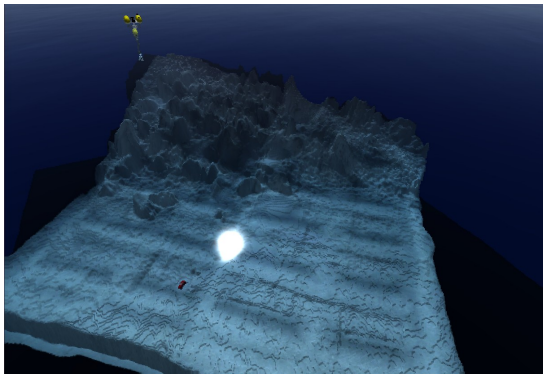
Autores: Javier Pérez Soler y Raúl Marín Prades

Introducción.



En este tutorial aprenderemos a usar UWSim un simulador open source de robots submarinos y un plugin de benchmarking que nos permitirá evaluar los algoritmos que utilicemos. Para ello necesitamos conocer el funcionamiento de ROS (Robot Operating System), y las interfaces que utiliza: topics y servicios. Finalmente se proponen una serie de ejercicios para conocer qué tipo de cosas podemos hacer con el simulador y cómo resolver problemas sencillos de robótica.

Instalación de la plataforma.

Podemos instalar el software en nuestra máquina o utilizar la máquina virtual de aquí. La máquina virtual requiere de un ordenador más potente para funcionar igual que una instalación local. La maquina virtual ya tiene los primeros pasos de instalación, ejecución e instalación de escenas.

- PASO 1: Instalar ROS versión indigo siguiendo las instrucciones de la página web: http://wiki.ros.org/indigo
- PASO 2: Debemos crear un workspace de ROS. El workspace de ROS es una carpeta donde se encuentran todos los fuentes de nuestro proyecto en una estructura lógica. Para crearlo hacemos:
  - mkdir -p ~/uwsim_ws/src
  - cd ~/uwsim_ws/src
  - catkin_init_workspace
  - cd ~/uwsim_ws
  - catkin_make install
- PASO 3: Añadir el workspace a nuestro bashrc. Si añadimos nuestro workspace al bashrc cada vez que arranquemos una consola estará disponible en las variables de entorno y podremos ejecutar sin problemas los programas. Para ello editamos el fichero "~/.bashrc" y añadimos al final:
  - source ~/uwsim_ws/install/setup.bash

- PASO 4: Instalar librerías para uwsim:
  - sudo apt-get install ros-indigo-uwsim-bullet ros-indigo-uwsim-osgbullet ros-indigo-uwsim-osgocean ros-indigo-uwsim-osgworks ros-indigo-visualization-osg
- PASO 5: Definir las fuentes desde donde se instalará uwsim.
  - cd ~/uwsim_ws/
  - gedit .rosinstall

    Rellenamos el fichero con lo siguiente:

```
- other: {local-name: /opt/ros/indigo/share/ros}
- other: {local-name: /opt/ros/indigo/share}
- other: {local-name: /opt/ros/indigo/stacks}
- setup-file: {local-name: /opt/ros/indigo/setup.sh}
- git: {local-name: src/underwater_simulation,
      uri: 'https://github.com/perezsolerj/underwater_simulation.git', version: indigo-devel}
- git: {local-name: src/benchmarks,
      uri: 'https://github.com/perezsolerj/uwsimbenchmarks.git', version: indigo-devel}
- git: {local-name: src/pipefollowing,
      uri: 'https://github.com/perezsolerj/pipefollowing.git', version: master}
```

- PASO 6: Descargar ficheros:
  - rosws update
- PASO 7: Instalar dependencias y compilar:
  - rosdep install --from-paths src --ignore-src --rosdistro indigo -y
  - catkin_make install

Primera ejecución.

Ahora ya tenemos todo el sistema instalado y compilado. Ahora vamos a ejecutar una escena predefinida de UWSim y comprobar que todo funciona como es debido.

- PASO 1: Asegurarse de que roscore está en marcha. Este es un servidor que se encarga de la comunicación entre procesos de ROS, y por tanto es necesario que esté funcionando en todo

momento. Abre una terminal nuevo y ejecuta:

- roscore

- PASO 2: Ejecuta UWSim.
  - rosrun uwsim uwsim

    NOTA: La primera vez que ejecutamos UWSim nos pide descargar un fichero de datos. Debemos decirle que si para que descargue los modelos básicos de terrenos, vehículos...etc que se instalarán en ~/.uwsim

    NOTA2: Si ves efectos de visualización raros o un segmentation fault al inicio puede ser debido a que los drivers de la gráfica no están correctamente instalados o tu gráfica no está soportada (cualquier gráfica Nvidia o ATI modernas deberían ser capaces de funcionar). Aún puedes ejecutar el simulador con la opción "- - disableShaders"

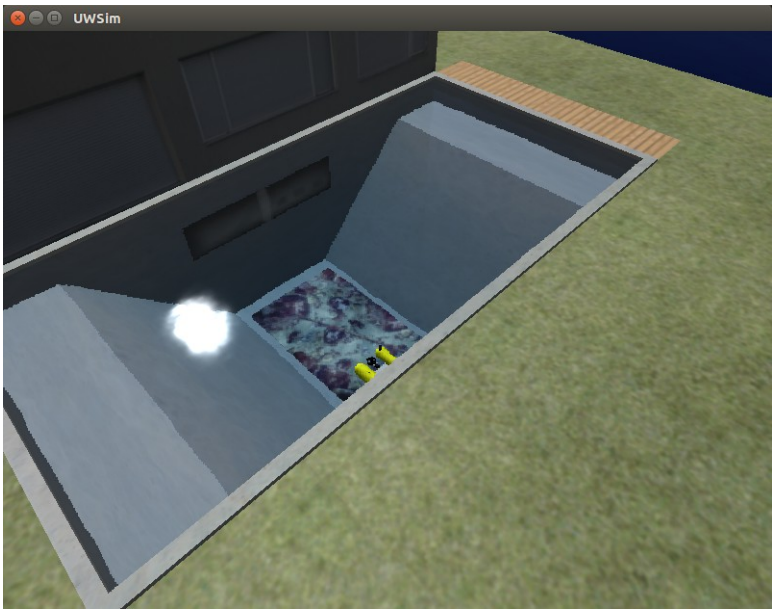


Imagen de ejecución correcta de UWSim.

- PASO 3: Comprueba que UWSim está publicando información. UWSim utiliza las interfaces de ROS para simular sensores como cámaras, sonares, dvl... podemos ver todo lo que se está publicando en una nueva terminal mientras UWSim se ejecuta con:
  - rostopic list

  NOTA: puedes ver lo que lleva cada mensaje con los siguientes comandos de ROS:
  - rostopic echo NOMBREDEINTERFAZ

- PASO 4: Ver las cámaras virtuales. Uno de los sensores más importantes son las cámaras. Para ver lo que están viendo puedes pulsar "c" en la ventana principal y ver un widget de las cámaras simuladas. Además puedes acceder a su publicación por ROS mediante:

- rosrun image_view image_view image:=/uwsim/camera1

Ejemplos básicos de uso comandos ROS

En este apartado vamos a realizar algunos ejemplos básicos de uso de los comandos ROS:

1. Visualizar la lista de nodos ROS activos:

$rosnode list

2. Comprueba la comunicación con el nodo especificado:

$rosnode ping /UWSim

3. Visualizar lista de topics, imprimir valores, y modificarlos:

$ rostopic -h

rostopic bw display bandwidth used by topic

rostopic echo print messages to screen

rostopic hz display publishing rate of topic

rostopic list print information about active topics

rostopic pub publish data to topic

rostopic type print topic type

4. Visualizar el estado de las articulaciones del brazo manipulador:

$rostopic echo /uwsim/joint_state

5. Visualizar el estado de publicaciones de topics, servicios y tipos de mensajes de un nodo:

$rosnode info /UWSim

6. Visualizar el estado del sensor de presión:

$rostopic echo /g500/pressure

7. Visualizar el estado del sensor de fuerza:

$rostopic echo /g500/ForceSensor

8. Visualizar el estado del sensor DVL:

$rostopic echo /g500/dvl

9. Visualizar el tipo de datos asociado al topic /dataNavigator:

$rostopic type /dataNavigator

10. Mover el robot en velocidad usando el comando setVehicleVelocity:

$rosrun UWSim setVehicleVelocity /dataNavigator 0.1 0 0 0 0 0

$rosrun UWSim setVehicleVelocity /dataNavigator 0 0.1 0 0 0 0

$rosrun UWSim setVehicleVelocity /dataNavigator 0 0 0.1 0 0 0

$rosrun UWSim setVehicleVelocity /dataNavigator 0.05 0.05 0.0 0 0 0.05

11. Mover el robot en posición usando el comando setVehiclePosition:

$rosrun UWSim setVehiclePosition /dataNavigator 0.1 0 0 0 0 0

12. Conocer la estructura de un mensaje ROS:

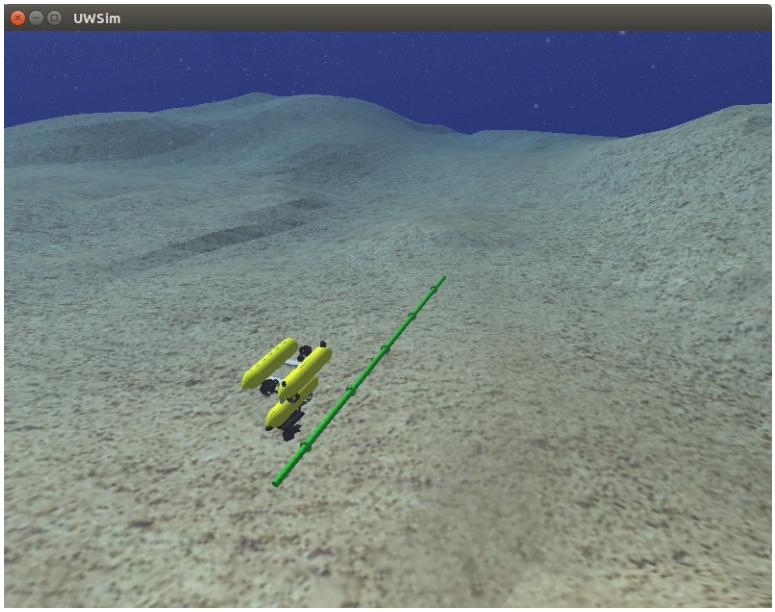
rosmsg show nav_msgs/Odometry

Escenas de seguimiento de tuberías.

En estas prácticas vamos a desarrollar un programa que nos permita seguir tuberías submarinas en UWSim. Estas escenas no se descargan por defecto con UWSim por tanto deberemos descargarlas y añadirlas al repertorio de escenas. Para ello debemos:

- PASO 1: Situarnos en la carpeta src de UWSim:
  - cd ~/uwsim_ws/src/underwater_simulation/uwsim
- PASO 2: Ejecutar el script de instalación para cada una de las escenas:
  - ./data/scenes/installScene -s pipeFollowing_basic.uws
  - ./data/scenes/installScene -s pipeFollowing_turns.uws
  - ./data/scenes/installScene -s pipeFollowing_heights.uws
- PASO 3: Compilar el workspace para que las nuevas escenas estén disponibles al ejecutar UWSim:
  - cd ~/uwsim_ws/
  - catkin_make install
- PASO 4: Comprobar que las nuevas escenas se ejecutan sin problemas:
  - rosrun uwsim uwsim - -configfile pipeFollowing_basic.xml
  - rosrun uwsim uwsim - -configfile pipeFollowing_turns.xml

- rosrun uwsim uwsim - -configfile pipeFollowing_heights.xml



Teleoperación de robots.

Ahora ya tenemos el setup básico instalado y funcionando para las prácticas. El objetivo final es conseguir que nuestro vehículo sea capaz de moverse siguiendo las tuberías verdes de las escenas. Como se ve las escenas tienen diferentes grados de dificultad, desde basic que es una linea recta, heights que contiene cambios de alturas y turns que las tuberias hacen diversas formas esquivando los cambios de alturas.

El simulador nos permite ir desarrollando nuestro código de manera escalonada para que la curva de aprendizaje sea más sencilla. Por tanto iremos afrontando retos cada vez más realistas hasta acabar con un código que podría ser puesto en marcha en un robot real.

El primer paso es por tanto ser capaces de enviar órdenes al robot para que se mueva por la escena y poder teleoperarlo por teclado siguiendo la tubería.

Lo primero es familiarizarse con el entorno de trabajo, ya conocemos como compilar el workspace con “catkin_make install” y como ejecutar el simulador con “rosrun uwsim uwsim - - configfile ESCENA”. Ahora abrimos un nuevo terminal y nos situamos en “**~/uwsim_ws/src/pipeFollowing**”.

Este es el paquete, el código en ROS se divide en paquetes, de seguimiento de tuberías. Esta carpeta ya está con el formato de paquete ROS, lo que significa que cuando ejecutemos **“catkin_make install**” se compilará junto con el resto. El código de nuestros programas se encuentra en “/src”.

Alli se encuentran los “esqueletos” de los programas que desarrollaremos en estas prácticas. El primero de ellos es “PF_teleop.py”. Este programa lee la entrada de teclado y envía comandos al vehículo. Para ejecutarlo seguimos los siguientes pasos:

- PASO 1: Abrir el simulador con la escena que queramos:

    - rosrun uwsim uwsim - -configfile pipeFollowing_basic.xml

- PASO 2: En un nuevo terminal ejecutamos el programa de teleoperación:
    - rosrun pipefollowing PF_teleop.py
- PASO 3: Controlar el vehículo. Pulsando las teclas de cursores (vertical) veremos como el vehículo se mueve arriba y abajo.

Desafortunadamente los movimientos que hay programados no son muy útiles para seguir tuberías. Edita por tanto el fichero PF_teleop.py y echemos un vistazo a lo más importante:

```
twist_topic="/g500/velocityCommand"

...

pub = rospy.Publisher(twist_topic, TwistStamped,queue_size=1)
```

Esta línea decide el topic al que enviamos, en este caso g500/velocityCommand.

Ya dentro del bucle, que se ejecutará mientras ROS esté activo:

```
msg = TwistStamped()
```

Crea un mensaje del tipo TwistStamped, este es el tipo de mensaje que nos permite mover el vehículo.

```
if c=='\n':
        start()
        print "Benchmarking Started!"
 elif c=='\x1b':  ##This means we are pressing an arrow!
        c2= sys.stdin.read(1)
        c2= sys.stdin.read(1)
       if c2=='A':
           msg.twist.linear.z=-baseVelocity
       elif c2=='B':
           msg.twist.linear.z=baseVelocity
```

Estas líneas son las que recogen la entrada de teclada en la variable “c” de tipo char. Vemos como de momento pulsar “intro” llama a start() (veremos más adelante que es) y las flechas arriba y abajo rellenan los campos de msg con velocidades lineales en z.

```
pub.publish(msg)
```

rospy.sleep(0.1)

Finalmente en estas lineas se publica el mensaje que acabamos de rellenar y que finalmente llegará al simulador para mover el vehículo.

EJERCICIO: Ahora el objetivo es completar este programa de forma que nos permita mover el vehículo para poder seguir la tubería en cualquier escenario. Para ello deberemos añadir mas condiciones al if para comprobar otras teclas pulsadas y rellenar los campos de msg de manera acorde a ello. Los valores que contiene msg son:

Velocidades lineales: msg.twist.linear.x, msg.twist.linear.y, msg.twist.linear.z

Velocidades angulares(giro): msg.twist.angular.x, msg.twist.angular.y, msg.twist.angular.z

Autoevaluacion.

Una vez tenemos nuestro programa de teleoperación listo hay que comprobar que funciona adecuadamente. Para ello tenemos el plugin de benchmarks. Este plugin se encarga de medir distintas cosas en los programas que se ejecutan en UWSim. En este caso medirá como de bueno es nuestro seguimiento a la tubería.

Para ejecutarlo tenemos unos ficheros launch ya creados en el paquete pipefollowing. Estos ficheros launch engloban diferentes comandos ROS y permiten ejecutar más de una orden a la vez. Sin embargo tendremos que avisar al módulo de benchmarks cuando empezamos nuestra ejecución y cuando la terminamos.

Para hacer esto utilizaremos servicios, a diferencia de los topics estas interfaces de ROS permiten simular una llamada a función entre procesos con unas entradas y salidas. En nuestro caso no hacen falta ni entradas ni salidas, con llamarlos es suficiente. Esto ya está incluido en el fichero de teleop original. Si le echamos un vistazo:

##wait for benchmark init service

#rospy.wait_for_service('/startBench')

#start=rospy.ServiceProxy('/startBench', Empty)

##wait for benchmark stop service

#rospy.wait_for_service('/stopBench')

#stop=rospy.ServiceProxy('/stopBench', Empty)

Es necesario descomentar estas lineas que esperan hasta ver los servicios start y stop activos. Además crean dos variables start y stop que al ser llamadas más tarde (“start()” o “stop()”) marcarán el inicio y final del seguimiento.

Una vez descomentadas podremos iniciar el benchmark y pararlo con “intro” y “space”

respectivamente. Por tanto para medir nuestro desempeño seguimos los siguientes pasos:

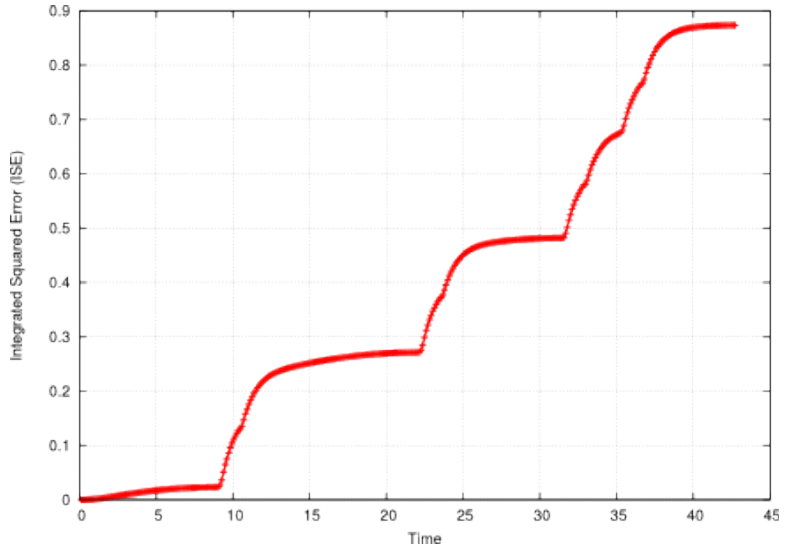
- PASO 1: Ejecutar el launch correspondiente:
  - roslaunch pipefollowing basic.launch
  - roslaunch pipefollowing turns.launch
  - roslaunch pipefollowing heights.launch
- PASO 2: Ejecuta el teleop:
  - rosrun pipefollowing PF_teleop.py
- PASO 3: Sigue la tubería:
  - Aprieta “intro” para empezar a medir.
  - Sigue la tubería, ten en cuenta que la linea verde es la trayectoria perfecta y la roja la seguida por el robot.
  - Aprieta “space” al llegar al final.

Una vez el benchmark termine se podrán consultar los resultados con el script **genResults.sh** dentro de la carpeta src del paquete pipefollowing. Este script genera graficas de la evolución del error a lo largo del tiempo del experimento. Este script utiliza gnuplot y por tanto es necesario tenerlo instalado para que funcione.

$sudo apt-get install gnuplot

$sudo apt-get install texlive-font-utils

El resultado que obtendremos serán dos gráficas. ResultsError.pdf es una gráfica que muestra el error en cada momento a lo largo del tiempo, mientras que resultsISE.pdf es la acumulación de este error. El ISE (Integrated Squared Error) es una medida de calidad del seguimiento de la trayectoria y por tanto nuestro objetivo a minimizar. Para ello es conveniente hacer el seguimiento cuanto más ajustado a la trayectoria ideal y rápido posible.



Salida del Integrated squared Error.

NOTA: Cada vez que ejecutamos un nuevo benchmark sobreescribiremos los datos recogidos por el anterior, por ello es conveniente guardar las gráficas de resultados de cada experimento. Si queremos los datos en crudo, se encuentran en "~/.uwsim" en dos ficheros ".dat" y ".data".

Dinámica.

Una vez hemos resuelto el problema cinematicamente, asignando velocidades al vehículo, es conveniente plantearse el problema dinámico, mediante fuerzas. Hasta el momento cuando mandamos un comando de velocidad al vehículo este se mueve a esa velocidad, sin importar la velocidad anterior. Esto no es realista ya que el vehículo tiene una inercia importante bajo el agua.

EJERCICIO: Por tanto a continuación se plantea resolver el mismo problema anterior pero con el modelo dinámico simulado del vehículo. Para ello tan solo es necesario cambiar el tipo de mensaje al que enviamos, puesto que ahora en lugar de enviar mensajes de velocidad mandaremos fuerzas a distintos motores del vehículo.

El vehículo que utilizamos en el ejemplo, girona500, tiene 5 motores los dos primeros están enfocados hacia adelante y proporcionan movimiento hacia adelante y giro (mandando fuerzas de diferente signo a cada uno). Los dos siguientes son verticales y consiguen movimiento en Z, y pueden proporcionar giro pitch aunque no es necesario para estos ejercicios. Finalmente el último motor es para movimientos laterales.

Para cambiar el tipo de mensaje que se envía será necesario modificar las líneas:

from geometry_msgs.msg import TwistStamped

Esta línea importa el tipo de mensaje que ahora será:

from std_msgs.msg import Float64MultiArray

El mensaje que crearemos ahora estará acorde a esto por tanto:

msg = Float64MultiArray()

De este mensaje unicamente tenemos que rellenar el campo data, el cual es un vector con la información de los motores descrita anteriormente. La fuerza de los motores va de -1 a 1. Para asignarlo podemos crear en una lista de python la fuerza de cada motor y asignarla después a esta variable.

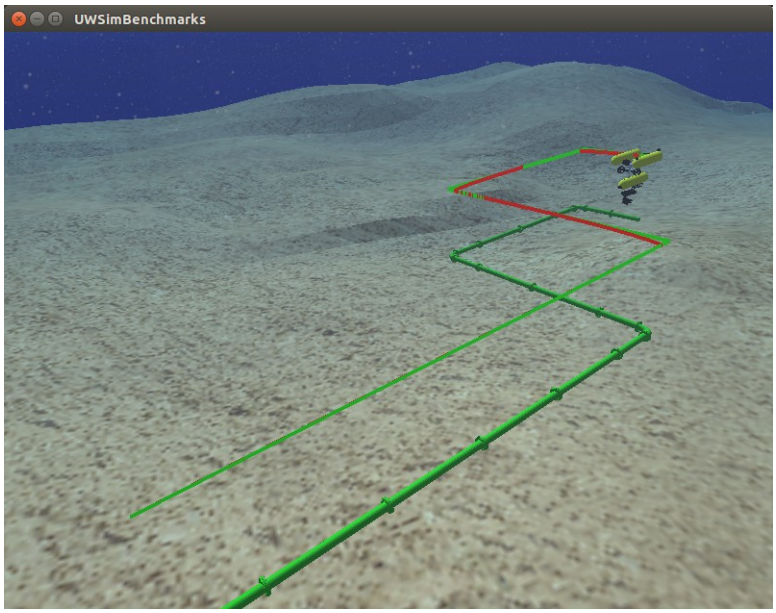
Para probar esta parte necesitamos lanzar un launch distinto al del caso cinemático ya que lanza la parte dinámica:

- roslaunch pipefollowing dyn_basic.launch
- roslaunch pipefollowing dyn_turns.launch
- roslaunch pipefollowing dyn_heights.launch

Navegación autónoma.

EJERCICIO: Ahora que denominamos los movimientos del vehículo, el objetivo es conseguir que navegue de manera autónoma de un punto a otro. De momento estos puntos seran parte de la entrada al problema, no será necesario calcularlos.

El “esqueleto” de este problema está en PF_waypoints.py. Un primer vistazo del problema vemos que estamos resolviendo la versión cinemática del problema, ya nos ocuparemos de la dinámica más tarde.



Ejecución de navegación autónoma en el escenario turns.

Existen múltiples formas de resolverlo, la aquí propuesta consiste en lo siguiente:

Mientras no estemos en el último punto:

1. Leer posición actual del vehículo.
2. Calcular distancia al destino respecto al vehículo.
3. Comprobar si hemos alcanzado el punto destino.
    1. Si hemos alcanzado el destino: saltar al siguiente punto.
    2. Si no: decidir la velocidad del vehículo.

Para leer la posición del vehículo utilizaremos TF. TF es una librería que nos sirve calcular posiciones de objetos respecto a otros. El simulador está continuamente publicando posiciones de vehículos respecto al mundo.

Para ello deberemos incluir la librería TF.

import tf

Crear un “transform listener”.

listener = tf.TransformListener()

Pedir al “listener” la transformación que queremos, en este caso del mundo “/world” al vehículo “/girona500”

```
try:
  (trans,rot) = listener.lookupTransform('/world', '/girona500', rospy.Time(0))
```

except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):

continue

Como vemos, hemos obtenido un transformacion “trans” y una rotación “rot”. Esto es importante porque los puntos que debemos alcanzar los tenemos respecto al mundo. Necesitamos por tanto calcular la diferencia de ambos puntos respecto al vehículo, de esta forma sabremos hacia donde debemos mover el vehículo para alcanzar el punto.

Para ello se utilizan las matrices homogeneas que nos permiten transformar facilmente la referencia de los puntos. Por tanto lo primero que debemos hacer es representar la translación y rotación como una matriz homogenea utilizando la libraría TF. Usaremos las funciones:

wRv=tf.transformations.quaternion_matrix(rot)

wTv=tf.transformations.translation_matrix(trans)

De esta forma obtenemos la rotacion (R) del vehículo (v) respecto al mundo(w) → wRv. Y la translacion (T) del vehículo (v) respecto al mundo (w) → wTv. Combinando estas dos con un “dot product” podemos obtener la matriz (M) de transformación del vehículo (v) respecto al mundo (w) ->wMv.

wMv=np.dot(wTv,wRv)

Ahora necesitamos trambién la matriz de transformacion del punto (p) respecto al mundo → wMp. Hay que tener en cuenta que el punto solo tiene translación ya que no nos importa el giro del vehículo respecto al punto. Esto implica que no necesitaremos “dot product”.

El resultado que queremos es la matriz de transformación del punto respecto al vehículo → vMp. Por tanto la operación necesaria es:

vMp = vMw . Wmp = inverse(wMv) . Wmp

Para hacer la inversa podemos utilizar:

tf.transformations.inverse_matrix(MATRIX)

Finalmente, unicamente necesitamos la translación de ahí, la cual podemos obtener con:
vTp=tf.transformations.translation_from_matrix(vMp)

Ahora ya tenemos la translación desde el vehículo al punto con la rotación del vehículo. Debería ser fácil calcular la distancia al punto para decidir si hemos llegado o no.

Por último queda lo mas importante, decidir la velocidad del vehículo. Aquí lo más común es aplicar una velocidad en función de la distancia al objetivo (con un máximo para no alcanzar velocidades demasiado altas).

Autoevaluación.

No nos olvidemos que una vez esté listo todo debemos autoevaluarlo con el módulo de benchmarking para comprobar el resultado del álgoritmo creado.

Dinámica.

Nuevamente, una vez tenemos resuelto el problema cinemático es conveniente plantearnos el problema dinámico. De forma análoga al apartado anterior se puede cambiar el tipo de mensaje que enviamos para enviar fuerzas en lugar de velocidades. Además deberemos lanzar la versión dinámica de la escena para que se active la simulación dinámica.

Navegación por visión.

EJERCICIO: Una vez hemos conseguido seguir de manera autónoma las tuberías el objetivo es seguirlas utilizando una cámara simulada. El objetivo es desde la posición inicial llegar al final de la tubería utilizando únicamente la cámara como entrada.

El “esqueleto” de este problema esta en PF_vision.py. En este caso también partirmos del caso cinemático. Aquí el abanico de posibilidades para resolver el ejercicio se abre mucho pero el método de resolución que se propone es el siguiente:

1. Obtener la imagen del sensor
2. Binarizar la imagen
3. Encontrar lineas rectas en la imagen
4. Decidir la velocidad del vehículo dependiendo de la posición e inclinación de las rectas.

Lo primero que vemos en el “esqueleto” del problema es que tenemos una clase llamada imageGrabber. Esta clase simplemente coge la imagen del simulador y la muestra en una ventana. El próposito es separar la percepción (lectura de sensores) de la acción. De forma que llamaremos a métodos de la clase como getSize() para obtener la información de la última imagen que se ha recibido para tomar decisiones.

Lo primero que debemos hacer es binarizar la imagen para separar tubería del resto. Esto lo podemos hacer en el mismo image_callback donde se recibe la imagen, de esta forma obtendremos todo lo que necesitamos de la imagen allí. Para ello usaremos la función de opencv:

cv2.inRange(image, lower_green, upper_green)

la cual recibe una imagen y dos arrays de numpy (“np.array([0,0,0])”) con los valores de inicio y fin para binarizar la imagen. Es decir, todos los valores entre uno y otro array quedarán en un valor y el resto en otro. El resultado es otra imagen que podemos visualizar con imshow.

Es recomendable que veamos que la imagen se binariza correctamente a lo largo de toda la tubería,

hay zonas con sombras que cambian los colores de la tubería. Podemos usar el programa de navegación autónoma anterior para mover el vehículo a lo largo de la tuberia.

Si no podemos separar la imagen correctamente en espacio RGB, quizás sea conveniente pasar a espacio HSV previamente con:

```
hsv = cv2.cvtColor(cv_image, cv2.COLOR_BGR2HSV)
```

Ahora hay que pasar un detector de bordes a la imagen binarizada, si la binarización es correcta y solo vemos la tubería deberíamos obtener unicamente dos lineas. La función que podemos utilizar para esto la de Canny:

```
edges = cv2.Canny(bin_image,50,150,apertureSize = 3)
```

Una vez tenemos los bordes de la imagen pasamos a un detector de bordes, en este caso hemos escogido Hough:

```
lines = cv2.HoughLinesP(edges,1,np.pi/180,50,60,20)
```

Puede ser conveniente ajustar los dos últimos parámetros “threshold” y “minLineLength” dependiendo de la binarización unos valores serán mejores que otros.

Para comprobar como lo estamos haciendo es conveniente dibujar estas lineas sobre la imagen original y mostrarla usando lo siguiente:

```
if lines!=None:
    for x1,y1,x2,y2 in lines[0]:
     cv2.line(cv_image,(x1,y1),(x2,y2),(0,255,0),2)
cv2.imshow("Image window", cv_image)
cv2.waitKey(3)
```

Se recomienda invertir un poco de tiempo comprobando que encuentra suficientes lineas a lo largo de la tubería, ya que el movimiento va a depender de la correcta detección de las mismas. Una vez tenemos hecho esto hemos acabado con la percepcion guardamos las lineas resultantes y creamos una función para consultarlas cuando sea necesario.

A continuación en el bucle principal, fuera de la clase, consultaremos esta función en cada iteración para decidir hacia donde mover el vehiculo. Una vez tenemos las lineas detectados existen una enorme cantidad de posibilidades, aquí se explica una dado su simplicidad aunque probablemente no sea la óptima.

La idea es mientras tengamos lineas verticales seguir hacia adelante. Una vez no detectemos lineas verticales o estén muy abajo giraremos para convertir las demas lineas en verticales. Para ello lo primero que debemos hacer es decidir si una linea es vertical. Esto es sencillo dado que tenemos las lineas definidas como (x1,y1)(x2,y2). La arcotangente de dx/dy nos dá el angulo de la linea. Cuanto

mas cercano a 0 mas vertical es.

Una vez las tenemos divididas en lineas verticales y “no verticales” vamos a calcular el centroide de estas lineas de manera separada. De esta forma el centroide de lineas verticales nos servirá para dos propósitos:

- Mover lateralmente el vehículo para que la tuberia se mantenga en el centro.
- Comprobar si se ha acabado la tubería, cuando el centroide está “muy abajo” de la imagen.

El centroide de las lineas “no verticales” nos servirá cuando las lineas verticales no existan saber hacia donde debemos girar. Si el centroide se encuentra mas a la izquierda giraremos hacia allí y al revés.

Además de los centroides es conveniente ver el grosor de la tubería en píxeles, especialmente para el escenario de cambio de alturas.

Por lo tanto para decidir como mover el vehículo tendremos en cuenta:

- X: diferencia del centroide de lineas verticales respecto al centro. Si el centroide está en el centro exacto debemos avanzar hacia adelante, si está próximo al limite inferior estamos llegando al final o a un giro.
- Y: diferencia del centroide de lineas verticales respecto al centro. Debemos mantener el centroide de las lineas en el centro de la imagen con movimientos laterales.
- Z: diferencia del grosor en pixeles respecto al valor “ideal”.
- Angular Z: Si el centroide de las lineas verticales está proximo al limite inferior giraremos hacia el centroide de las lineas “no verticales”.

Autoevaluación.

No nos olvidemos que una vez esté listo todo debemos autoevaluarlo con el módulo de benchmarking para comprobar el resultado del álgoritmo creado.

Dinámica.

Nuevamente, una vez tenemos resuelto el problema cinemático es conveniente plantearnos el problema dinámico. De forma análoga al apartado anterior se puede cambiar el tipo de mensaje que enviamos para enviar fuerzas en lugar de velocidades. Además deberemos lanzar la versión dinámica de la escena para que se active la simulación dinámica.