euclidian distance from the neighbor cell to the goalCell
          CP.hCost=calculateHCost(neighborCells[i],goalCell);
          //calculate fcost

          CP.fCost=CP.gCost+CP.hCost;
         // neighborsCellsOrdered.push_back(CP);
          OPL.push_back(CP);
        }
      }
```

Función footprintCost usada para obtener parte del coste sobre G

```
// Si l = sqrt(r^2 + r^2) --> l/sqrt(2) = radio
double lado = sqrt((pow(footprint[0].x - footprint[1].x,2))+pow(footprint[0].y - footprint[1].y, 2));
double radio = lado/(sqrt(2)) * theta_i;// Theta porcentaje de reduccion en radio

if(radio==0)
    radio = lado/2;
//Se comprueba que las casillas adyacentes al vecino no sean un obstaculo
for (int x=-1;x<=1;x++)
{
    for (int y=-1; y<=1;y++){
        //Pasar las coordenadas del mundo a coordenadas en el mapa, para la posicion vecina
        costmap_->worldToMap(x_i+radio*x, y_i+radio*y, cell_mx, cell_my);
        //ROS_INFO("Punto cell: x,y: %d, %d", cell_mx, cell_my);
        //check whether the index is valid
        if ((cell_mx>=0)&&(cell_mx < costmap_->getSizeInCellsX())&&(cell_my >=0 )&&(cell_my < costmap_->getSizeInCellsY()))
        {
            //Si hay un obstaculo en la casilla
            if(costmap_->getCost(cell_my,cell_my) == costmap_2d::LETHAL_OBSTACLE   && (!(x==0 && y==0))){
              seguridad += 0.5;
            }//Si hay un obstaculo en parte de la celda... ??
            else if(costmap_->getCost(cell_my,cell_my) == costmap_2d::INSCRIBED_INFLATED_OBSTACLE   && (!(x==0 && y==0))){
              seguridad += 0.2;
            }
            else if(costmap_->getCost(cell_my,cell_my) == costmap_2d::FREE_SPACE   && (!(x==0 && y==0))){
              seguridad -= 0.1;
            }
        }
    }
}

return seguridad;
}
```

Función G con Seguridad

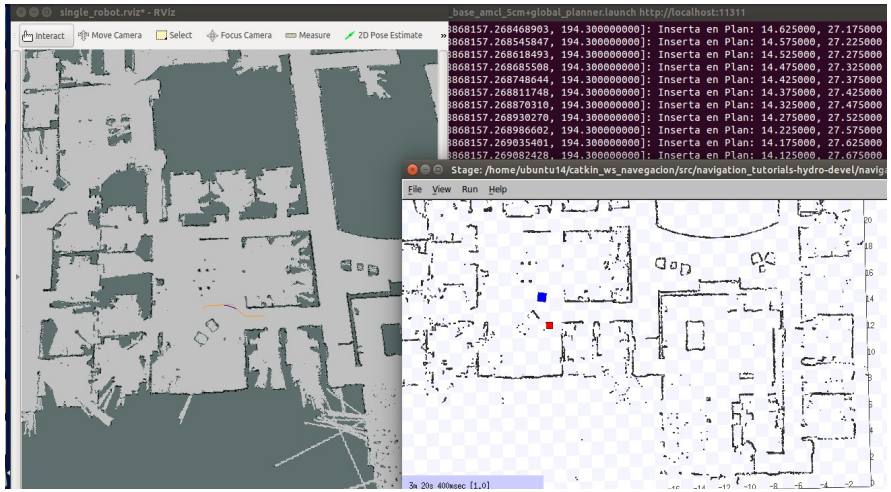
- lado → se calcula la distancia Euclídea de los dos puntos del robot que marcan su parte trasera para obtener así el lado, igual a 0.64.
- radio → depende del parámetro “theta”, que será proporcional al lado. Theta, será en este caso un porcentaje.

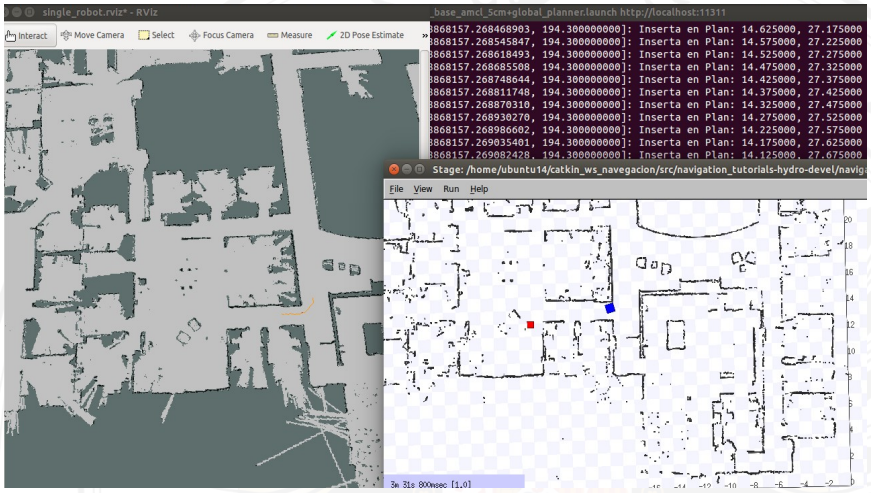
Pruebas parámetros:

- Cirscunscrita → la idea es que el radio sea como un circulo interno para el cuadrado del robot, sin tener en cuenta la “cabeza”. Con un valor theta = 0.0, calcula el radio = lado/2. Radio=0.34.
- Inscrita → Se ha probado con el theta=0.20 siendo el radio=theta*lado. Lo cual, es un 0.128 mayor que el radio anterior.

Caminos encontrados

- El siguiente camino muestra el mapa del mundo tanto en el visualizador rviz como en el recorrido real del robot.
- Se puede observar como una vez planificado de forma global el camino, lo recorre hasta el objetivo previamente encontrado con el algoritmo A*.





- El siguiente camino encontrado, se muestra con el mapa global en el visualizador.
- Se puede observar como hay muchos caminos que no se podrían planificar con el algoritmo A*. Por ejemplo, el pasillo de enfrente a la posición inicial del robot.
- El resultado de la planificación es satisfactoria.