POLITECNICO DI BARI

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE (DEI)

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA DELL'AUTOMAZIONE - ROBOTICS

Progetto in

Robotics – Mobile and field Robotics

AUTONOMOUS TRACTOR

Prof.:

Luca DE CICCO

Tutor:

Carlo CROCE

Studente:

Arianna RANA 577380

Anno Accademico 2019-2020

SOMMARIO

1. INTRODUZIONE

In un mondo in continua evoluzione tecnologica, anche l'agricoltura procede in questa direzione.
Se fino a pochi anni fa era un lavoro svolto esclusivamente da persone, oggigiorno l'agricoltura tradizionale viene affiancata da dispositivi intelligenti capaci di semplificare il lavoro e migliorare e aumentare la produzione.
Droni e trattori a guida autonoma sono tra i principali dispositivi usati per la modernizzazione dell'agricoltura. Dotati di "intelligenza", riescono a muoversi agilmente e in collaborazione con altri robot all'interno dei campi coltivati, grazie all'ausilio di sensori che permettono la conoscenza dell'ambiente circostante.

Lo scopo di questo progetto è quello di far eseguire un percorso prestabilito al robot in maniera autonoma, senza l'intervento umano e senza la conoscenza della mappa dell'ambiente in cui andrà ad operare.
Il progetto è stato sviluppato attraverso l'uso di ROS e con l'ausilio di pacchetti preesistenti che hanno permesso di implementare la navigazione autonoma. A tal proposito, è stato necessario equipaggiare il robot di sensori, come il LiDAR e il GPS, che permettessero allo stesso di avere la percezione dell'ambiente circostante.

2. MODELLO DEL ROBOT

Il modello usato per il progetto è INNOK HEROS, un robot all-terrain modulare disponibile in configurazioni a 2 o 4 ruote motrici (in figura si può osservare quello con 4 ruote motrici).

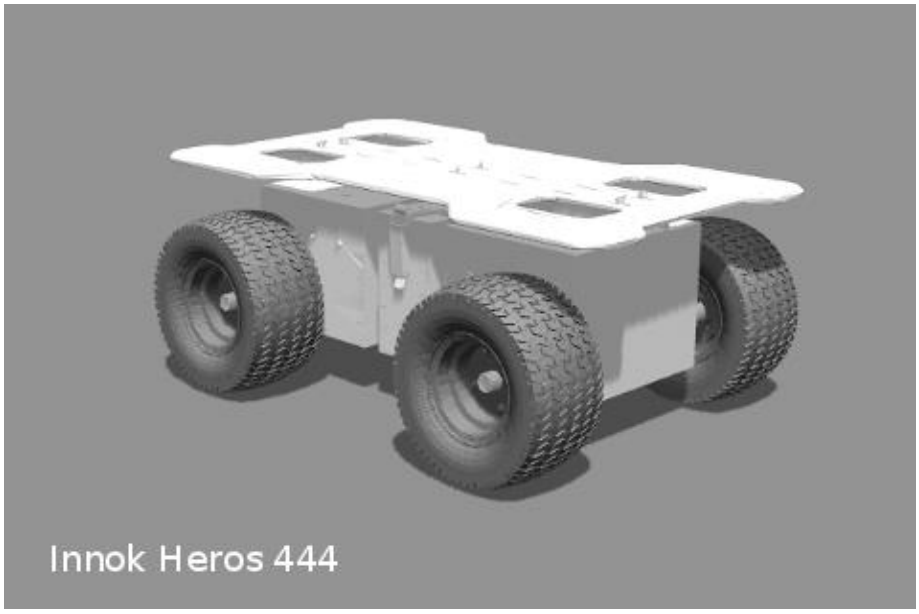


Figura 1: modello del robot

Facilmente modificabile grazie alla sua struttura modulare, si può adattare a diverse applicazioni, quali agricole e industriali.

È stato scelto questo robot per le sue caratteristiche strutturali che lo rendono molto robusto, capace di trascinare un autoveicolo e di trasportare sulla piattaforma carichi fino a 200 kg. Inoltre, è equipaggiabile di innumerevoli sensori e accessori in base al lavoro che deve svolgere.

2.1 URDF

L'URDF (Unified Robot Descripion Format) è un modo per descrivere la geometria del robot con le parti che lo compongono tramite il linguaggio XML. Con questo formato è possibile costruire il modello visuale del robot e definirne i giunti fissi e/o mobili. Per ridurre la lunghezza del codice e renderlo riutilizzabile, viene usato il linguaggio XACRO.

Nel caso specifico, essendo un URDF creato da terze parti, si è optato per utilizzare il modello di Innok come fosse una dipendenza, creando un file URDF specifico che importa il modello di Innok e lo arricchisce adattandolo alle nostre esigenze. Questo è stato possibile proprio grazie all'uso delle xacro

Per poter simulare in un ambiente che riproduce l'ambiente reale, è necessario aggiungere alla descrizione le proprietà di <collision> e <inertial>:

```
<!-- Base Link -->
  <link name="link1">
    <collision>
      <origin xyz="0 0 ${height1/2}" rpy="0 0 0"/>
      <geometry>
        <box size="${width} ${width} ${height1}"/>
      </geometry>
    </collision>

    <visual>
      <origin xyz="0 0 ${height1/2}" rpy="0 0 0"/>
      <geometry>
        <box size="${width} ${width} ${height1}"/>
      </geometry>
      <material name="orange"/>
    </visual>

    <inertial>
      <origin xyz="0 0 1" rpy="0 0 0"/>
      <mass value="1"/>
      <inertia
        ixx="1.0" ixy="0.0" ixz="0.0"
        iyy="1.0" iyz="0.0"
        izz="1.0"/>
    </inertial>
  </link>
```

Figura 2: esempio tag <collision> e <inertial>

L'URDF si basa sulla creazione di un albero con un'unica radice e vari figli. Generalmente, il <base_link> è la radice a cui vengono collegate tutte le altre parti del modello come figlie, tramite i <joint>:

```
<joint name="joint2" type="continuous">
    <parent link="link2"/>
    <child link="link3"/>
    <origin xyz="0 ${width} ${height2 - axel_offset*2}" rpy="0 0 0"/>
    <axis xyz="0 1 0"/>
  </joint>
```

Figura 3: esempio tag <joint>

Si crea così una dipendenza tra le componenti come osservabile nel grafico:

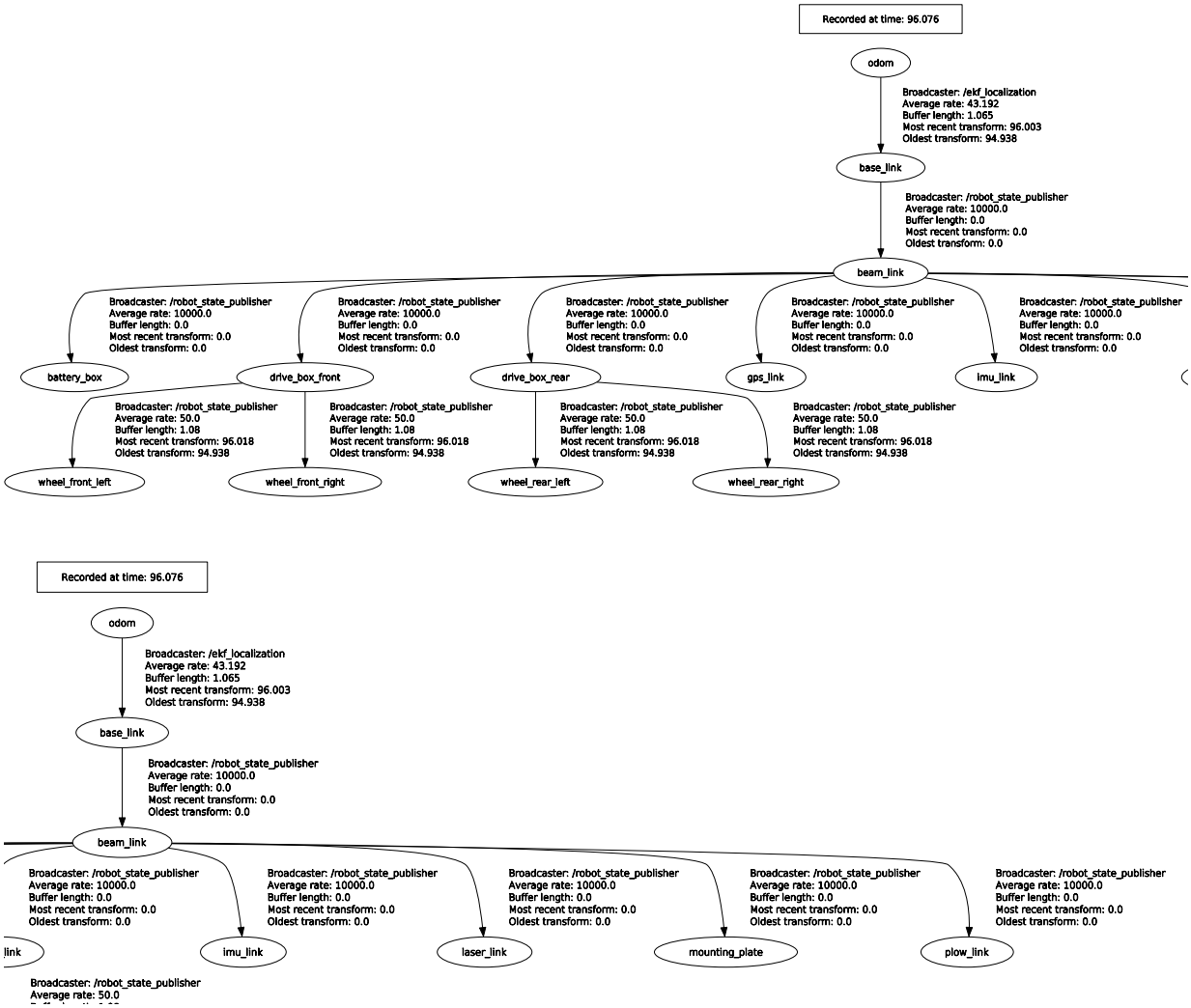


Figura 4: tf_tree

Come si evince dal grafico, oltre alle componenti base del robot, sono stati aggiunti due link che rappresentano il laser e il gps, che serviranno per la pianificazione della traiettoria, e il link “plow” che rappresenta l’aratro. Le caratteristiche dei due sensori verranno discusse ampliamente nel capitolo 4: SENSORI.

Per completare il modello, è necessario inserire la proprietà di <transmission> che permette di descrivere la relazione tra un attuatore e un giunto non fisso:

```
<transmission name="tran1">
    <type>transmission_interface/SimpleTransmission</type>
    <joint name="joint1">
      <hardwareInterface>EffortJointInterface</hardwareInterface>
    </joint>
    <actuator name="motor1">
      <hardwareInterface>EffortJointInterface</hardwareInterface>
      <mechanicalReduction>1</mechanicalReduction>
    </actuator>
  </transmission>
```

Figura 5: esempio tag <transmission>

Avendo scelto come configurazione quella con quattro ruote motrici, avrà bisogno di un tag <transmission> per ogni ruota.

3. AMBIENTE DI SIMULAZIONE

Per il progetto sono stati usati RVIZ E GAZEBO

Il primo è uno strumento che permette la visualizzazione 3D del modello definito precedentemente e di acquisire e riprodurre le informazioni provenienti dai sensori implementati sul robot. In particolare, se il robot è dotato di un LiDAR, aggiungendo il tool <`LaserScan`> su Rviz si potranno vedere gli ostacoli che vede il robot attraverso il sensore.

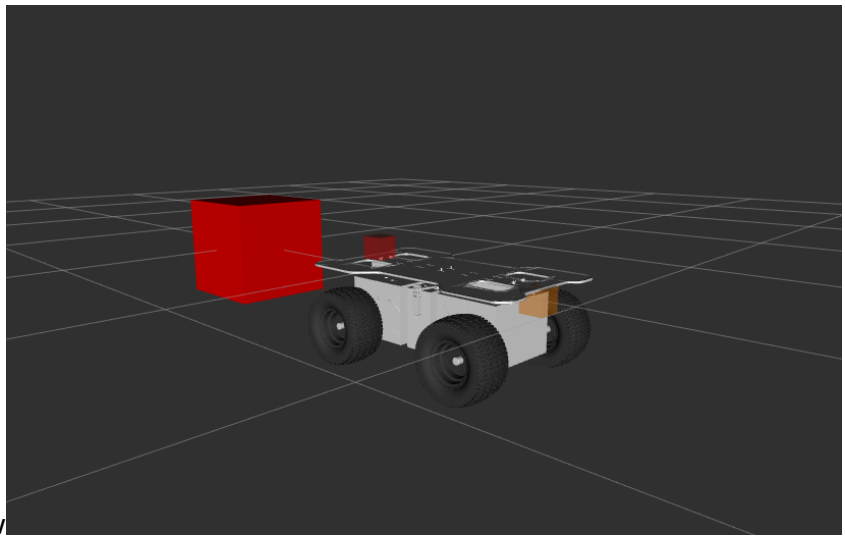
ew 

Figura 6: Rviz

Gazebo invece, è uno strumento per la simulazione e viene usato per la generazione di un ambiente reale, interno o esterno, nel quale il robot deve muoversi. Essendo un robot che verrà usato in ambito agricolo, si può simulare la presenza di alberi e altri oggetti tipici di tale spazio.

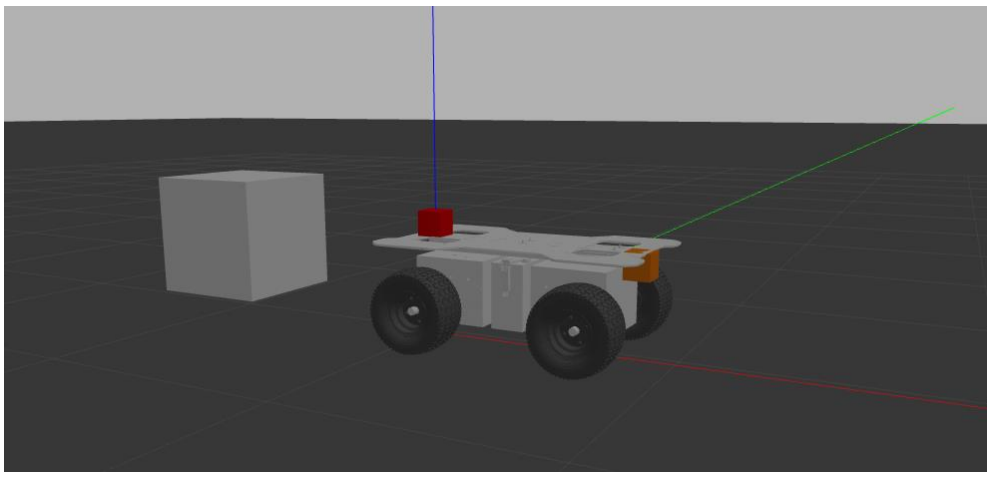


Figura 7: visualizzazione in Gazebo

Nell’URDF è necessario aggiungere un plugin di Gazebo che lo colleghi a ROS:

```
<gazebo>
  <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
    <robotNamespace>/MYROBOT</robotNamespace>
  </plugin>
</gazebo>
```

Figura 8: Esempio plugin Gazebo

4. SENSORI

Il robot è stato dotato di tre sensori per la percezione dell'ambiente esterno e per la localizzazione: LiDAR, GPS e IMU.

4.1 LiDAR

L'acronimo LiDAR (Light Detection and Ranging) identifica la tecnologia che permette di determinare la distanza da un oggetto illuminandolo con una luce LASER. È molto diffuso in ambito Automotive come strumenti visivo per realizzare veicoli a guida autonoma poiché riescono a individuare ostacoli sia fissi che mobili, come altri veicoli o pedoni.

Sapendo che la velocità di propagazione della luce è di 300.000 km/s, si può calcolare facilmente il tempo impiegato da un raggio luminoso per andare da una sorgente verso un bersaglio e per tornare indietro verso il rilevatore di luce. Questo principio di misura viene indicato come "Time of Flight (ToF)"

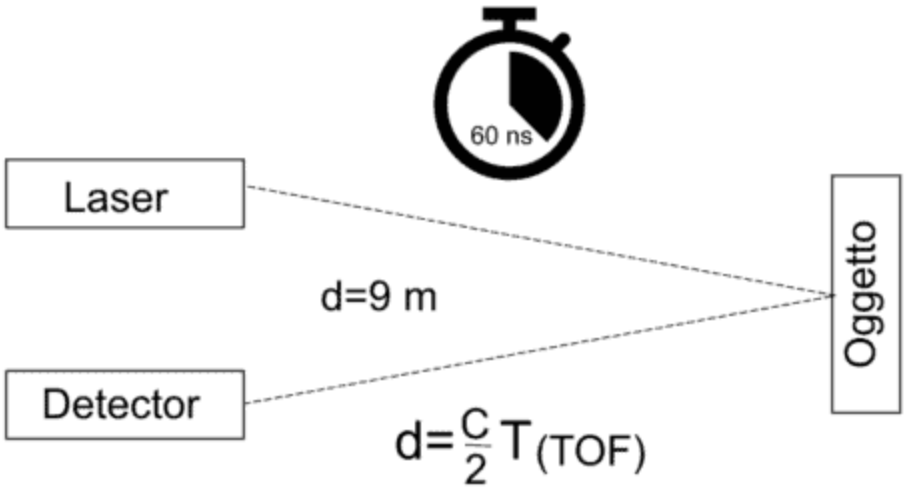


Figura 9: principio di funzionamento del LiDAR

Come si osserva dalla figura 8, si misura il tempo impiegato dal raggio di luce per raggiungere il bersaglio e tornare indietro. La distanza d si calcola moltiplicando il ToF per la velocità della luce C e dividendo per 2 (perché il raggio compie due volte la distanza). Per cui, la distanza è rappresentata dalla formula

$$d = \frac{C}{2} \cdot ToF$$

Ipotizzando che il tempo impiegato sia di 60 secondi, la distanza è pari a 9 metri.

Usare una tecnologia LiDAR è favorevole perché garantisce una misurazione veloce e precisa e sono di semplice utilizzo e installazione.

Per implementare il LiDAR sul robot è necessario come prima cosa creare un link e un joint che lo rappresenti esteticamente nel modello URDF e che crei un legame con il body centrale. Bisogna quindi, aggiungere un plugin

```
<!-- hokuyo -->
  <gazebo reference="laser_link">
    <sensor type="ray" name="head_hokuyo_sensor">
      <pose>0 0 0 0 0 0</pose>
      <visualize>true</visualize>
      <update_rate>50</update_rate>
      <ray>
        <scan>
          <horizontal>
            <samples>720</samples>
            <resolution>1</resolution>
            <min_angle>-1.570796</min_angle>
            <max_angle>1.570796</max_angle>
          </horizontal>
        </scan>
        <range>
          <min>0.10</min>
          <max>10.0</max>
          <resolution>0.01</resolution>
        </range>
        <noise>
          <type>gaussian</type>
          <mean>0.0</mean>
          <stddev>0.01</stddev>
        </noise>
      </ray>
      <plugin name="gazebo_ros_head_hokuyo_controller" filename="libgazebo_ros_gpu_laser.so">
        <topicName>robot/scan</topicName>
        <frameName>laser_link</frameName>
```

```
      </plugin>
    </sensor>
  </gazebo>
```

Figura 9: esempio di plugin per il laser

Tra i principali tag ci sono:

- `<gazebo reference = "/">`: nome del link del laser dichiarato nell'URDF;
- `<sensor type = "/" name = "/">`: tipo di sensore e nome unico da fare allo stesso;
- `<range>`: range massimo e minimo di visione del laser;
- `<topicName>`: nome del topic su cui pubblicherà il laser;
- `<frameName>`: nome del link a cui è collegato il laser.

I messaggi che verranno pubblicati sul topic saranno di tipo `sensor_msgs/LaserScan`.

Come spiegato precedentemente, per visualizzare cosa "vede" il robot attraverso il sensore, bisogna aggiungere in Rviz il tool LaserScan e settare il topic con il nome del topic dichiarato nel plugin, quindi sarà /robot/scan:

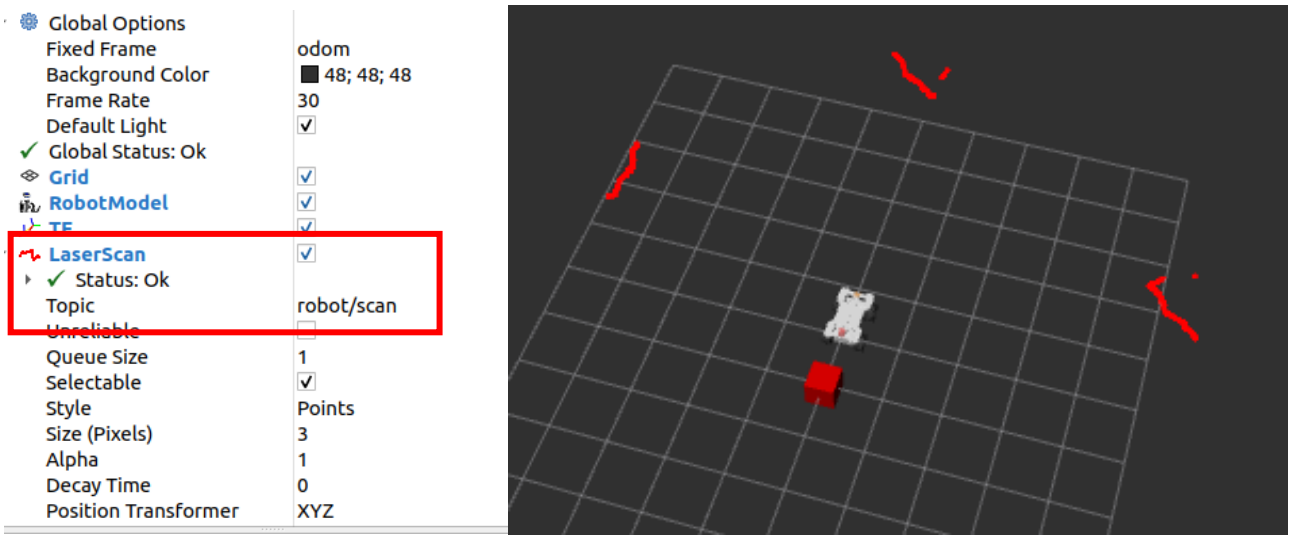


Figura 10: LaserScan

4.2 GPS

Il GPS è un sistema per il posizionamento globale, come si deduce dal suo nome (Global Positioning System). Permette di conoscere la longitudine e la latitudine di oggetti e persone tramite i satelliti che orbitano attorno alla terra.

Inizialmente usati come strumento per visualizzare le coordinate terresti, oggi è uno strumento imprescindibile per la robotica mobile e la navigazione autonoma.

Come già specificato per il Lidar, anche in questo caso è necessario inserire nell'URDF un link, un joint e il plugin. La sintassi di quest'ultimo è molto simile a quella descritta per il Lidar, infatti presenta principalmente gli stessi tag. Essendo un sistema per il posizionamento globale però, deve conoscere le coordinate a cui si trova il robot, per cui troviamo:

- `<referenceLatitude>`: latitudine di riferimento in gradi nord (di default 49.9)
- `<referenceLongitude>`: longitudine di riferimento in gradi est (di default 8.9)

Il topic sul quale pubblica il GPS è `/fix`. I messaggi che pubblica sono di tipo `sensor_msgs/NavSatFix` in cui ci sono latitudine, longitudine e altitudine.

4.3 IMU

L'IMU (Inertial Measurement Unit) è la combinazione di diversi sensori, in particolare di un accelerometro e di un giroscopio. L'IMU, montata su un veicolo, fornisce le accelerazioni lungo gli assi X, Y e Z e le rotazioni attorno agli assi X, Y e Z , quindi imbardata φ, beccheggio ϑ e rollio ψ.
Analogamente a quanto fatto per i sensori precedenti, viene aggiunto un link e un joint nell'urdf e inserito il plugin:

```
<gazebo reference="imu_link">
  <gravity>true</gravity>
  <sensor name="imu_sensor" type="imu">
    <always_on>true</always_on>
    <update_rate>50</update_rate>
    <visualize>true</visualize>
    <topic>robot/imu</topic>
    <plugin filename="libgazebo_ros_imu_sensor.so" name="imu_plugin">
      <topicName>imu</topicName>
      <bodyName>imu_link</bodyName>
      <updateRateHZ>10.0</updateRateHZ>
      <gaussianNoise>0.0</gaussianNoise>
      <xyzOffset>0 0 0</xyzOffset>
      <rpyOffset>0 0 0</rpyOffset>
      <frameName>imu_link</frameName>
      <initialOrientationAsReference>false</initialOrientationAsReference>
    </plugin>
    <pose>0 0 0 0 0 0</pose>
  </sensor>
```

```
  </gazebo>
```

Figura 10: esempio di plugin per l'IMU

Il topic sul quale il sensore pubblica messaggi di tipo `sensor_msgs/Imu` è `/imu`.

5. CONTROLLO

Dopo aver settato correttamente le caratteristiche del robot e aver collegato ROS e Gazebo, è il momento di far muovere il robot.

Inizialmente è stato controllato manualmente tramite un'interfaccia grafica (`rqt_robot_steering`) che permette di pubblicare la velocità lineare e angolare del robot sui giunti delle ruote.

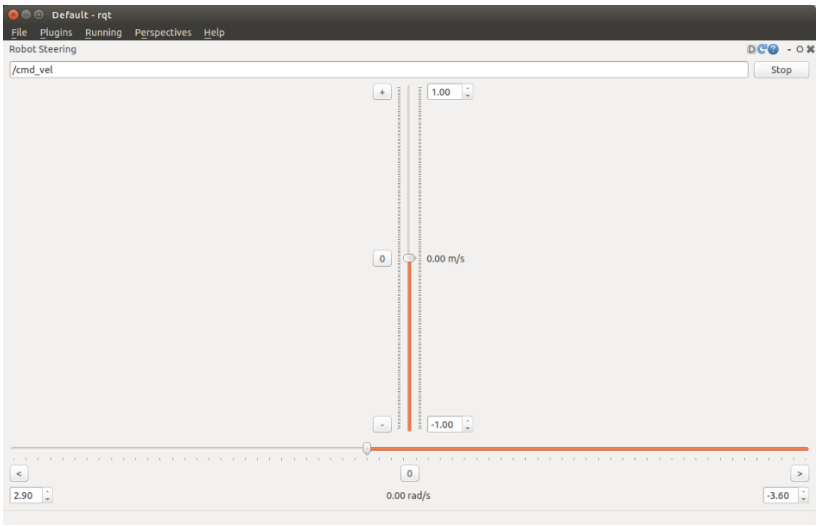


Figura 11: rqt_robot_steering

La gui pubblica sul topic `/cmd_vel` messaggi di tipo `geometry_msgs/Twist`, cioè le 3 velocità lineari e le 3 velocità angolari.

Per fare ciò però, è stato necessario usare il `diff_drive_controller`. Viene effettuato un controllo di velocità che viene sdoppiato e inviato sulle due ruote motrici. Tale controllore può essere applicato anche a robot *skid steering*, caso in esame, con cui si controllano separatamente le due o più ruote di sinistra e le due o più ruote di destra, come se fossero un'unica ruota per lato.

Il controller ha dei parametri da settare:

```
type: "diff_drive_controller/DiffDriveController"
  left_wheel: ['joint_base_wheel_front_left', 'joint_base_wheel_rear_left']
  right_wheel: ['joint_base_wheel_front_right', 'joint_base_wheel_rear_right']
  publish_rate: 50
  pose_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.03]
  twist_covariance_diagonal: [0.001, 0.001, 0.001, 0.001, 0.001, 0.03]
  cmd_vel_timeout: 0.25
```

```
# Odometry fused with IMU is published by robot_localization, so
# no need to publish a TF based on encoders alone.
enable_odom_tf: false

# Wheel separation and radius multipliers
wheel_separation_multiplier: 2.2 # default: 1.0
wheel_radius_multiplier    : 1.0 # default: 1.0

linear:
  x:
    has_velocity_limits    : true
    max_velocity           : 1.0   # m/s
    has_acceleration_limits: true
    max_acceleration       : 3.0   # m/s^2
angular:
  z:
    has_velocity_limits    : true
    max_velocity           : 2.0   # rad/s
    has_acceleration_limits: true
    max_acceleration       : 6.0   # rad/s^2
```

Figura 12: diff_drive_controller

Tra i principali ci sono:

- `<left_wheels>`: nome del giunto o lista di giunti della ruota di sinistra
- `<right_wheels>`: nome del giunto o lista di giunti della ruota di destra
- `<publish_rate>`: frequenza alla quale viene pubblicata l'oodmetria
- `<base_frame_id>`: nome del frame base
- Velocità lineare:
  `<max_velocity>` `<min_velocity>` `<max_acceleration>` `<min_acceleration>`
- Velocità angolare:
  `<max_velocity>` `<min_velocity>` `<max_acceleration>` `<min_acceleration>`

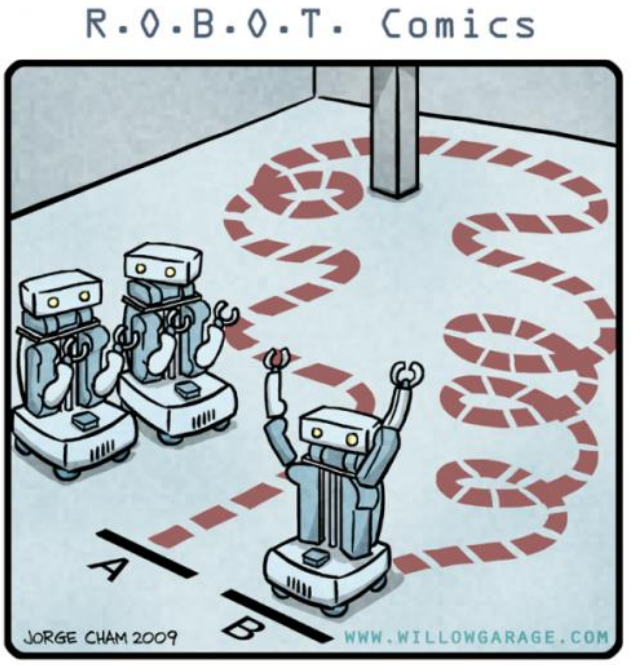
Il controller si sottoscrive al topic `/cmd_vel`, topic sul quale pubblica l'interfaccia grafica, e pubblica messaggi di tipo `nav_msgs/Odometry` sul topic `odom`.

Si osserva che nel file `control.launch` del progetto, la gui pubblica sul topic `heros_velocity_controller/cmd_vel` perché al controller è stato dato il nome di `heros_velocity_controller`.

6. NAVIGAZIONE AUTONOMA

Step successivo è quello di implementare la navigazione autonoma.

A tal fine è stato usato il NAVIGATION STACK. Il Navigation Stack è un insieme di nodi e algoritmi che prende in input dati provenienti dall'odometria e dai sensori e produce in output le velocità da mandare agli attuatori affinché il robot raggiunga la posizione desiderata.



Sebbene il navigation stack sia stato progettato per essere il più generico possibile e applicabile a diversi robot, sono necessari tre requisiti hardware principali che ne limitano l'uso:

1. Si può usare solo con robot differenziali o olonomi;
2. È necessario che abbia un sensore laser utilizzato per la creazione della mappa e per la localizzazione;
3. È stato progettato per robot di forma quadrata, quindi le sue prestazioni sono migliori su robot squadrati o circolari. Non è escluso l'uso su robot di forma arbitraria.

Il Navigation Stack non richiede necessariamente la conoscenza di una mappa per operare; infatti, l'idea principale di questo progetto è proprio quella di far muovere il robot senza conoscerla.

La pianificazione del percorso si divide in due parti:

- Global planner: trova il percorso ottimale conoscendo a priori l'ambiente in cui deve muoversi il robot e la posizione degli ostacoli statici;

- Local planner: elabora in tempo reale il nuovo percorso che eviti gli ostacoli dinamici

6.1 COSTMAP

IL Navigation Stack usa la COSTMAP per immagazzinare informazioni circa gli ostacoli presenti nel mondo usando i dati provenienti dai sensori e le informazioni della mappa statica (ove presente).

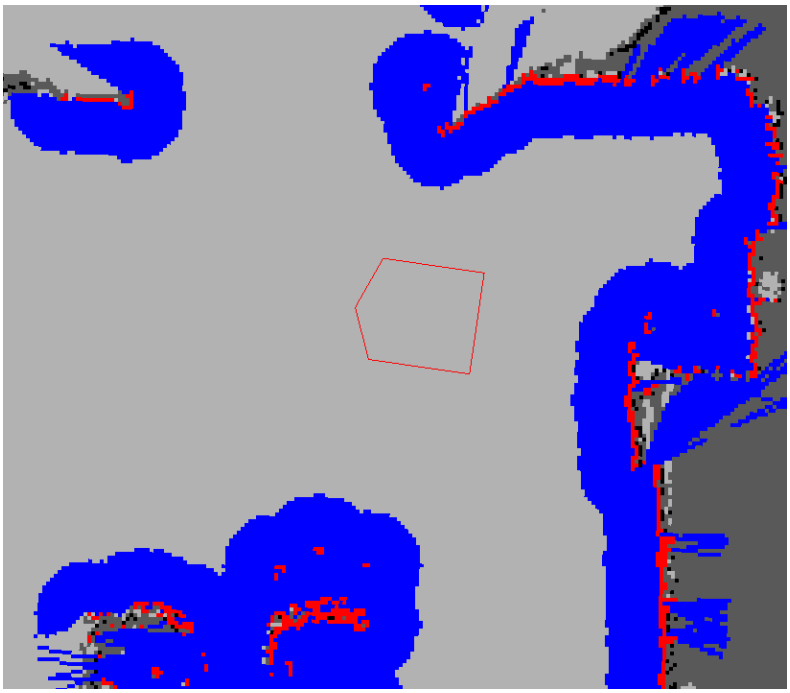


Figura 13: esempio di costmap in RVIZ

Il pacchetto `costmap_2d` fornisce una struttura configurabile che salva le informazioni riguardanti i luoghi in cui il robot può navigare sotto forma di griglia di occupazione. Le celle possono assumere solamente tre stati: libera, occupata, sconosciuta. Ad ogni stato è assegnato un valore di costo. In particolare, in figura possiamo distinguere tre colori: rosso indica l'ostacolo, blu indica l'ostacolo "gonfiato" con il raggio inscritto del robot, grigio indica lo spazio inesplorato. Sempre in figura, è possibile distinguere un poligono rosso che rappresenta l'impronta del robot. Affinché non vi sia collisione con l'ostacolo, l'impronta non deve mai intersecare le celle rosse e il centro del robot non deve mai toccare una cella blu.

Sono stati definiti tre file di configurazione in cui sono stati settati i parametri della costmap.

I parametri generali, comuni sia alla GLOBAL COSTMAP che alla LOCAL COSTMAP, sono:

```
robot_base_frame: base_link
update_frequency: 4.0
publish_frequency: 4.0
```

```
obstacle_range: 5.5
raytrace_range: 6.0
footprint: [[1.11,0.5], [-1.40,0.5], [-1.40, -0.5], [1.11, -0.5]]
inflation_radius: 1
resolution: 0.05


observation_sources: laser_scan_sensor


laser_scan_sensor: {sensor_frame: laser_link, data_type: LaserScan, topic: robot
/scan, marking: true, clearing: true}
```

Figura 14: parametri comuni costmap

- <robot_base_frame>: definisce il frame a cui deve fare riferimento la costmap per la base del robot;
- <update_frequency>: definisce la frequenza, in Hz, a cui si aggiornerà la costmap;
- <publish_frequency>: è la frequenza, in Hz, a cui viene pubblicata la costmap;
- <obstacle_range>: definisce la distanza massima tra la base e l'ostacolo. Se l'ostacolo si troverà oltre questa distanza, non verrà segnato nella mappa (anche se il laser lo vede);
- <raytrace_range>: impostando questo parametro, in metri, il robot tenterà di liberare davanti a sé fino al valore impostato in base a una lettura del sensore;
- <footprint>: viene impostata l'impronta del robot, o il raggio se il robot è circolare;
- <inflation_radius>: è la distanza massimo dall'ostacolo. Impostando questo parametro ad un certo valore, il robot tratterà tutti i percorsi che si trovano a distanze uguali o superiori a quella dichiarata, con lo stesso costo dell'ostacolo;
- <observation_sources>: elenco di sensori che passeranno informazioni alla costmap;
- <laser_scan_sensor>: vengono impostati dei parametri per il sensore dichiarato nella riga precedente.

Successivamente è stato creato un file yaml per i parametri di configurazione della GLOBAL COSTMAP:

```
global_costmap:
  global_frame: odom
  rolling_window: true
```

Figura 15: parametri global costmap

- `<global_frame>:` definisce in quali coordinate deve essere eseguita la costmap. Nel nostro caso specifico, il global frame verrà settato su odom poiché non disponiamo della mappa;
- `<rolling_window>:` se non si usa una mappa statica, come in questo caso, il parametro va impostato a true.

Infine, l'ultimo file da settare è quello relativo alla LOCAL COSTMAP:

```
local_costmap:
  global_frame: odom
  rolling_window: true
  width: 6.0
  height: 6.0
```

Figura 16: parametri local costmap

- `<width>:` larghezza della costmap
- `<heigth>:` altezza della costmap

Gli stessi parametri possono essere settati anche per la global costmap. Nello specifico, non vengono impostati nel rispettivo file .yaml, bensì nel file `navigation.launch`.

La mappa dei costi del progetto appare come in figura:

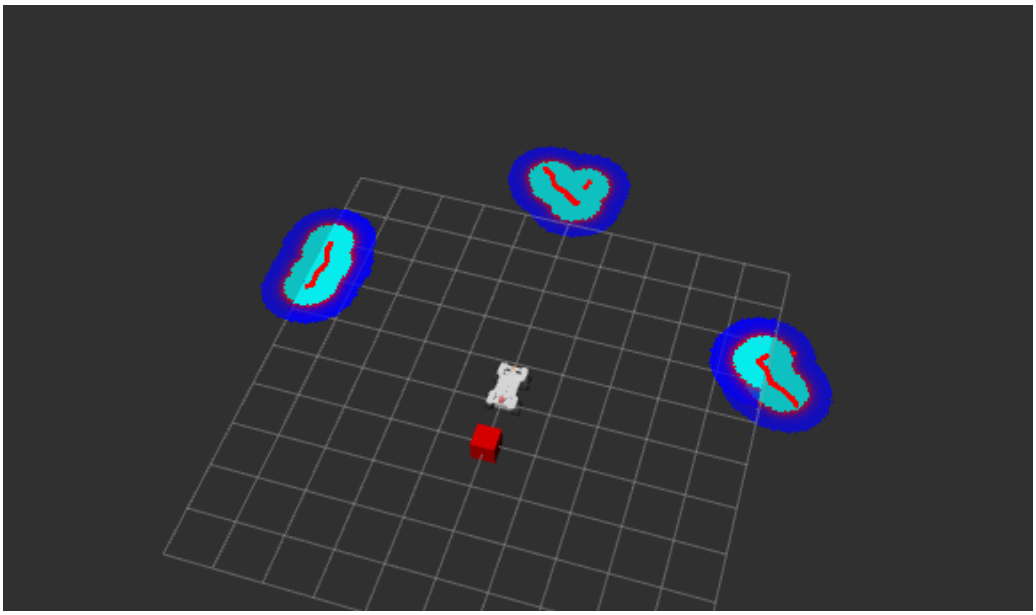


Figura 17: costmap

6.2 BASE LOCAL PLANNER

Il `base local planner` è un pacchetto che si occupa del calcolo dei comandi di velocità da mandare all'attuatore. Sulla base del percorso da seguire e della costmap, il controller produce i comandi di velocità affinché il robot segua il percorso desiderato.

Lungo il percorso il planner crea una funzione di valore, rappresentata tramite una grid map. Il controller sulla base dei costi delle celle della griglia attribuiti dalla funzione determina le velocità $dx, dy, d\theta$ da inviare al robot.

È necessario settare delle opzioni di configurazione in base alle specifiche del nostro robot, e questo verrà fatto all'interno del file `planner.yaml`:

```
DWAPlannerROS:

  acc_lim_x: 2.5
  acc_lim_y: 0
  acc_lim_th: 3.2

  max_vel_x: 0.5
  min_vel_x: 0.0
  max_vel_y: 0.0
  min_vel_y: 0.0

  max_vel_th: 1.0
  min_vel_th: 0.2

  yaw_goal_tolerance: 0.1
  xy_goal_tolerance: 0.2

  holonomic_robot: false
```

Figura 18: parametri local planner

È facile intuire che la prima sezione riguarda i limiti di accelerazione del robot, mentre la seconda i limiti di velocità, la terza le tolleranze quando si raggiunge l'obiettivo. L'ultima opzione invece, va posta su false se il robot è non olonomo come nel caso in esame.

6.3 GLOBAL PLANNER

Si avvale dell'uso del pacchetto `Navfn` per il calcolo del percorso per il robot. Sulla base della costmap e utilizzando l'algoritmo di Dijkstra's (algoritmo usato per cercare i cammini minimi in un grafo), il planner trova un percorso di "costo minimo" che collega un punto iniziale a un punto finale. Sempre nel file `move_base.yaml`, sono state impostate due opzioni per questo planner:

```
allow_unknown: true
default_tolerance: 0.1
```

Figura 19: parametri global planner

- `<allow_unknown>:` è un parametro che, se settato a true, consente all'algoritmo di creare percorsi in spazi sconosciuti;
- `<default_tolerance>:` specifica una tolleranza sul punto di arrivo per il planner il quale tenterà di creare un percorso che sia il più vicino possibile all'obiettivo desiderato nei limiti di tale tolleranza.

6.4 ROBOT LOCALIZATION

Dopo avere impostato tutti i parametri della navigation stack, il robot non era ancora in grado di navigare correttamente. Come si può evincere dalla seguente figura

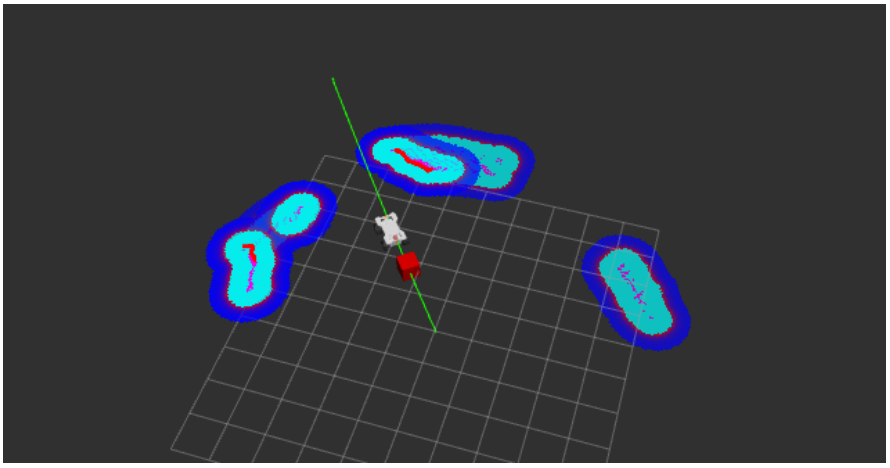


Figura 20: navigazione sbagliata

vi sono dei problemi con la localizzazione del robot e degli ostacoli all'interno dello spazio.

Per correggere la localizzazione e migliorare la navigazione allora, è stato utilizzato il pacchetto `robot_localization`, il quale fornisce una stima dello stato non lineare tramite l'utilizzo combinato di dati provenienti da molteplici sensori. Nello specifico, il robot è stato dotato di Imu,

che pubblicherà messaggi di tipo `sensor_msgs/Img`, legati all'inerzia del robot, e gli encoder del controllore, che pubblicheranno messaggi di tipo `nav_msgs/Odometry`.

All'interno del file `localization.yaml` sono stati configurati i parametri relativi all'imu e agli encoder:

- `<frequency>`;
- `<frame>`:
  - `odom_frame: odom`;
  - `map_frame: map` : se il sistema non ha questo parametri, bisogna rimuoverlo e impostare il `world_frame` con lo stesso valore dell'`odom_frame`;
  - `base_link_frame: base_link` : definisce il sistema di coordinate fissate al robot;
  - `world_frame: odom`;
- `<two_d_mode>`: da settare a `true` se il robot sta operando in un ambiente planare (2D) Questo elimina le variabili 3D (Z, rollio, beccheggio e le rispettive velocità e accelerazioni);
- `<sensor>`: bisogna definire per ogni sensore dei parametri
  - `odom0: heros_velocity_controller/odom`
  - `imu0: robot/imu`

    Il numero per ogni nome del parametro inizia da 0 e deve essere incrementato sequenzialmente se vi sono più sensori dello stesso tipo (es odom0, odom1, etc). Il valore per ogni parametro rappresenta il topic di quel sensore (es topic dell'imu è imu/data);
- `<sensor_config>`: Per ciascuno dei messaggi del sensore definiti al punto precedente, bisogna specificare quali variabili di tali messaggi devono essere unite nella stima dello stato
  - `odom0_config: […]`
  - `imu0_config: […]`

    NB: l'ordine dei valori booleani è: $X, Y, Z, roll, pitch, yaw, \dot{X}, \dot{Y}, \dot{Z}, \dot{roll}, \dot{pitch}, \dot{yaw}, \ddot{X}, \ddot{Y}, \ddot{Z}$.

Dopo avere introdotto questo pacchetto, la navigazione è migliorata notevolmente come visibile in figura:

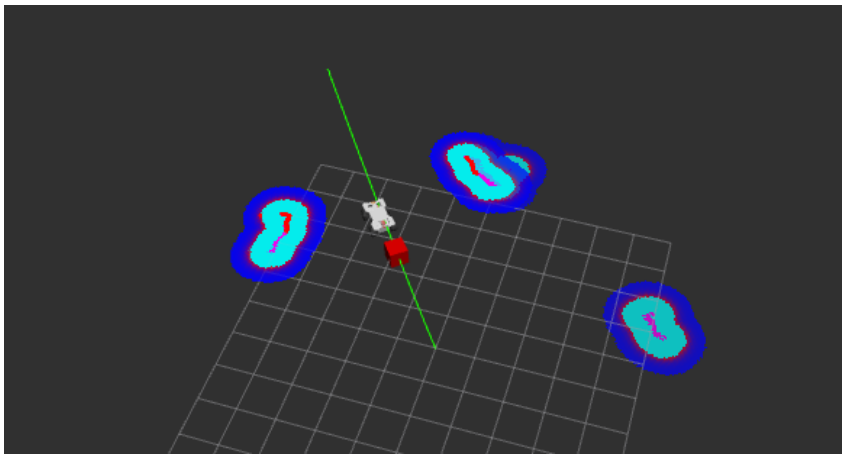


Figura 21: navigazione migliorata

7. NODO

Ultimo step del progetto è quello di scrivere un nodo che pubblichi sul topic `/move_base_simple/goal` la posa che deve raggiungere il robot.

È stato così definito:

```
1  import rospy
2  from geometry_msgs.msg import PoseStamped
3
4  rospy.init_node('GoalPublisher', anonymous=True)
5
6  sendGoal = rospy.Publisher('/move_base_simple/goal', PoseStamped, queue_size=10)
7
8
9  def movebase():
10
11     goal = PoseStamped()
12     goal.header.frame_id = "odom"
13     goal.header.stamp = rospy.Time.now()
14     goal.pose.position.x = 3.0
15     goal.pose.position.y = 0.0
16     goal.pose.orientation.w = 1.0
17     rate = rospy.Rate(2.0)
18
19     while not rospy.is_shutdown():
20         sendGoal.publish(goal)
21         rate.sleep()
22
23
24
25 if __name__ == '__main__':
26     try:
27         movebase()
28     except rospy.ROSInterruptException:
29         pass
```

Figura 22: nodo GoalPublisher

- Riga 5: dichiarazione nome del nodo;
- Riga 8: inizializzazione publisher che deve pubblicare sul topic `/move_base_simple/goal` dei messaggi geometrici di tipo `PoseStamped`;
- Riga 16, 17, 18: impostazione della posa finale da raggiungere.

8. CONCLUSIONI E LAVORO FUTURO

Obiettivo del progetto era quello di implementare su un robot la navigazione autonoma. Ciò è stato possibile grazie all'utilizzo di ROS e alla Navigation Stack.

Il robot, infatti, tramite il comando `2D Nav Goal` di Rviz, è in grado di navigare autonomamente dalla posizione attuale a quella desiderata. Inoltre, è in grado di evitare gli ostacoli che incontra durante il percorso grazie alla mappa dei costi implementata dal pacchetto `costmap_2d`.

Per una simulazione più reale, in Gazebo è stato ricreato tramite l'inserimento di alberi, un ambiente simile a quello in cui il robot andrà a lavorare

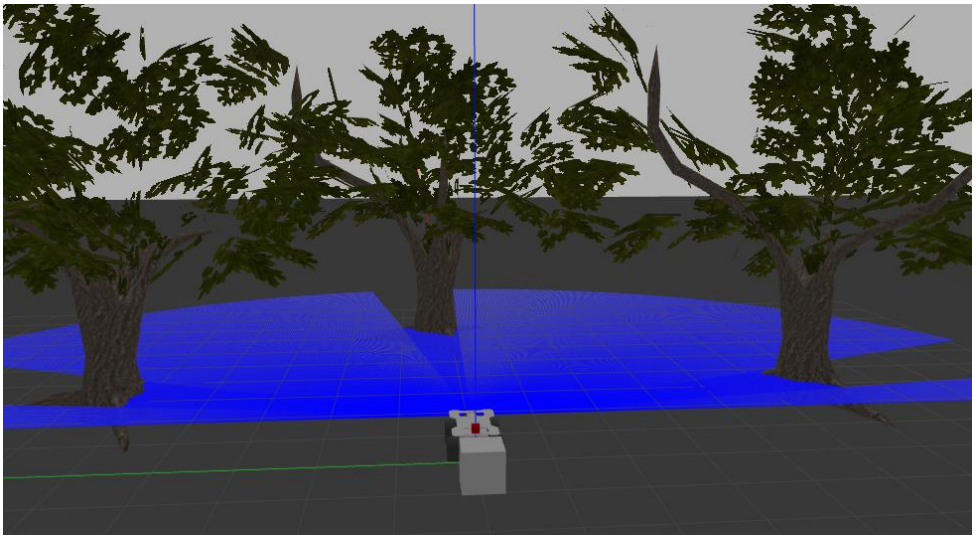


Figura 23: Ambiente simulato

Come lavoro futuro mi auspico di creare un nodo che permetta al robot di navigare in funzione di coordinate gps reali. Le coordinate verranno usate come check point, quindi una volta raggiunta la i-esima posizione, il nodo aggiornerà l'obiettivo e il robot avanzerà fino al raggiungimento della i+1-esima posizione.

A tal fine, per semplificare il lavoro futuro, è stato già scritto il nodo che pubblica la posizione gps target su un topic preposto:

```python
import rospy
from sensor_msgs.msg import NavSatFix

rospy.init_node('GpsTargetPublisher', anonymous=True)
gps = rospy.Publisher('/gps_target', NavSatFix, queue_size=10)


def got_position():

    fix=NavSatFix()
    fix.header.frame_id="gps_target"
    fix.status.status=0
    fix.status.service=1

    fix.latitude=40.79882399539434
    fix.longitude=16.92312700802616
    rate = rospy.Rate(2.0)

    while not rospy.is_shutdown():
        gps.publish(fix)
        rate.sleep()

if __name__ == '__main__':
    try:
        got_position()
    except rospy.ROSInterruptException:
        pass

```

Figura 24: nodo GpsTargetPublisher

Al nodo è stato dato il nome `GpsTargetPublisher`; esso pubblica messaggi di tipo `fix` sul topic `/gps_target`. La posizione da raggiungere viene dichiarata dalle righe 15 e 16 in cui viene impostata la latitudine e longitudine desiderata. Tali punti si potrebbero aggiornare ogni qual volta il robot raggiunge la coordinata.

Una navigazione basata su questa logica potrebbe essere utile per l'aratura di campi seminativi o anche nei vigneti o negli uliveti.

Le coordinate gps target posso essere rilevate o sul campo, se si dispone degli strumenti adatti a tale rilevamento, o per una simulazione più semplice è possibile avvalersi di Google Earth che permette la creazione di file .kml (file in cui vengono memorizzati dati geografici e contenuti associati a Google Earth).

Un esempio di questo secondo approccio è rappresentato dalla seguente figura:

Figura 25: Esempio Google Earth

9. RIFERIMENTI

[1] https://github.com/innokrobotics/innok_heros_description

[2] https://robots.ros.org/innok-heros/

[3] http://wiki.ros.org/urdf

[4] http://wiki.ros.org/urdf/Tutorials/Adding%20Physical%20and%20Collision%20Properties%20to%20a%20URDF%20Model

[5] http://wiki.ros.org/urdf/Tutorials/Building%20a%20Movable%20Robot%20Model%20with%20URDF

[6] http://wiki.ros.org/urdf/Tutorials/Using%20a%20URDF%20in%20Gazebo

[7] https://consystem.it/faq/tecnologia-lidar-che-cosa-e-come-funziona/

[8] http://gazebosim.org/tutorials?tut=ros_gzplugins

[9] http://wiki.ros.org/diff_drive_controller

[10] http://wiki.ros.org/navigation

[11] http://wiki.ros.org/costmap_2d

[12] http://wiki.ros.org/navigation/Tutorials/RobotSetup

[13] http://wiki.ros.org/base_local_planner

[14] http://wiki.ros.org/navfn

[15] http://docs.ros.org/en/melodic/api/robot_localization/html/index.html