

Escuela Técnica Superior de Ingeniería
Universidad de Huelva
Grado en Ingeniería Informática



Robótica

Memoria de Prácticas

Raúl Jiménez Suárez

Wadadi Sidelgaum Limam



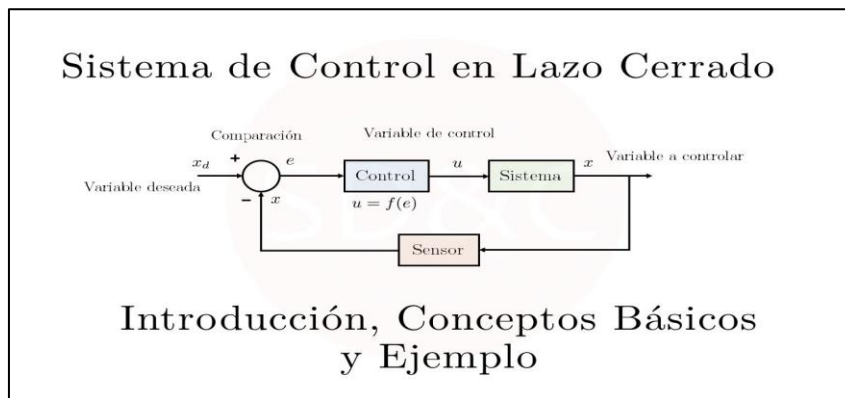
Fecha: 22/12/2024

Contenido

Práctica 1: Introducción al Sistema de Control y Sensores del EV3	3
Ejercicio1	3
Ejercicio2	4
Ejercicio3	5
Práctica 2: Control de movimiento Simulados de un Manipulador de 6 grados de Libertad ...	7
Procedimiento	7
Primera Parte: Planificación de Tareas de "Pick and Place"	8
1. Inicialización del Sistema y Conexión a RoboDK	9
2. Configuración de la Posición y Orientación de las Piezas	9
3. Definición de la Configuración de la Pinza	9
4. Cinemática Inversa y Configuración del Bucle de Recogida y Colocación de Piezas.....	9
5. Aproximación y Recogida de la Pieza ("Pick")	10
6. Levantamiento de la Pieza	10
7. Aproximación y Colocación de la Pieza ("Place")	10
8. Levantamiento y Actualización de la Posición de la Pieza	11
Segunda Parte	11
Práctica 3: Introducción al entorno de ROS	13
Práctica 4. ROS: RVIZ y ROSBAG	16
Práctica 5: ROS: TOPICS.....	16
Práctica 6: ROS: GAZEBO Y TURTLEBOT	17
Práctica 7: ROS: Maestro Remoto y TURTLEBOT Real	22

Práctica 1: Introducción al Sistema de Control y Sensores del EV3

El objetivo de esta práctica fue la familiarización con el Robot y su estructura, realizando programas sencillos, el programa realizado fue uno para mover la cabeza del robot a través del motor de la cabeza. Posteriormente, se añadió controlar esta rotación con un Sensor de Contacto.



En esta práctica se introdujeron los controladores y el encoder. Se estudiaron 3 tipos de controladores:

- Proporcional (P)
- Proporcional-Integral (PI)
- Proporcional-Integral-Derivativo (PID)

La práctica se divide en las siguientes partes:

Ejercicio1

El ejercicio pide realizar un Script que realice medidas con el sensor de ultrasonido y acumule las medidas en una matriz. Deberá registrarse el tiempo de forma que al final pueda realizarse una gráfica de la evolución temporal de la medida. Experimente variando la distancia de los objetos frente al sensor.

Para la resolución se ha utilizado como base el script del controlador proporcionado en Moodle:

```
%Definición de los sensores

Pulsador=touchSensor(mi_Robot,2); %pulsador
Sonar = sonicSensor(mi_Robot,4); %definición del sonar

N=10;
tiempo= zeros(1,N);
distancias= zeros(1,N);
t0=1;
for i=1:N
    tiempo(i)=t0;
    distancias(i)=readDistance(Sonar);
    pause(1);
    t0=t0+1;
end
```

En primer lugar, se conecta al robot LEGO EV3 mediante USB con `mi_Robot=legoev3('usb')`, después se definen los motores conectados a los puertos B y C como `motor_rueda_B` y `motor_rueda_C`, y se inicializan ambos motores con `start(motor_rueda_B)` y `start(motor_rueda_C)`, luego definimos un sensor táctil en el puerto 2 (Pulsador) y un sensor ultrasónico en el puerto 4 (Sonar), finalmente, para la medición de distancias hemos definido dos arrays (`tiempo` y `distancias`) para almacenar los tiempos y las lecturas del sensor ultrasónico. En un bucle de 10 iteraciones (`for`).

Ejercicio2

En este apartado se pide un script que simula y controla el movimiento de un robot LEGO EV3 conectado mediante USB, que lea el ángulo girado (manualmente) por parte del eje del motor A y haga girar el mismo ángulo el motor sobre el que está situado el sensor de ultrasonido. El programa ha de comportarse de forma que parezca que la cabeza del robot “sigue” el movimiento del motor A.

Para el giro del motor se ha fijado un valor de referencia (90°), finalmente, el controlador proporcional se encarga de ajustar la velocidad del motor según el error entre la posición actual y la referencia, el código de la resolución es el siguiente:

```
while readTouch(Pulsador)==0
    %actualización
    k=k+1;
    t(k)=toc(tstart);
    referencia(k)=90;%200*t(k);
    Giro_Cabeza(k)=double(readRotation(motor_Cabeza));
    error(k)= referencia(k) - Giro_Cabeza(k);
    %-----
    % Control Proporcional
    %-----
    Ganancia_p = 0.69184;%pidtuner Matlab
    Potencia_Cabeza=int8(Ganancia_p*error(k));
    if Potencia_Cabeza>100
        Potencia_Cabeza=100;
    elseif Potencia_Cabeza<-100
        Potencia_Cabeza=-100;
    end
    motor_Cabeza.Speed=int8(Potencia_Cabeza);
    Giro_Cabeza(k)=readRotation(motor_Cabeza);
end
motor_Cabeza.Speed=0;
stop(motor_Cabeza);
plot(t,Giro_Cabeza(k), 'b'),hold on
plot(t,error, 'r'), hold on
plot(t,referencia, 'g')
```

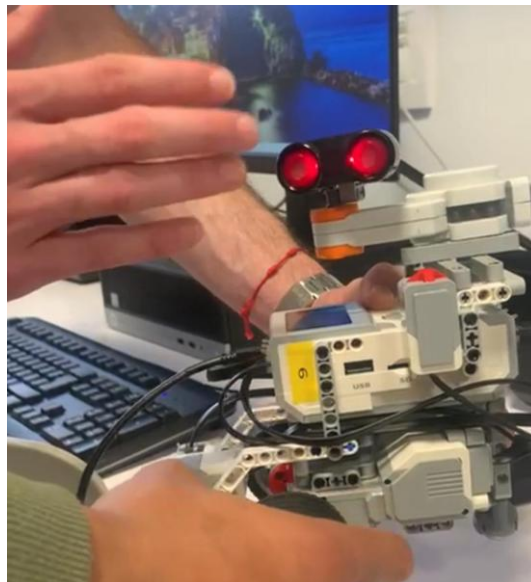
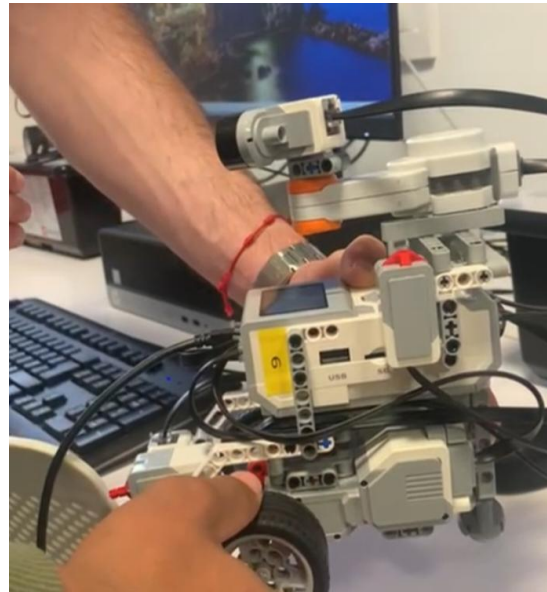
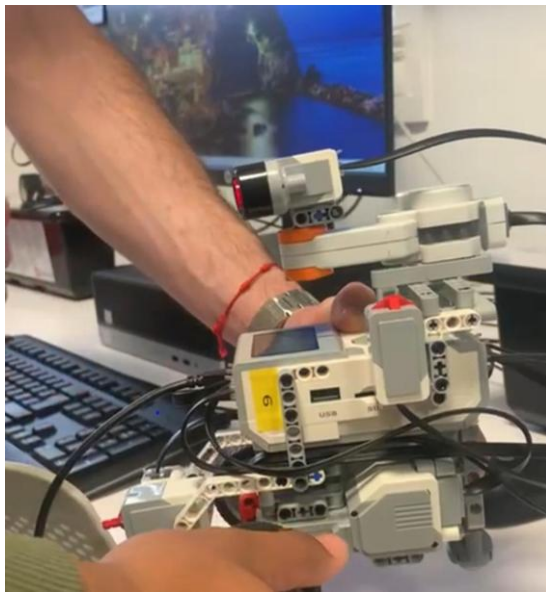
Ejercicio3

Utilizando el mismo controlador, de la actividad anterior, realice un Script que lea el ángulo girado (manualmente) por parte del eje del motor A y haga girar el mismo ángulo el motor sobre el que está situado el sensor de ultrasonido. El programa ha de comportarse de forma que parezca que la cabeza del robot “sigue” el movimiento del motor A.

```
while readTouch(Pulsador)==0
    %actualización
    k=k+1;
    t(k)=toc(tstart);
    Giro_Cabeza(k)=double(readRotation(motor_Cabeza));
    distancia(k)=readDistance(sonar);
    referencia(k)=signal_vf_v2( t(k), periodo, delay, Amplitud );
    error(k)=referencia(k)-Giro_Cabeza(k);
    %-----
    % Control Proporcional
    %-----
    Ganancia_p = 0.69184;%pidtuner Matlab
    Potencia_Cabeza=int8(Ganancia_p*error(k));
    if Potencia_Cabeza>100
        Potencia_Cabeza=100;
    elseif Potencia_Cabeza<-100
        Potencia_Cabeza=-100;
    end
    motor_Cabeza.Speed=int8(Potencia_Cabeza);
    Giro_Cabeza(k)=readRotation(motor_Cabeza);
end
motor_Cabeza.Speed=0;
stop(motor_Cabeza);
plot(t,Giro_Cabeza, 'b'),hold on
plot(t,error, 'r'), hold on
plot(t,referencia, 'g')
```

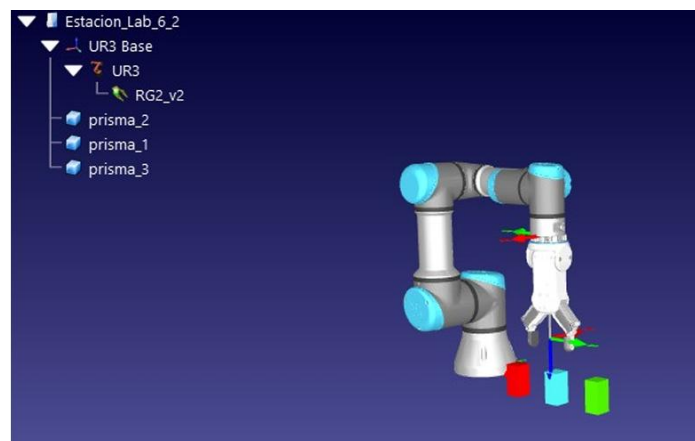
En este ejercicio el controlador proporcional se encarga de ajustar la velocidad del motor según el error entre la posición angular del motor y la referencia, de forma que se registran el tiempo, la posición del motor, la distancia medida y el error durante el proceso, finalmente los resultados se grafican, mostrando la evolución de la posición, el error y la referencia a lo largo del tiempo, permitiendo evaluar el comportamiento del controlador proporcional frente a una referencia variable (Se nos ha olvidado guardar las graficas de los resultados).

A continuación, se muestran fotografías de la implementación del Robot Lego EV3 en el laboratorio de prácticas:



Práctica 2: Control de movimiento Simulados de un Manipulador de 6 grados de Libertad

La práctica 2 se centra en el control de movimientos simulados de un manipulador con seis grados de libertad utilizando MATLAB y RoboDK. El objetivo principal de esta práctica es familiarizarse con la programación de manipuladores robóticos mediante la simulación en RoboDK y el control desde MATLAB. Se debe lograr que el robot simule una tarea de "pick and place", moviéndose entre distintas posiciones para recoger y soltar bloques en ubicaciones específicas. Esta actividad requiere el uso de funciones específicas de RoboDK para controlar las articulaciones del robot y su herramienta de agarre.



Procedimiento

1. Inicialización de la Conexión con RoboDK:

- Se establece la conexión entre MATLAB y RoboDK mediante la función `Iniciacion('RBDK', codigo)`, que devuelve un identificador necesario para el control del robot en la simulación.
- Este identificador es crucial para las llamadas posteriores a las funciones que permiten mover el robot y activar su pinza de agarre.

2. Control de Movimientos Articulares:

- Se utiliza la función `MoveJ_Robot_lab(angulos, aceleracion, velocidad, Identificador, codigo)`, la cual permite que el manipulador se mueva a

posiciones específicas configurando los ángulos de cada articulación. Los valores de velocidad y aceleración ajustan la rapidez y suavidad de los movimientos.

- Para posicionar correctamente el robot, cada articulación del manipulador se ajusta en radianes, permitiendo que el robot llegue a una **configuración de aproximación** antes de posicionarse sobre el objeto a manipular.

3. Control de la Herramienta de Agarre (Pinza):

- La función `RG2_lab(100, Identificador, codigo)` se encarga de accionar la pinza del robot. Cuando el valor de abertura es menor que 100, la pinza se cierra para agarrar un bloque; cuando el valor es igual o superior a 100, la pinza se abre para soltar el objeto.
- Esta herramienta permite que el robot ejecute la tarea de “pick and place”, manipulando bloques en el escenario de RoboDK.

4. Movimientos Programados y Retorno a la Posición Inicial:

- Después de manipular el objeto, el robot puede regresar a su **posición inicial** utilizando la función `Ready_lab(Identificador, codigo)`, lo que facilita la realización de múltiples tareas de recogida y colocación en el escenario simulado.

Primera Parte: Planificación de Tareas de "Pick and Place"

A partir de lo aprendido en los movimientos básicos, se realiza una secuencia de configuraciones para que el robot se acerque a un bloque, lo agarre, y luego lo desplace a la ubicación deseada. Para ello, se determina una **configuración de aproximación** y una **configuración de manipulación**.

En este ejercicio, se ajustan las posiciones y la apertura de la pinza para asegurar que el bloque sea recogido y colocado de manera adecuada, simulando un ciclo de trabajo común en aplicaciones industriales de "pick and place", a continuación, se detalla cada sección y el propósito de cada bloque del código implementado.

1. Inicialización del Sistema y Conexión a RoboDK

El código comienza configurando la conexión entre MATLAB y RoboDK mediante la función `Iniciacion`. Esta conexión es necesaria para que MATLAB envíe comandos al simulador y controle el robot UR3.

```
codigo=1;  
Identificador = Iniciacion('RBDK',codigo);
```

2. Configuración de la Posición y Orientación de las Piezas

El código establece la **posición y orientación** de las piezas que serán manipuladas. Para ello, se utiliza una transformación de desplazamiento y rotación para especificar la posición y orientación de las piezas en el espacio. Las matrices de transformación definen la ubicación inicial de cada pieza.

```
%configuración de la pieza  
posicion=[40 -10 0];  
alfa=0; beta=0; gamma=0;  
  
matriz_pieza=Desplazamiento(posicion(1), posicion(2), posicion(3))*Rotacionz(alfa)*Rotaciony(beta)*Rotacionx(gamma);
```

3. Definición de la Configuración de la Pinza

La pinza del robot se configura mediante la `matriz_agarre`, que define la orientación específica necesaria para agarrar la pieza. En este caso, se utiliza una rotación sobre el eje X.

```
matriz_agarre = Desplazamiento(0,0,4)*Rotacionx(pi);
```

4. Cinemática Inversa y Configuración del Bucle de Recogida y Colocación de Piezas

Para cada pieza en la tarea de "pick and place", el robot ejecuta una secuencia de movimientos. En el bucle for, se define el número de piezas que se moverán y se actualizan las posiciones para cada iteración.

```
numero_piezas=3;
for i=1:numero_piezas

    G_T_Pinza = matriz_pieza*matriz_agarre;
    G_T_Pinza1 = matriz_pieza1*matriz_agarre;
```

5. Aproximación y Recogida de la Pieza ("Pick")

El robot se aproxima a la posición de la pieza mediante la transformación de desplazamiento en el eje Z, bajando 5 unidades. Se aplica la función de cinemática inversa `inv_kinema_ur3_new` para calcular los ángulos necesarios para alcanzar esta posición, y el comando `MoveJ_Robot_lab` mueve el robot.

```
%aproximacion pick

[q1 q2 q3 q4 q5 q6] = inv_kinema_ur3_new(G_T_Pinza*Desplazamiento(0,0,-5), codo, avance, simetrico);
MoveJ_Robot_lab([q1 q2 q3 q4 q5 q6], 1, 1,Identificador, codigo )

%pick

[q1 q2 q3 q4 q5 q6] = inv_kinema_ur3_new(G_T_Pinza, codo, avance, simetrico);
MoveJ_Robot_lab([q1 q2 q3 q4 q5 q6], 1, 1,Identificador, codigo)
```

Una vez en la posición de recogida exacta, el robot acciona la pinza para agarrar la pieza, usando `RG2_lab` con un valor de 50 para cerrar la pinza.

```
RG2_lab(50, Identificador, codigo);
```

6. Levantamiento de la Pieza

Después de recoger la pieza, el robot eleva la pinza para despegar la pieza de su posición actual. Se calcula nuevamente la cinemática inversa para desplazar la pinza hacia arriba.

```
%despeje pick

[q1 q2 q3 q4 q5 q6] = inv_kinema_ur3_new(Desplazamiento(0,0,5)*G_T_Pinza, codo, avance, simetrico);
MoveJ_Robot_lab([q1 q2 q3 q4 q5 q6], 1, 1,Identificador, codigo)
```

7. Aproximación y Colocación de la Pieza ("Place")

Para colocar la pieza en su nueva posición, el robot se aproxima al destino utilizando `G_T_Pinza1` y se baja en el eje Z una distancia calculada para cada iteración ($6 * i$). Esto

permite que cada pieza se coloque a una altura diferente, en función del número de la pieza en el bucle, luego el robot abre la pinza con un valor de 120 para soltar la pieza.

```
%aproximacion place
[q1 q2 q3 q4 q5 q6] = inv_kinema_ur3_new(Desplazamiento(0,0,6*i)*G_T_Pinza1, codo, avance, simetrico);
MoveJ_Robot_lab([q1 q2 q3 q4 q5 q6], 1, 1, Identificador, codigo)

%place
[q1 q2 q3 q4 q5 q6] = inv_kinema_ur3_new(G_T_Pinza1*Desplazamiento(0,0,(-6)*(i-1)), codo, avance, simetrico);
MoveJ_Robot_lab([q1 q2 q3 q4 q5 q6], 1, 1, Identificador, codigo)

RG2_lab(120, Identificador, codigo)
```

8. Levantamiento y Actualización de la Posición de la Pieza

Tras soltar la pieza, el robot se eleva nuevamente en el eje Z y actualiza matriz_pieza para posicionar la siguiente pieza en la posición correcta para la siguiente iteración del bucle.

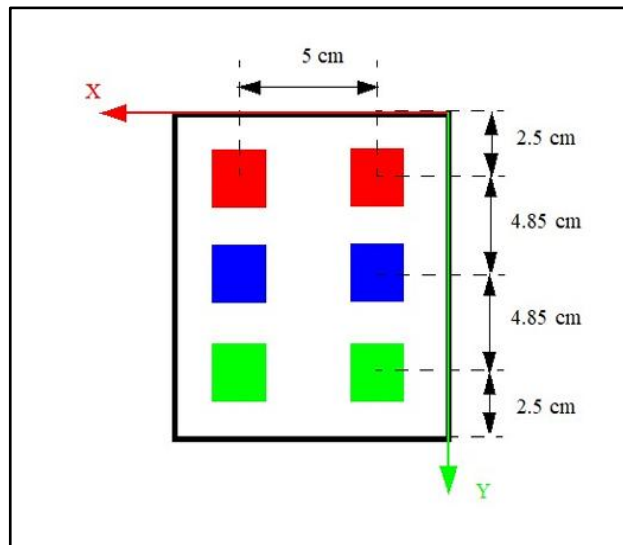
```
%despeje place
[q1 q2 q3 q4 q5 q6] = inv_kinema_ur3_new(G_T_Pinza*Desplazamiento(0,0,(-6)*(i)), codo, avance, simetrico);
MoveJ_Robot_lab([q1 q2 q3 q4 q5 q6], 1, 1, Identificador, codigo)

%actualizacion
matriz_pieza = matriz_pieza*Desplazamiento(-10,0,0);
```

Segunda Parte

Este ejercicio pide implementar un programa que simule el proceso de empaquetado de seis piezas en una caja utilizando el robot UR3, el objetivo del programa es establecer una rutina que permita al robot recoger y colocar cada pieza por separado.

En la siguiente figura se muestra las posiciones iniciales de las cajas y las mediadas de separación entre ellas:



Para la resolución de este ejercicio se ha usado la misma configuración que el anterior, modificando las posiciones y los ángulos de Euler de las piezas y la caja:

```
%configuración de la pieza
posicion=[20 -10 0];
alfa=0; beta=0; gamma=-(pi/4);
matriz_caja=Desplazamiento(posicion(1), posicion(2), posicion(3))*Rotacionz(alfa)*Rotaciony(beta)*Rotacionz(0);

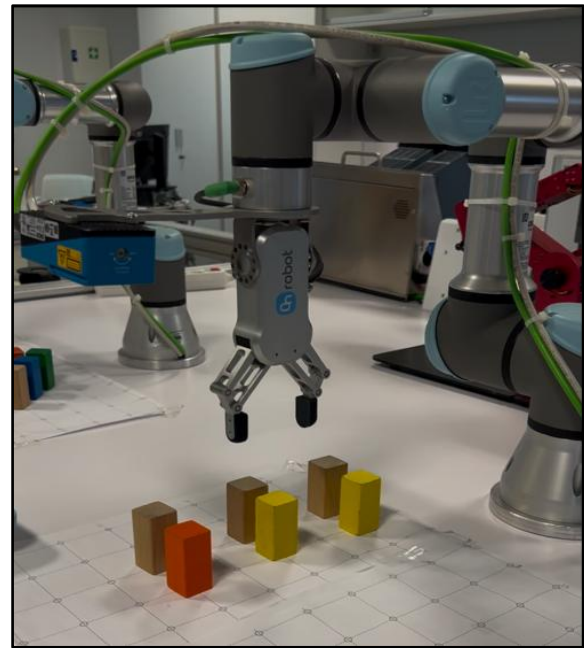
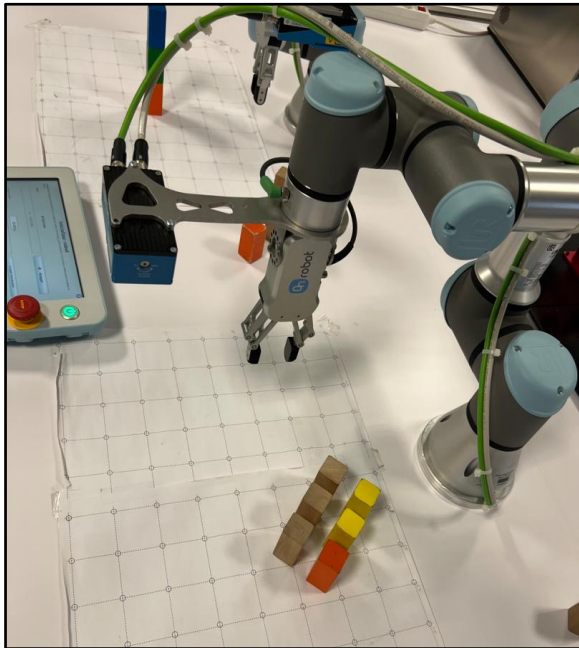
%configuración de la pieza
posicion=[24 5 0];
matriz_pieza1=Desplazamiento(posicion(1), posicion(2), posicion(3))*Rotacionz(0)*Rotaciony(0)*Rotacionz(gamma);
matriz_agarre = Desplazamiento(0,0,4)*Rotacionz(pi/2)*Rotaciony(pi);
```

Además, se ha añadido al final del for un bucle if donde actualizamos las posiciones de las matrices del place(caja) y el pick(pieza):

```
%actualizacion
if mod(i,3)==0
    matriz_caja=matriz_caja*Desplazamiento(-20,-5,0);
    matriz_pieza1=matriz_pieza1*Desplazamiento(5,-9.7,0);
else
    matriz_caja=matriz_caja*Desplazamiento(10,0,0);
    matriz_pieza1=matriz_pieza1*Desplazamiento(0,4.85,0);
end
end
```

De manera que si la pieza actual es la primera de la segunda fila se modifica el desplazamiento del pick y el place a la posición adecuada, y así las siguientes piezas se sitúan a un desplazamiento de 4.85 cm.

Finalmente, se muestran fotografías de la implementación en el laboratorio de prácticas.



Práctica 3: Introducción al entorno de ROS

En esta práctica se describen y trabajan los conceptos básicos del sistema operativo ROS, las principales instrucciones básica para ROS, así como la introducción a la programación de los robots móviles en el entorno de ROS. En esta sesión vimos algunos conceptos como:

- **Nodos:** Son programas que realizan cálculos. ROS es modular, por lo que utiliza un nodo para cada tarea.
- **Maestro:** El maestro es la parte más importante de ROS ya que, sin él, los demás nodos no podrían comunicarse entre ellos.
- **Mensajes:** Es el elemento que utilizan todos los nodos para comunicarse entre ellos.
- **Topics:** Son un tipo de mensaje que utilizan un sistema de transporte basado en la suscripción y publicación.

Para utilizar ROS ejecutaremos la máquina virtual que nos proporcionan en Moodle que tiene instalado el sistema operativo Ubuntu y el software necesario para poder ejecutar las instrucciones de ROS con su versión Indigo.

Para comenzar a usar ROS debemos lanzar primero el nodo maestro mediante la instrucción con roscore.

Para lanzar la ejecución de nuevos nodos utilizaremos la instrucción con rosrn. En primer lugar, iniciaremos el simulador del robot turtlebot con la instrucción:

```
turtlebot@rosindigo :~$ rosrn turtlesim turtlesim_node
```

Podemos ejecutar un nodo llamado turtle teleop key que permite controlar el robot mediante las flechas del teclado. Lo ejecutamos utilizando:

```
turtlebot@rosindigo :~$ rosrn turtlesim turtle_teleop_key
```

También vimos que podemos consultar la información que publican los topics en ROS con el comando:

```
turtlebot@rosindigo :~$ rostopic list
```

```
turtlebot@rosindigo :~$ rostopic info / turtle1 / pose
```

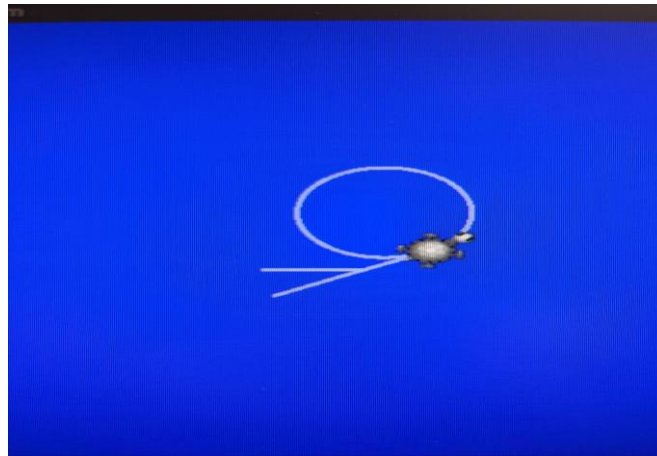
```
turtlebot@rosindigo :~$ rostopic echo / turtle1 / pose
```

Con los comandos anteriores podemos ver cómo cambia la posición de la tortuga en el simulador.

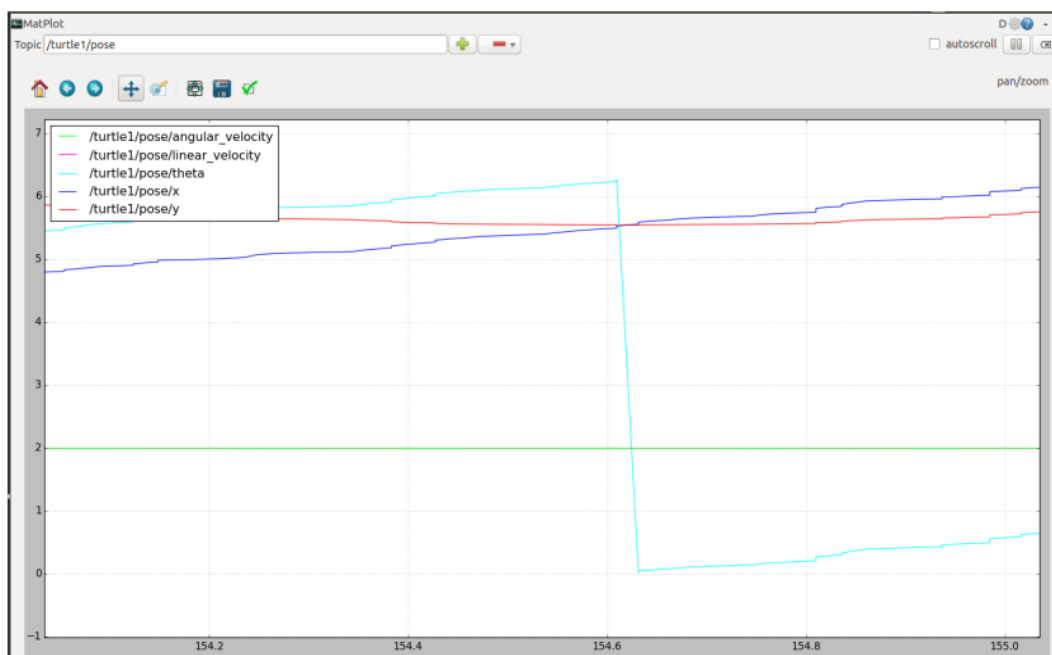
También podemos hacer movimientos predefinidos publicando en el nodo de velocidad del simulador:

```
turtlebot@rosindigo :~$ rostopic pub -r 1 / turtle1 / cmd_vel geometry_msgs / Twist  
'[2.0 ,0.0 ,0.0] ' '[0.0 ,0.0 ,2] '
```

Con la instrucción anterior haríamos una circunferencia de radio 1.



Por último, visualizamos las variables de estado del robot utilizando rqt plot.



Práctica 4. ROS: RVIZ y ROSBAG

En esta práctica se utilizará la herramienta rviz, un software de visualización en 3 dimensiones diseñado para ROS. Se ejecuta como un nodo del entorno, pero requiere ciertas condiciones para representar correctamente los elementos.

Se puede representar el simulador del robot en rviz. Para ello se utilizan nodos llamados broadcasters. Se utilizará el paquete TF para mostrar la representación del robot. Seguiremos los siguientes pasos:

1. Lanzar el maestro ROS:

```
turtlebot@rosindigo :~$ rosrunc turtlesim turtlesim_node
```

2. Lanzar el nodo de control del turtlebot mediante teclado:

```
turtlebot@rosindigo :~$ rosrunc turtlesim turtle_teleop_key
```

3. Crear un nodo broadcaster para traducir los movimientos de la tortuga a un sistema de coordenadas global.

```
turtlebot@rosindigo :~$ rosrunc turtle_tf turtle_tf_broadcaster " turtle1 "
```

4. Iniciar rviz:

```
turtlebot@rosindigo :~$ rosrunc rviz rviz -d `rospack find turtle_tf`/rviz/ turtle_rviz .  
rviz
```

Si desplazamos la tortuga con el teclado veremos cómo se mueve la representación en rviz.

Práctica 5: ROS: TOPICS

El enlace del video de la explicación es el siguiente: [Video.mp4](#)

Práctica 6: ROS: GAZEBO Y TURTLEBOT

En esta práctica hemos implementado un controlador en ROS, similar al que hemos implementado en las clases de teoría para que el robot turtlebot siga una trayectoria predefinida dentro del simulador Gazebo.

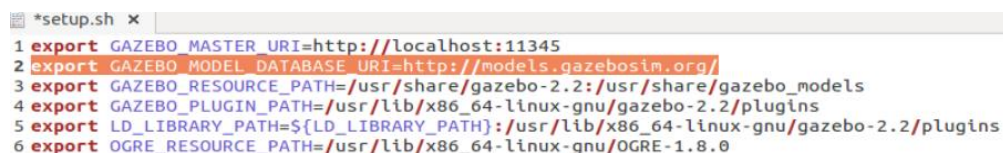
El desarrollo del controlador se divide en las siguientes partes:

1. Para poder cargar del robot dentro del escenario, se ha modificado el fichero “setup.sh”, para ello nos situamos desde la terminal en la carpeta adecuado y ejecutamos el comando gedit para poder editar el fichero:



```
turtlebot@rosindigo: /usr/share/gazebo
turtlebot@rosindigo: /usr/share/gazebo 67x24
turtlebot@rosindigo: /usr/share/gazebo$ cd /usr/share/gazebo
turtlebot@rosindigo: /usr/share/gazebo$ sudo gedit setup.sh
```

Modificamos la variable GAZEBO_MODEL_DATABASE_URI:



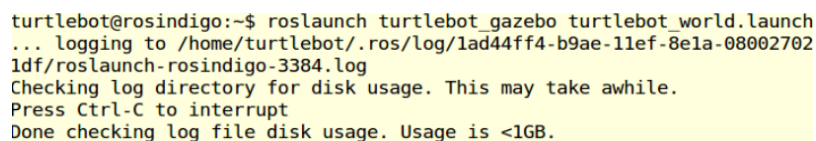
```
*setup.sh x
1 export GAZEBO_MASTER_URI=http://localhost:11345
2 export GAZEBO_MODEL_DATABASE_URI=http://models.gazebosim.org/
3 export GAZEBO_RESOURCE_PATH=/usr/share/gazebo-2.2:/usr/share/gazebo_models
4 export GAZEBO_PLUGIN_PATH=/usr/lib/x86_64-linux-gnu/gazebo-2.2/plugins
5 export LD_LIBRARY_PATH=${LD_LIBRARY_PATH}:/usr/lib/x86_64-linux-gnu/gazebo-2.2/plugins
6 export OGRE_RESOURCE_PATH=/usr/lib/x86_64-linux-gnu/OGRE-1.8.0
```

2. Una vez modificado el fichero setup, cargamos el mundo vacío sin obstáculos con el siguiente comando:



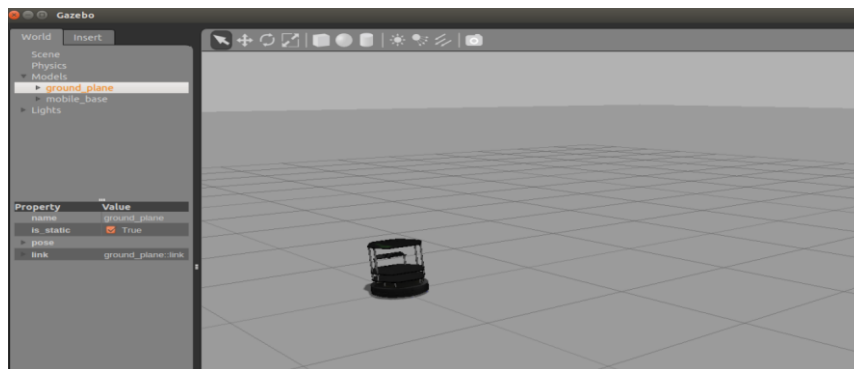
```
turtlebot@rosindigo: ~
turtlebot@rosindigo: ~ 73x24
turtlebot@rosindigo:~$ export TURTLEBOT_GAZEBO_WORLD_FILE=/opt/ros/indigo
/share/turtlebot_gazebo/worlds/empty.world
```

Después ejecutamos la siguiente línea para cargar el entorno de simulación y el modelo del robot:

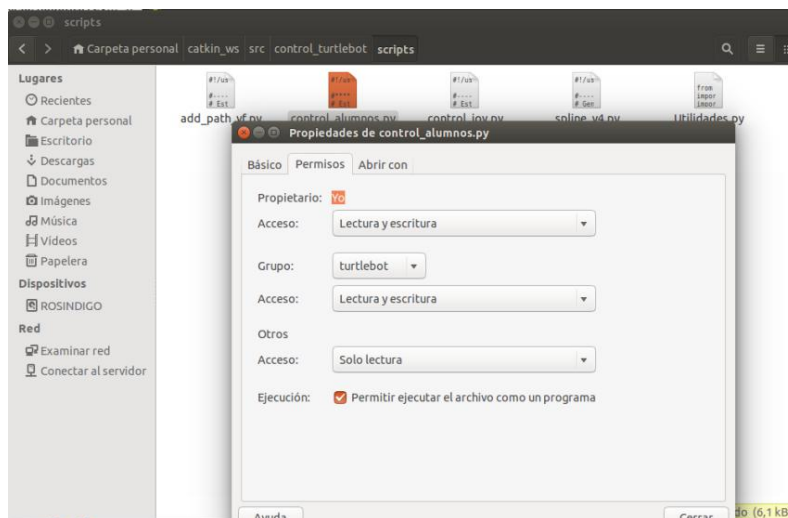


```
turtlebot@rosindigo:~$ roslaunch turtlebot_gazebo turtlebot_world.launch
... logging to /home/turtlebot/.ros/log/1ad44ff4-b9ae-11ef-8e1a-08002702
1df/roslaunch-rosindigo-3384.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.
```

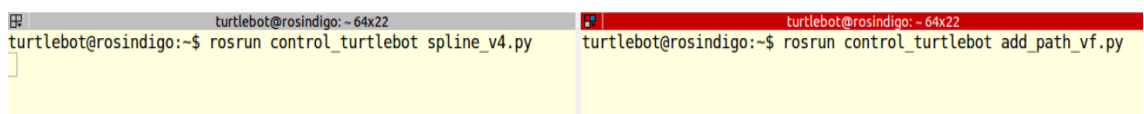
A continuación, se muestra una captura del programa gazebo con el robot:



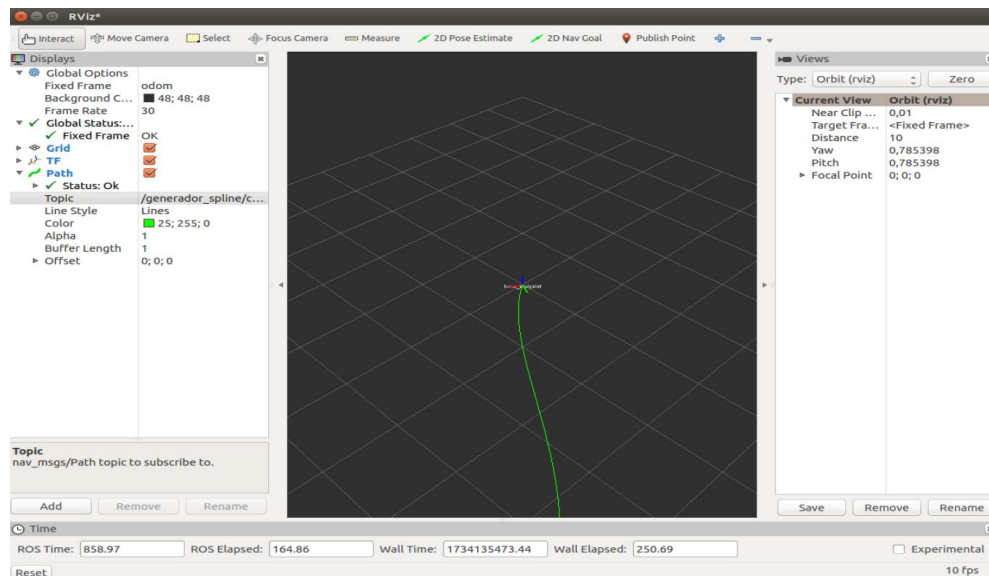
3. Descargamos los ficheros de ayuda de la práctica y los movemos al directorio `src/scripts/`, y comprobamos que todos los ficheros tienen permiso de administración.



4. Después en dos terminales distintas ejecutamos el los ficheros `add_path_vf` y `spline_v4` :

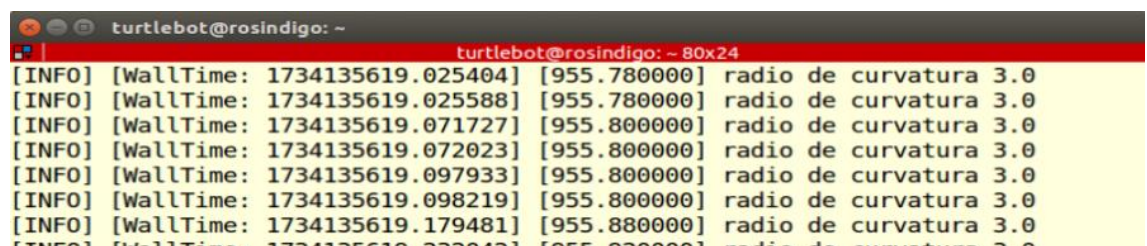


5. Abrimos el rviz, modificamos el modo a `odom` y después le añadimos un TF que corresponde al robot, `path` le asignamos como topic el camino que se encuentra en `/generador_spline/camino`:

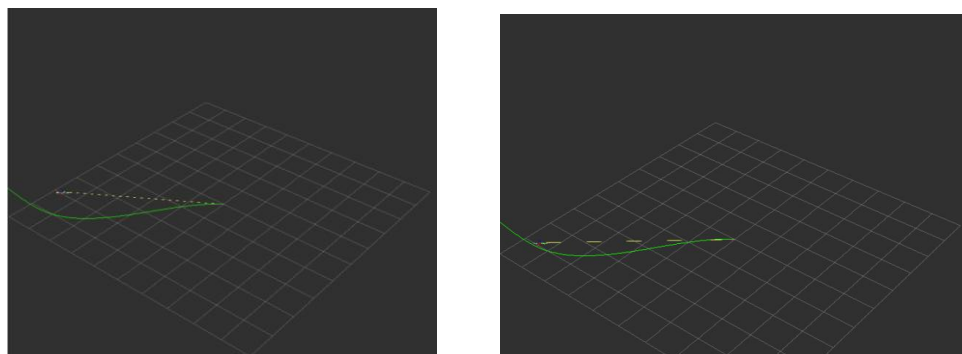


6. Ejecutamos en otra terminal el fichero control_alumnos.

Como se muestra en la siguiente imagen el radio de curvatura con el cálculo de la velocidad lineal constante y angular $0.15/2$ es de 3.0:



Además se observa el movimiento del robot en rviz:



7. Modificamos el fichero control_alumnos para que el robot siga un camino predefinido, se ha modificado las siguientes partes:

Comentamos la última línea del método calcula velocidad lineal de manera que la velocidad sea proporcional a la distancia en vez de constante:

```
#Calculo de la velocidad lineal
def velocidad_lineal_calculada(current_pose, goal_pose):
    #-----
    # Este es el controlador de velocidad lineal
    #-----
    k=0.1; #constante de control proporcional
    error_final_distancia=0.03; # cm expresados en metros
    distancia=euclidean_distance(current_pose, goal_pose);
    velocidad=k*distancia;
    if (velocidad > 0.15):
        velocidad=0.15;
    elif (velocidad<-0.15):
        velocidad=-0.15;
    elif (distancia)<error_final_distancia:
        velocidad=0;
    #velocidad=0.15;

    return velocidad
```

Ajustamos la velocidad angular de forma que sea proporcional al error de la orientación (delta) y la distancia hacia el punto final (L_H):

```
#-----
#Calculo de la velocidad angular
#-----
def velocidad_angular_calculada(current_pose, goal_pose, velocidad_lineal):

    orientation_list = [current_pose.pose.pose.orientation.x, current_pose.pose.pose.orientation.y,
current_pose.pose.pose.orientation.z, current_pose.pose.pose.orientation.w]
    (roll, pitch, yaw) = euler_from_quaternion (orientation_list);

    #-----
    # Este es el controlador de velocidad angular
    #-----
    delta=(current_pose.pose.pose.position.x-goal_pose.pose.position.x)*math.sin(yaw)-
(current_pose.pose.pose.position.y-goal_pose.pose.position.y)*math.cos(yaw);
    LH=euclidean_distance(current_pose, goal_pose);
    rho=2*delta/pow(LH,2);
    velocidad_angular=2*delta/pow(LH,2)
    #velocidad_angular=0.15/3;
    #-----
    return velocidad_angular
```

Además, se ha añadido el método callback la implementación del seguimiento, donde se calcula el índice del punto más cercano del camino, se añade el look_ahead y se comprueba que el índice más el look_ahead no se pasa de la longitud del camino, si se pasa se asigna la última posición del camino.

```

if (len(camino_leido) != 0): #si se ha recibido un camino a seguir

#-----
# Aquí se implementa el seguimiento
#-----

#Calcula el índice mas cercano
indice=get_index_min_distance(camino_leido,data);

#final del camino
final_point= PoseStamped();
final_point.pose.position.x=camino_leido[0][-1]
final_point.pose.position.y=camino_leido[1][-1]
#final_point.pose.position.x=0
#final_point.pose.position.y=6

#punto a seguir
punto_destino= PoseStamped();

# Inicialmente vamos a converger a punto final del camino
#punto_destino=final_point

Look_ahead=3;

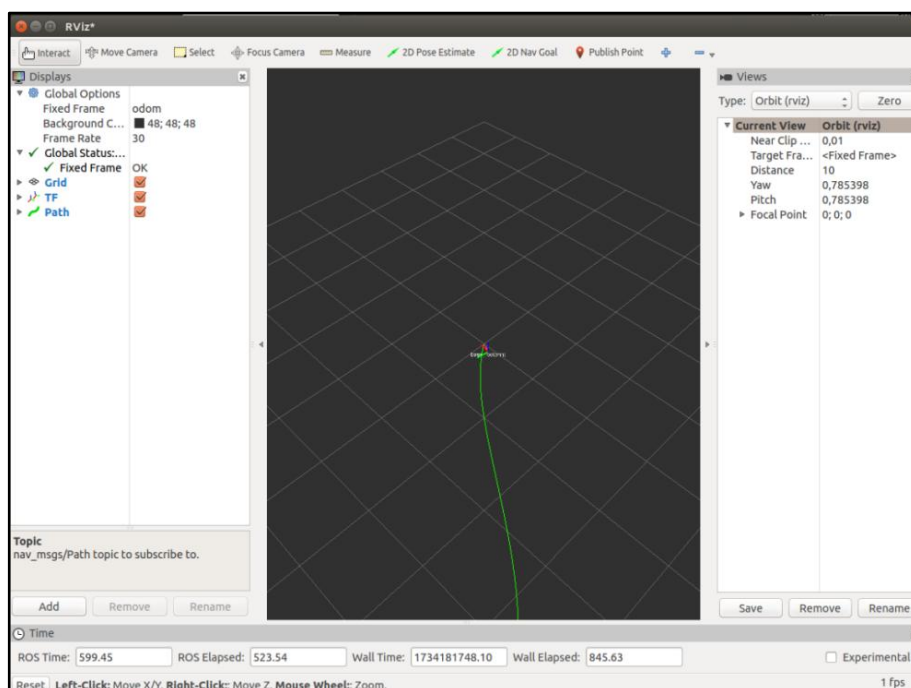
try:
    punto_destino.pose.position.x=camino_leido[0][indice+Look_ahead];
    punto_destino.pose.position.y=camino_leido[1][indice+Look_ahead];
except IndexError:
    punto_destino.pose.position.x=camino_leido[0][-1];
    punto_destino.pose.position.y=camino_leido[1][-1];

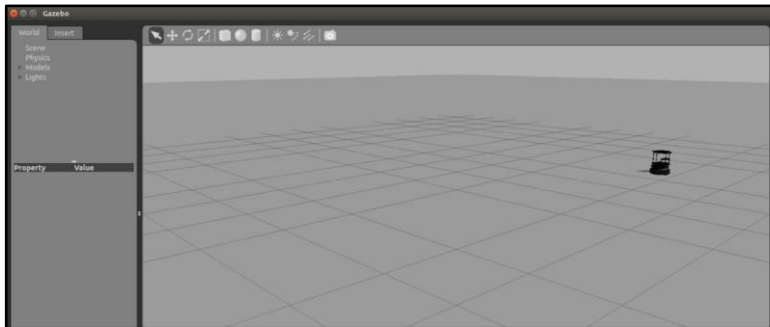
velocidad= Twist();

```

8. Volvemos a ejecutar el fichero control_alumnos:

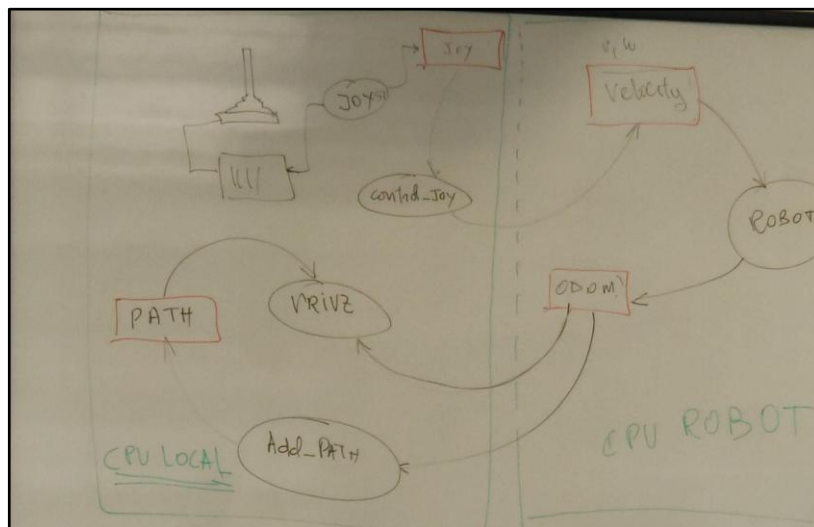
Como observamos en la siguiente imagen se modifica la trayectoria del robot con respecto a la trayectoria anterior:





Práctica 7: ROS: Maestro Remoto y TURTLEBOT Real

En esta práctica se continúa con la configuración y uso de ROS para interactuar con un robot Turtlebot real, distribuyendo la carga de trabajo entre diferentes equipos, donde se detalla cómo configurar un nodo 'roscore' maestro en una máquina remota y cómo acceder a los topics gestionados por este nodo desde otro equipo, para ello nuestra configuración será la que se muestra en el siguiente esquema:



la implementación de la práctica se divide en los siguientes pasos:

- Se ha configurado el archivo /etc/hosts añadiendo las direcciones IP y nombres de las distintas máquinas (para no estar trabajando con las direcciones IPs de cada máquina), con la configuración podemos definir el nodo 'roscore' como maestro:


```

hosts (/etc) - gedit
Archivo Editar Ver Buscar Herramientas Documentos Ayuda
Abrir Guardar Deshacer
hosts x
1 127.0.0.1 localhost
2 127.0.1.1 turtlebot-VirtualBox
3
4 192.168.255.1 ROSMAESTRO
5 192.168.255.2 ROSNOETIC2
6 192.168.255.3 ROSINDIGO
7
8 192.168.2.58 MrBarret
9 192.168.2.59 Ernest
10 192.168.2.60 Castor
11 192.168.2.61 Pollux
12 192.168.2.62 Gertrude
13
14 # The following lines are desirable for IPv6 capable hosts
15 ::1 ip6-localhost ip6-loopback
16 fe00::0 ip6-localnet
17 ff00::0 ip6-mcastprefix
18 ff02::1 ip6-allnodes
19 ff02::2 ip6-allrouters

```

- Después, ejecutamos el comando ping Gertrude para comprobar la conexión:

```

turtlebot@rosindigo:~$ ping Gertrude
PING Gertrude (192.168.2.62): 56(84) bytes of data.
64 bytes from Gertrude (192.168.2.62): icmp_seq=1 ttl=64 time=40.9 ms
64 bytes from Gertrude (192.168.2.62): icmp_seq=2 ttl=64 time=10.1 ms
64 bytes from Gertrude (192.168.2.62): icmp_seq=3 ttl=64 time=4.39 ms
64 bytes from Gertrude (192.168.2.62): icmp_seq=4 ttl=64 time=5.90 ms
64 bytes from Gertrude (192.168.2.62): icmp_seq=5 ttl=64 time=3.92 ms

```

- Configuramos el entorno ejecutando los siguientes comandos en una terminal, que permiten a cualquier nodo ejecutado en dicha terminal pueda tener acceso a los topics del rescote:

turtlebot@rosindigo :~\$ export ROS_MASTER_URI=<http://Gertude:11311>

turtlebot@rosindigo :~\$ export ROS_HOSTNAME=192.168.209.2

- Abrimos otra terminal, y ejecutamos el ssh con Gertrude:

```

turtlebot@rosindigo: ~
turtlebot@rosindigo: ~ 80x24
turtlebot@rosindigo:~$ ssh tb2@ros

```

- Después conectamos el joystick con los comandos que se encuentran en la práctica 6, y trabajaremos con él en la primera ventana:

```

turtlebot@rosindigo:~$ ls /dev/input
by-id event0 event2 event4 event6 js0 js2 mouse0
by-path event1 event3 event5 event7 js1 mice mouse1
turtlebot@rosindigo:~$ ls /dev/input
by-id event0 event2 event4 event6 js1 mouse0
by-path event1 event3 event5 js0 mice mouse1
turtlebot@rosindigo:~$ ls /dev/input
by-id event0 event2 event4 event6 js0 js2 mouse0
by-path event1 event3 event5 event7 js1 mice mouse1
turtlebot@rosindigo:~$ rosparam set joy_node/dev "/dev/input/js2"

```

- Ejecutamos el comando rosparam para poder leer el topic del puerto de entrada:

- Abrimos otra terminal, configuramos el nodo maestro y ejecutamos el siguiente comando:

```
turtlebot@rosindigo :~$roslaunch control_turtlebot control_joy.py
```

- Abrimos otra ventana del robot ssh y ejecutamos:

```
turtlebot@rosindigo :~$ roslaunch turtlebot_bringup minimal.launch
```