# Differential Drive

Un **differential-drive** è uno dei più comuni e semplici esempi di robot mobile. Esso consiste principalmente di 2 ruote principali, indipendenti (ognuna dotata di attuazione separata), montate lungo un asse comune di rotzione. Oltre alle due ruote principali vengono montate una o più ruote ausiliare non motorizzate con il solo scopo di stabilizzare fisicamente la struttura meccanica. Il sistema di curvatura è realizzato andando ad attuare diversamente le due ruote motrici, così che il veicolo ruoti verso la ruota più lenta. Così come molti altri veicoli classici, il differential-drive è un sistema *non olonomo*, il che vuol dire che vi sono dei vincoli meccanici che ne limitano il cambio di posa in maniera arbitraria, nel suo caso questi si traducono nell'impossibilità di movimenti traslazionali laterali.

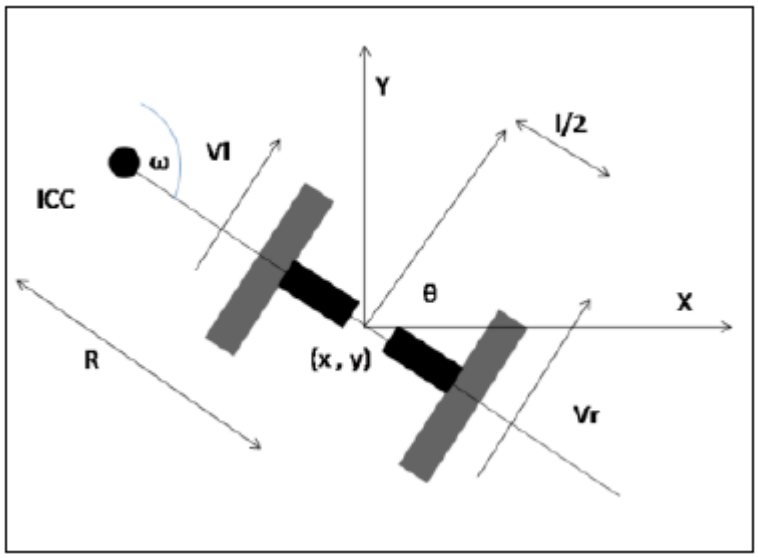
## Modello Cinematico

La **cinematica** è la branca della meccanica che studia il moto dei corpi indipendentemente dalle cause che lo provocano o lo modificano. Ciò implica che in tale fase di modellazione non vengono considerate le forze che inducono il movimento, bensì vengono principalmente effettuate analisi di tipo geometrico per individuare le diverse relazioni che intercorrono tra le diverse componenti del sistema.
In generale, un corpo libero nello spazio ha 6 **gradi di libertà** (**DOF** - *degrees of freedom*) espressi da un insieme di coordinate generalizzate dette **posa** (*posizione + orientamento*). Un possibile insieme di variabili atte a identificare la posa del corpo nello spazio è rappresentato da tre *coordinate cartesiane* (*x, y, z*) per idintificare la posizione, e tre *coordinate angolari* (*roll, pitch, yaw*) per identificare l'assetto del corpo. Le tre coordinate angolari - note come **angoli di Eulero** - esprimono l'assetto del corpo lungo i suoi tre assi principali: il **rollio** (*roll*) identifica la rotazione intorno l'asse longitudinale (asse che va dalla coda al muso), il **beccheggio** (*pitch*) misura l'inclinazione lungo l'asse trasversale (quanto il muso è piegato in avanti), l'**imbardata** (*yaw* o spesso anche *heading*) è l'angolo di un corpo intorno a un asse verticale passante per il suo baricentro.
Convenzionalmente i movimenti di rollio indicano le rotazioni intorno l'asse *x*, il beccheggio lungo l'asse *y*, mentre l'imbardata intorno l'asse *z*.
Nel caso del differential-drive il movimento permesso avviene all'interno del piano *x-y*, il che rende sufficienti alla completa descrizione della sua posa un insieme di tre variabili generalizzate: le coordinate cartesiane $x$ e $y$ per identificare la posizione e l'angolo di orientamento $\theta$, che esprime l'angolo tra l'asse x del sistema di riferimento inerziale e la direzione di avanzamento del robot.
In a differential-drive robot the motion can be controlled by adjusting the velocity of the two independently controlled motors attached on the wheels.
Nella robotica mobile, le **equazioni della cinematica diretta** vengono utilizzate per conoscere la posizione del robot nel sistema di coordinate globali a partire dalle velocità dell'oggetto nel sistema di coordinate solidali al corpo. L'obiettivo è determinare un modello a tempo discreto che metta in relazione la posa del robot nel tempo in funzione delle sue velocità istantanee.

$$\begin{bmatrix} x_{k+1} \\ y_{k+1} \\ \theta_{k+1} \end{bmatrix} = f\left( \left[ x_k,\ y_k,\ \theta_k \right]^{\mathrm{T}} \right) + g\left( \left[ v_k,\ \omega_k \right]^{\mathrm{T}} \right)$$

Gli ingressi del modello cinematico sono la velocità d'avanzamento $v_k$ e la velocità angolare $\omega_k$ lungo l'asse verticale che passa nel baricentro del robot.
Le equazioni cinematiche possono essere ricavate a partire da semplici considerazioni di natura fisica e

geometrica. Durante una generica fase di curvatura, il robot sta eseguendo una rotazione intorno a un punto virtuale posizionato lungo l'estensione dell'asse di rotazione comune alle due ruote, tale punto è detto **Centro Istantaneo di Curvatura (ICC)**.




Purtroppo però attuatori e sensori difficilmente lavorano con tali grandezze, perciò si rende necessario risalire a tali valori a partire dalle informazioni che attuatori e sensori possono fornire, come ad esempio le velocità di rotazione delle singole ruote $\omega_k^{(R)}$,$\omega_k^{(L)}$ e le rispettive velocità lineari associate $v_k^{(R)}$, $\mathrm{v}_k^{(L)}$ . È nota a tutti la relazione base che lega velocità lineare e angolare:

$$\mathrm{v}_k = r \times \omega_k$$

dove $r$ è la distanza del punto dall'asse di rotazione, e, nel caso di una ruota, il suo raggio.

Se si assume che sulle ruote siano montati dei sensori in grado di fornire i valori delle velocità di rotazione delle singole ruote, è possibile individuare le corrispettive velocità lineari.

> **ENCODER**
>
> Un encoder rotazionale con risoluzione $N$ è caratterizzato dalla seguente relazione:
> $2\pi : N = \Delta\theta : n$
> dove $n$ è il numero di impulsi (*ticks*) contati in un intervallo di tempo $\Delta t$. Le velocità angolari e lineari per la singola ruota possono essere calcolate mediante le formule:
> $\omega_k^{(i)} = \frac{2\pi n^{(i)}}{N\Delta t} \qquad v_k^{(i)} = \frac{2\pi r n^{(i)}}{N\Delta t}$

La velocità angolare intorno all centro istantaneo di curvatura è la stessa per entrambe le ruote, ciò che cambia è la distanza di ognuna di queste dall'ICC, nello specifico la differenza è data dalla lunghezza $\ell$ che separa le due ruote:

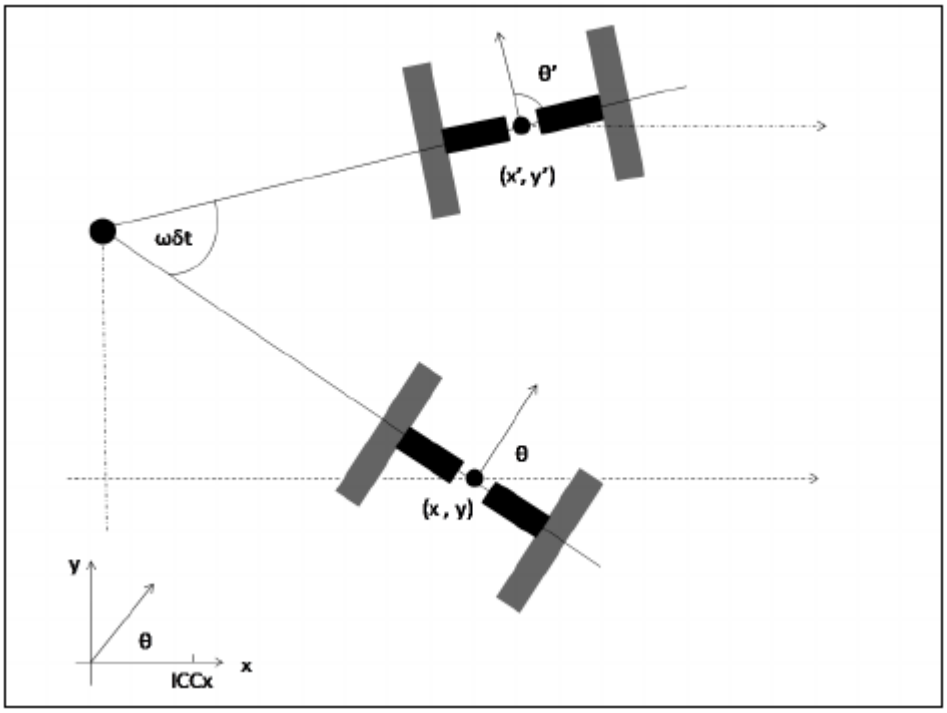
$$\begin{cases} v_k^{(L)} = (R - \frac{\ell}{2}) \times \omega_k \\ v_k^{(R)} = (R + \frac{\ell}{2}) \times \omega_k \end{cases}$$

Tale sistema è composto da due equazioni e due incognite, da cui è possibile ottenere:

$$\begin{cases} R = \frac{v_k^{(R)} + v_k^{(L)}}{v_k^{(R)} - v_k^{(L)}} \frac{\ell}{2} \\ \omega_k = \frac{v_k^{(R)} - v_k^{(L)}}{\ell} \end{cases} \quad \Longrightarrow \quad \mathrm{v}_k = \frac{v_k^{(R)} + v_k^{(L)}}{2}$$

Per ricavare delle equazioni cinematiche che descrivono il modello è necessario preliminarmente calcolare le coordinate dell'ICC a partire dalla posizione del robot e dalle sue velocità. Tale calcolo può ssere ricavato a partire da semplici considerazioni trigonometriche

$$ICC_x = x_k - R\cos(\tfrac{\pi}{2} - \theta_k) = x_k - R\sin(\theta_k)$$
$$ICC_y = y_k + R\sin(\tfrac{\pi}{2} - \theta_k) = y_k + R\cos(\theta_k)$$




A partire da queste ultime relazioni, ipoizzando che le velocità si mantengano costanti durante l'intervallo di tempo $\Delta t$, è possibile scrivere la posizione del robot all'istante successivo come:

$$\begin{cases} x_{k+1} = ICC_x + R\cos(\tfrac{\pi}{2} - \theta_{k+1}) = ICC_x + R\sin(\theta_{k+1}) \\ y_{k+1} = ICC_y + R\sin(\tfrac{\pi}{2} - \theta_{k+1}) = ICC_y + R\cos(\theta_{k+1}) \\ \theta_{k+1} = \theta_k + \omega_k \Delta t \end{cases}$$

Rimpiazzando le ccordinate dell'ICC e scrivendo R in funzione delle velocità lineare e angolare, si ottengono le equazioni della cinematica del differential-drive:

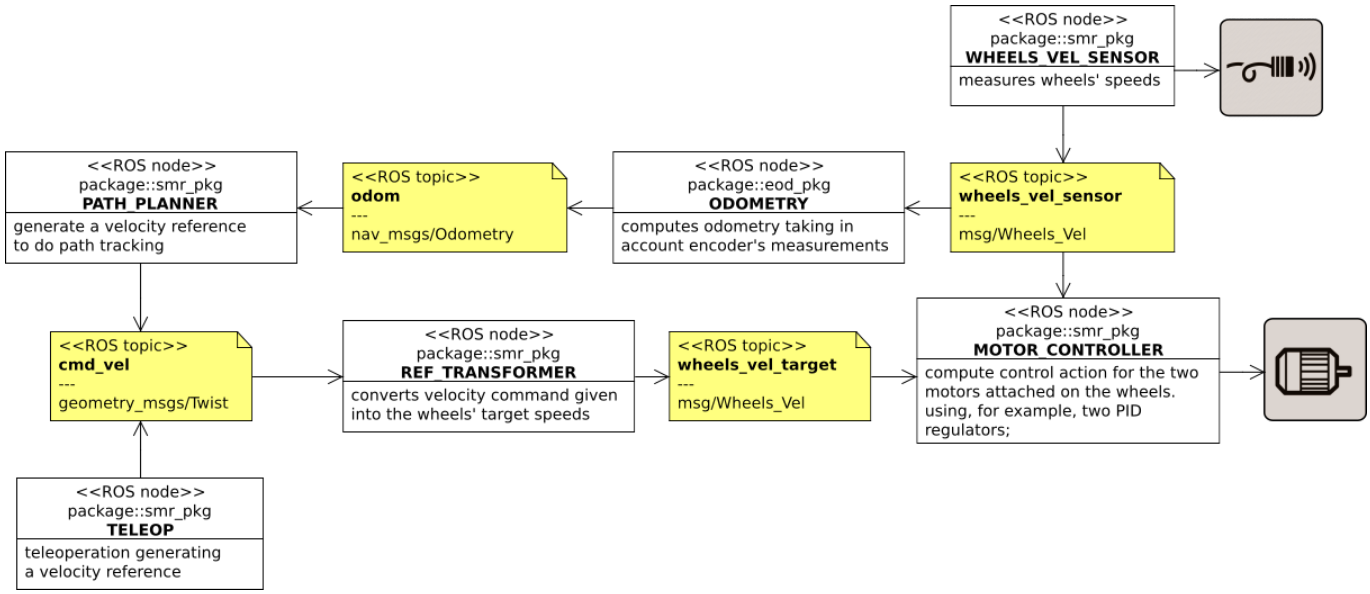
$$\begin{cases} x_{k+1} = x_k + \frac{v_k}{\omega_k}\Big(\sin(\theta_k + \omega_k \Delta t) - \sin(\theta_k)\Big) \\ y_{k+1} = y_k + \frac{v_k}{\omega_k}\Big(\cos(\theta_k + \omega_k \Delta t) - \cos(\theta_k)\Big) \\ \theta_{k+1} = \theta_k + \omega_k \Delta t \end{cases}$$

Un caso particolare è dato dal caso in cui $\omega_k = 0$, ovvero quando il robot sta procedendo in linea retta, in tale scenario le equazioni diventano:

$$\begin{cases} x_{k+1} = x_k + v_k \Delta t \cos(\theta_k) \\ y_{k+1} = y_k + v_k \Delta t \sin(\theta_k) \\ \theta_{k+1} = \theta_k \end{cases}$$

## Architettura ROS

In questa sezione è riportato un esempio di sviluppo di una semplice architettura software basata su ROS per realizzare le principali componenti che compongono un robot differential-drive. Lo scopo principale è quello si progettare un insieme di nodi ROS che modellino le diverse entità necessarie al funzionamento del robot: dall'attuazione delle due ruote indipendenti, alla lettura dei sensori, alla stima della posizione attuale del veicolo per la corretta navigazione sul piano. La principale funzionalità è quella di riuscire a seguire dei riferimenti di velocità $[v_k, \omega_k]$ calcolati in modo da far seguire al robot un determinato percorso.




### STRUTTURA COMUNE PER I NODI

Con lo scopo di migliorare la leggibilità e la riusabilità del codice, è proposta di seguito una stuttura standard che può essere adoperata per implementare le singole entità modellate dai nodi ROS. Tale struttura standard prevede l'implementazione di ciascun nodo attraverso un'apposita classe Python, nel cui `__init__` viene inizializzato il nodo ROS e tutti i parametri, topic, servizi e azioni che saranno utilizzati; e un metodo `loop` rapresentante il *ciclo di vita* del nodo, che sarà eseguito con una frequenza pre determinata.

```
import rospy
class NodeName:
  def __init__(self):
    rospy.init_node('node_name')
    # ------- PARAMETERS -------
    self.freq = rospy.get_param('freq', 10)
    self.rate = rospy.Rate(self.freq)
    # ------- VARIABLES -------
    self.var1 = ...
    self.var2 = ...
```

```
    # -------   TOPICS   -------
    rospy.Subscriber(...
    self.publisher = ...
    # -------   SERVICES   -------
    ...
    # -------   ACTIONS   -------
    ...

  # -------   LIFE CYCLE   -------
  def loop(self):
    while not rospy.is_shutdown():
      ...  # node operations
      self.rate.sleep()

  # -------   CALLBACK   -------
  def topic_callback(self, msg):
    ... # operations on msg coming from topic

  # -------   UTILITY FUNCTIONS   -------
  def fun_00(self):
    ...

# -------   INITIALIZE INSTANCE  -------
if __name__ == '__main__':
  node = NodeName()
  node.loop()
```

In accordo ad un approccio *top-down* è possibile analizzare le principali entità che compongono il sistema software, implementabile all'interno di un nuovo package ROS creato *ad hoc* per questo semplice robot mobile (`smr_pkg`).

### 🔁 REF_TRANSFORMER NODE

Le velocità desiderate di riferemento $[v_k, \omega_k]$ possono essere ricevute a seguito del calcolo da parte di un apposito algoritmo di *path-following* o da parte di un modulo di teleoperazione, che legge comandi da un joystik o da una tastiera. Questi riferimenti di velocità vengono pubblicati da appositi nodi sul topic `cmd_vel` attraverso messaggi di tipo `Twist` del package `geometry_msgs`, contenente i seguenti dati:

```
Vector3  linear
Vector3  angular
```

dove `Vector3` è un altro tipo di messaggio appartenente al package `geometry_msgs` contenente a sua volta tre valori di tipo `float`

```
float64 x
float64 y
float64 z
```

Per un differential-drive sono ammessi valori di velocità lineare ($v_k$) solo lungo l'asse *x* e valori di velocità angolare ($\omega_k$) intorno a l'asse *z*, dunque la mappatura nel messaggio di tipo `Twist` sarà la seguente:

> twist.linear.x = $v_k$
> twist.angular.z = $\omega_k$

Le velocità di riferimento sono espresse rispetto al baricentro del robot e dovranno essere dunque trasformate nelle corrispettive velocità di rotazione dell due ruote indipendenti, ciò può essere fatto invertendo le relazioni viste nelle sezioni precedenti:

$$\omega_k^{(R)} = \frac{1}{r}\left(v_k + \frac{\ell}{2}\omega_k\right) \qquad \omega_k^{(L)} = \frac{1}{r}\left(v_k - \frac{\ell}{2}\omega_k\right)$$

Effettuare tale trasformazioni è il ruolo prinipale del nodo `ref_transformer`, il quale è sottoscritto al topic `cmd_vel` e pubblica $\left[\omega_k^{(R)}, \omega_k^{(L)}\right]$ sul topic `wheels_vel_target` utilizzando un messaggio personalizzato `Wheels_Vel` che contienei due valori di velocità angolare per le due ruote.

```
# Wheels_Vel.msg
float32 w_R
float32 w_L
```

Un'implementazione triviale di tale nodo è data dal seguente scrpt Python:

```
#! /usr/bin/env python3
import rospy
from smr_pkg.msg import Wheels_Vel
from std_msgs.msg import Float32
from geometry_msgs.msg import Twist

class RefTransformer():
  def __init__(self):
    rospy.init_node('ref_transformer')

    # ------ PARAMETERS ------
    self.l = rospy.get_param('smr/physics/l') # wheels distance
    self.r = rospy.get_param('smr/physics/r') # wheels radius

    # ------ TOPICS ------
    rospy.Subscriber('cmd_vel', Twist, self.transform_callback)
    self.v_target_pub = rospy.Publisher('wheels_vel_target', Wheels_Vel,
queue_size=1)

  def loop(self):
    rospy.spin()

  def transform_callback(self, msg):
    v_k = msg.linear.x
    w_k = msg.angular.z

    w_R = (1/self.r) * (v_k + (self.l/2) * w_k)
```

```
    w_L = (1/self.r) * (v_k - (self.l/2) * w_k)

    msg_vel = Wheels_Vel()
    msg_vel.w_R = w_R
    msg_vel.w_L = w_L
    self.v_target_pub.publish(msg_vel)

if __name__ == '__main__':
  rt = RefTransformer()
  rt.loop()
```

### 🚂 MOTOR_CONTROLLER NODE

Il nodo `motor_controller`legge i riferimenti di velocità per le due ruote dal topic precedente e, contemporaneamente, sul topic `wheels_vel_sensor` riceve le informazioni della velocità corrente delle due ruote, provenienti dagli appositi nodi sensori. Con queste informazioni è possibile calcolare i segnali di controllo da applicare ai motori delle due ruote attraverso un opportuno algoritmo di controllo. Il ciclo di vita del nodo verrà eseguito con un tempo di campionamento `Ts` e implmenterà una classica azione di controllo automatica come ad esempio un PID. L'implementazione seguente suppone l'implementazione del software sua *Raspberry* e l'utilizzo della libreria `gpiozero` per controllare due motori DC a 12V collegati attraverso appositi driver *H-bridge*.

```
#! /usr/bin/env python3
import rospy
from smr_pkg.msg import Wheels_Vel
from smr_pkg.lib.PID_Sat_controller import PID_Saturation
import RPi.GPIO as GPIO
from gpiozero import Motor

class MotorController():
  def __init__(self):
    rospy.init_node('motor_controller')

    # ------ PARAMETERS ------
    self.Ts = rospy.get_param('/smr/control/Ts', 0.01)
    self.rate = rospy.Rate(1/self.Ts)

    Kp = rospy.get_param('/smr/control/Kp')
    Ki = rospy.get_param('/smr/control/Ki')
    Kd = rospy.get_param('/smr/control/Kd', 0)
    u_max = rospy.get_param('/smr/control/u_max', 12)

    self.N = rospy.get_param('/smr/sensor/buffer_len', 10)

    # Motors pin using Raspberry
    GPIO.setmode(GPIO.BCM)
    pin_forward_R = rospy.get_param('/smr/motor/FPR')
    pin_backward_R = rospy.get_param('/smr/motor/BPR')
    pin_forward_L = rospy.get_param('/smr/motor/FPL')
    pin_backward_L = rospy.get_param('/smr/motor/BPL')
```

```
    # ------ VARIABLES ------
    self.pid_R = PID_Saturation(Kp,Ki,Kd,u_max)
    self.pid_L = PID_Saturation(Kp,Ki,Kd,u_max)

    self.motor_R = Motor(self.pin_forward_R, self.pin_backward_R, True)
    self.motor_L = Motor(self.pin_forward_L, self.pin_backward_L, True)

    self.buffer_R = []
    self.buffer_L = []
    self.w_R_meas = 0
    self.w_L_meas = 0
    self.w_R_ref = 0
    self.w_L_ref = 0

        # ------ TOPICS ------
    rospy.Subscriber('wheels_vel_target', Wheels_Vel,
self.reference_callback)
    rospy.Subscriber('wheels_vel_sensor', Wheels_Vel,
self.measurement_callback)

  def loop(self):
    while not rospy.is_shutdown():
      u_R = self.pid_R.compute(self.w_R_meas, self.w_R_ref, self.Ts)
      u_L = self.pid_L.compute(self.w_L_meas, self.w_L_ref, self.Ts)

      self.__actuate_motors(u_R,u_L)

      self.rate.sleep()

  def reference_callback(self, msg):
    self.w_R_ref = msg.w_R
    self.w_L_ref = msg.w_L

  def measurement_callback(self, msg):
    if len(self.buffer_R) = self.N:
      self.buffer_R.pop(0)
    self.buffer_R.append(msg.w_R)

    if len(self.buffer_L) = self.N:
      self.buffer_L.pop(0)
    self.buffer_L.append(msg.w_L)

    self.w_R_meas = sum(self.buffer_R)/len(self.buffer_R)
    self.w_L_meas = sum(self.buffer_L)/len(self.buffer_L)

  def __actuate_motors(self, u_R, u_L):
    if u_R >= 0:
      self.motor_R.forward(u_R)
    else:
      self.motor_R.backward(abs(u_R))
    if u_L >= 0:
      self.motor_L.forward(u_L)
    else:
```

```
        self.motor_L.backward(abs(u_L))

if __name__ == '__main__':
  mc = MotorController()
  mc.loop()
```

### 📌 WHEELS_VEL_SENSOR NODE

Il nodo `wheels_vel_sensor` ha lo scopo di inteffacciarsi con gli encoder rotativi posti sulle due ruote e comunicare una stima delle velocità istantanee ai nodi interessati. L'implementazione di tale nodo può prevedere due possibili strategie differenti: la lettura dei dati di continuo e la conseguente pubblicazione delle informazioni a una determinata frequenza (attraverso l'uso di un topic); o l'invio della velocità istantanea su richiesta da parte di un nodo implementato, eseguendo, ad esempio, una lettura a *burst* (strategia implementabile con l'ausilio dei servizi). Di seguito è riportata un'implementazione della prima strategia proposta, nell'ipotesi che i sensori montati sulle ruote siano encoder rotativi a due canali con risoluzione $N$.

```
#! /usr/bin/env python3
import rospy
from smr_pkg.msg import Wheels_Vel
import RPi.GPIO as GPIO
import math
import threading

class WheelsVelSensor():
  def __init__(self):
    rospy.init_node('wheels_vel_sensor')

    # ------ PARAMETERS ------
    self.f = rospy.get_param('/smr/sensor/freq', 20)
    self.rate = rospy.Rate(self.f)

    self.l = rospy.get_param('smr/physics/l', 0.2) # wheels distance
    self.r = rospy.get_param('smr/physics/r', 0.04) # wheels radius
    N = rospy.get_param('smr/sensor/enc_N', 256) # encoder resolution
    self.step = 2*math.pi/N

    # Encoders pin using Raspberry
    GPIO.setmode(GPIO.BCM)
    self.pin_channel_A_R = rospy.get_param('/smr/sensor/CAR')
    self.pin_channel_B_R = rospy.get_param('/smr/sensor/CBR')
    self.pin_channel_A_L = rospy.get_param('/smr/sensor/CAL')
    self.pin_channel_B_L = rospy.get_param('/smr/sensor/CBL')
    GPIO.setup(self.pin_channel_A_R, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)
    GPIO.setup(self.pin_channel_B_R, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)
    GPIO.add_event_detect(self.pin_channel_A_R, GPIO.RISING,
callback=callback_enc_R)
    GPIO.setup(self.pin_channel_A_L, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)
    GPIO.setup(self.pin_channel_B_L, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)
    GPIO.add_event_detect(self.pin_channel_A_L, GPIO.RISING,
```

```
callback=callback_enc_L)

    self.lock = threading.Lock()

    # ------ VARIABLES ------
    self.ticks_R = 0
    self.ticks_L = 0
    self.prev_ticks_R = 0
    self.prev_ticks_L = 0

    # ------ TOPICS ------
    self.v_sensor_pub = rospy.Publisher('wheels_vel_sensor', Wheels_Vel,
queue_size=10)

  def loop(self):
    msg_vel = Wheels_Vel()
    while not rospy.is_shutdown():
      with self.lock:
        msg_vel.w_R = (self.ticks_R-self.prev_ticks_R)*self.step*self.f
        msg_vel.w_L = (self.ticks_L-self.prev_ticks_L)*self.step*self.f
        self.prev_ticks_R = self.ticks_R
        self.prev_ticks_L = self.ticks_L
      self.v_sensor_pub.publish(msg_vel)
      self.rate.sleep()

def callback_enc_R(ch):
  global wvs
  with wvs.lock
    # Rising-edge detected on channel A
    if GPIO.input(wvs.pin_channel_B_R) == 1
      wvs.ticks_R += 1
    else # GPIO.input(wvs.pin_channel_B_R) == 0
      wvs.ticks_R -= 1

def callback_enc_R(ch):
  global wvs
  with wvs.lock
    # Rising-edge detected on channel A
    if GPIO.input(wvs.pin_channel_B_R) == 1
      wvs.ticks_R += 1
    else # GPIO.input(wvs.pin_channel_B_R) == 0
      wvs.ticks_R -= 1

if __name__ == '__main__':
  wvs = WheelsVelSensor()
  wvs.loop()
```

### ODOMETRY NODE

L'*odometria* è la tecnica per stimare la posizione di un veicolo su ruote che si basa su informazioni provenienti da sensori che misurano lo spazio percorso da alcune delle ruote e, qualora sia presente, dall'angolo dello sterzo. L'odometria ha lo scopo di fornire la posa attuale del robot istante dopo istante,

tramite la quale è possibile applicare algoritmi per il controllo automatico di posizione e realizzare dunque la navigazione autonoma. Gli encoder posti sulle ruote permettono di stimare, oltre alla velocità di rotazione, lo spazio percorso dalla singola ruota nel corso del tempo. A partire da tali informazioni o dalle velocità stesse è possibile implementare una procedura di localizzazione in tempo reale che sfrutti tali informazioni per risalire alla posizione attuale del robot (rispetto a un sistema di riferimento fisso), facendo uso delle equazioni della cinematica che descrivono l'evoluzione del veicolo nel tempo. Tale descrizione qualitativa può essere implementata aal'interno del nodo `odometry`, il quale è sottoscritto al topic `wheels_vel_sensor` dal quale attinge le informazioni di velocità delle singole ruote e provvede a implementare le equazioni della cinematica diretta per fornire una stima della posa attuale del differential-drive. Tale posa stimata viene pubblicata all'interno del messaggio `Odometry` del package `nav_msgs`, e sarà utilizzata da un eventuale nodo di pianificazione del moto o da un semplice terminale per la visualizzazione in tempo reale della posizione del veicolo teleoperato.

### 📈 PATH_PLANNER NODE

Il nodo `path_planner` modella essenzialmente un entità che ha lo scopo di generare dei riferimenti di velocità necessari a raggiungere un punto *target* precedentemente ricevuto. Spesso gli algoritmi di *path planning* possono spesso adottare strategie di tipo *divide-et-impera* per costruire traiettorie tra punti intermedi in cui viene discretizzata la traiettoria desiderata. Gli algoritmi di pianificazione del moto possono utilizzare algoritmi più o meno sofisticati (campi di potenziali artificiali, ricerca di cammini minimi su grafi - basati a loro volta su algoritmi esaustivi o euristiche di ricerca - o costruzioni di percorsi basati su grafi di *Voronoi*) per determinare il percorso che il robot deve seguire, a tali percorsi viene associata successivamente una legge oraria da cui viene successivamente derivata la velocità di riferimento. Un algoritmo triviale per il raggiungimento di un goal può essere schematizzata con i seguenti passi:

- ruotare fino ad allineare il roprio orientamento con il punto che si vuole raggiungere;
- muoversi in linea retta a velocità costante fintanto che non si sia raggiunto il target (in genere si seleziona un range di tolleranza dal punto obiettivo);
- ruotare sul posto sino al raggiungimento dell'orientazione di riferimento.

### 🎮 TELEOP NODE

La teleoperazione può essere realizzata attraverso un vasta gamma di differenti dispositivi di controllo remoto: tastiera, joystick, radiocomandi, ecc. Lo scopo del nodo è quello di generare un messaggio di tipo `Twist` da pubblicare sul topic `cmd_vel`, topic sul quale vengono pubblicati i valori di velocità di riferimento per il baricentro del veicolo. ROS fornisce una vasta gamma di soluzioni già implementate ai fini di realizzare tale task, per la teleoperazione da tastiera possono essere utilizzati i nodi `teleop_twist_keyboard` o `turtlebot3_teleop_key`, o, in alternativa, sviluppare un proprio algoritmo di telecontrollo. Un esempio di nodo per il controllo remoto da tastiera è riportato di seguito (è da precisare che l'implementazione di tale nodo è funzionante solo per installazioni di ROS da *superuser*).

```
#!/usr/bin/env python3
import rospy
import time
import keyboard
from geometry_msgs.msg import Twist
```

```
class Teleop:
  def __init__(self):
    rospy.init_node('teleop_node')

    # ------- PARAMETERS -------
    self.f = rospy.get_param('/smr/control/Ts', 0.01)
    self.rate = rospy.Rate(self.f)
    # Velocities bounds
    self.v_max = rospy.get_param('/smr/physics/v_max', 0.2)
    self.v_min = rospy.get_param('/smr/physics/v_min', 0.01)
    self.w_max = rospy.get_param('/smr/physics/w_max', 3)
    self.w_min = rospy.get_param('/smr/physics/w_min', 0.7)
    self.v_step = rospy.get_param('/smr/physics/v_step', 10)

    # ------- VARIABLES -------
    self.cmd = Twist()
    self.cmd.linear.x = 0.0
    self.cmd.angular.z = 0.0

    # -------   TOPICS   -------
    self.cmd_vel_pub = rospy.Publisher('cmd_vel', Twist, queue_size = 1)

  def loop(self):
    while not rospy.is_shutdown():
      # Detect new command
      if keyboard.is_pressed('up'):
        self.cmd.linear.x = min([self.cmd.linear.x + (self.v_max -
self.v_min)/self.v_step, self.v_max])
      elif keyboard.is_pressed('down'):
        self.cmd.linear.x = max([self.cmd.linear.x - (self.v_max -
self.v_min)/self.v_step, -self.v_max])
      else:
        self.cmd.linear.x = 0.0
      if keyboard.is_pressed('left'):
        self.cmd.angular.z = min([self.cmd.angular.z + (self.w_max -
self.w_min)/self.v_step, self.w_max])
      elif keyboard.is_pressed('right'):
        self.cmd.angular.z = max([self.cmd.angular.z - (self.w_max -
self.w_min)/self.v_step, -self.w_max])
      else:
        self.cmd.angular.z = 0.0

      self.cmd_vel_pub.publish(self.cmd)
      self.rate.sleep()

if __name__ == '__main__':
  t = Teleop()
  t.loop()
```

# Modellazione Fisica

Nella sezione corrente verrà descritto come modellare dal punto di vista fisico un robot. In ROS vengono utilizzati appositi file di descrittivi in formato *XML*, per fornire una modellazione delle varie parti che costituiscono il robot; tali file prendono il nome di **Unified Robot Description Format** (**URDF**). In tali file è possibile fornire una descrizione di:

- modelli cinematici e dinamici delle componenti del robot;
- rapresentazione grafica delle componenti;
- modelli di collisione del robot per interazioni con l'ambiente esterno.

I robot vengono modellati attraverso un insieme di elementi detti **link**, ed elementi di tipo **joint** per modellare le connessioni tra i diversi corpi. Un *link* è un corpo rigido, come un telaio o una ruota; gli elementi *joint* connettono 2 link definendo come questi si possono muovere l'uno rispetto a l'altro. Per ogni link vengono specificate proprietà di tipo grafico (attraverso il tag `<visual>`), proprietà di natura geometrica (tag `<geometry>`) come la forma del corpo, o proprietà riguardo al materiale/colore di costituzione del corpo (`<material>`). Oltre a tali informazioni viene definito per ogni link un sistema di riferimento ad esso solidale, la cui origine sarà il punto di riferimento per quel corpo. Per gli elementi di tipo *joint* viene specificato anzitutto il tipo di giunto in accordo alla seguente tabella:

| Tipo | Descrizione |
|---|---|
| `continuous` | Un giunto rotoidale che può ruotare indefinitamente lungo un asse |
| `revolute` | Un giunto rotoidale con limiti fisici di rotazione |
| `prismatic` | Un giunto prismatico che consente la traslazione lungo un asse, specificando i limiti minimo e massimo |
| `planar` | Un giunto planare che consente la traslazione lungo un piano e la rotazione intorno l'asse normale al piano |
| `floating` | Un giunto virtuale che consente tutte e sei le traslazioni e le rotazioni nello spazio |
| `fixed` | Un giunto virtuale che che non consente alcun tipo di movimento tra i corpi |

Oltre al tipo, per ogni giunto vengono specificati i due link che questo interconnette, specificando i nomi dei due frame di riferimento attraverso i tag `<parent>` e `<child>`; infine viene specificato il punto di interconnessione fra i due corpi (dove si trova il giunto), espresso mediante la posizione relativa del frame `child` rispetto al frame `parent`, usata per costruire la matrice di rototraslazione per passare da un frame a l'altro.
In aggiunta ai file URDF è possibile usare file **Xacro** (*XML Macros* language) per creare delle macro XML (l'equivalente delle funzioni in altri linguaggi) per creare dei file XML più corti e migliorarne la leggibilità. Tramite `xacro` è possibile automatizzare la creazione di blocchi di codice associati alla modellazione di elementi fisici uguali. A titolo d'esempio verrà modellato un robot differential-drive, supponendo che questo sia caratterizzato dalle seguenti proprietà fisiche: il telaio è una scatola squadrata di dimensioni (0.3x0.2x0.03m), le ruote attuate hanno un raggio di 3cm e sono larghe 2cm, la ruota d'ausilio è una ruota sferica (*caster wheel*) anch'essa di raggio 3cm, posizionata per garantire la stabilità del telaio. Il file di modellazione è `smr.urdf.xacro` locato nella cartella `description/urdf/` del package `smr_pkg`.

```
<?xml version="1.0"?>
<robot name="smr">

  <link name="base_footprint"/>

  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.3 0.2 0.03"/>
      </geometry>
      <material name="avana brown">
        <color rgba="0.6314 0.4275 0.2157 1"/>
      </material>
    </visual>
  </link>

  <joint name="base_joint" type="fixed">
    <parent link="base_footprint"/>
    <child link="base_link" />
    <origin xyz="0.05 0 0.03" rpy="0 0 0"/>
  </joint>

  <link name="right_wheel">
    <visual>
      <geometry>
        <cylinder length="0.02" radius="0.03"/>
      </geometry>
      <material name="black">
    <color rgba="0 0 0 1"/>
      </material>
    </visual>
  </link>

  <joint name="right_wheel_joint" type="continuous">
    <axis xyz="0 0 1"/>
    <parent link="base_link"/>
    <child link="right_wheel"/>
    <origin rpy="-1.5708 0 0" xyz="0.05 -0.11 0"/>
  </joint>

  <link name="left_wheel">
    <visual>
      <geometry>
        <cylinder length="0.02" radius="0.03"/>
      </geometry>
      <material name="black">
    <color rgba="0 0 0 1"/>
      </material>
    </visual>
  </link>

  <joint name="left_wheel_joint" type="continuous">
```

```
    <axis xyz="0 0 1"/>
    <parent link="base_link"/>
    <child link="left_wheel"/>
    <origin rpy="-1.5708 0 0" xyz="0.05 0.11 0"/>
  </joint>

  <link name="caster_wheel">
    <visual>
      <geometry>
        <sphere radius="0.015"/>
      </geometry>
      <material name="black">
    <color rgba="0 0 0 1"/>
      </material>
    </visual>
  </link>

  <joint name="caster_wheel_joint" type="fixed">
    <axis xyz="0 1 0"/>
    <parent link="base_link"/>
    <child link="caster_wheel"/>
    <origin rpy="0 0 0" xyz="-0.1 0 -0.015"/>
  </joint>

</robot>
```

È inoltre possibile includere forme più complesse per modellare le geometrie dei corpi, includendo modelli *mesh* ad alta risoluzione (usando ad esempio file `*.stl` ricavati da un modello CAD 3D); attraverso il tag `<mesh>` è possibile sostituire le forme base per la geometria del link, specificando il nome del file contenente la *mesh*.
Per visualizzare il risultato del modello assemblato (anche man mano che lo si costruisce), risulta molto utile fare ricorso allo strumento di visualizzazione messo a disposizione da ROS: **Rviz**. Per eseguire `rviz` e caricare il modello URDF del robot è possibile configurare un file `*.launch` nel modo seguente:

```
<launch>
  <arg name="model" default="$(find
smr_pkg)/description/urdf/smr.urdf.xacro"/>

  <param name="robot_description" command="$(find xacro)/xacro.py $(arg
model)" />
  <param name="use_gui" value="true"/>

  <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
  <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
urdf_tutorial)/rviz/urdf.rviz" required="true" />
</launch>
```

La prima riga

```
<arg name="model" default="$(find
smr_pkg)/description/urdf/smr.urdf.xacro"/>
```

specifica il path del file URDF e lo assegna come *argomento* alla variabile `model`, per richiamarlo nella riga seguente.

```
<param name="robot_description" command="$(find xacro)/xacro.py $(arg
model)" />
```

Per convenzione il modello URDF del robot viene caricato all'interno del *parameter server* con il nome standard `robot_description`. Tale operazione viene eseguita dall'interprete `xacro` che riceve come parametro il file URDF che deve interpretare. Dopo aver caricato il modello è necessario eseguire alcuni nodi per consentire la corretta visualizzazione e movimentazione dei corpi che costituiscono il robot.

```
<node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />
```

Il nodo `joint_state_publisher` dell'omonimo package pubblica in real-time la posizione si ogni giunto che compone il sisema, utilizzando messaggi di tipo `sensor_msgs/JointState` pubblicati sul topic `joint_states`.

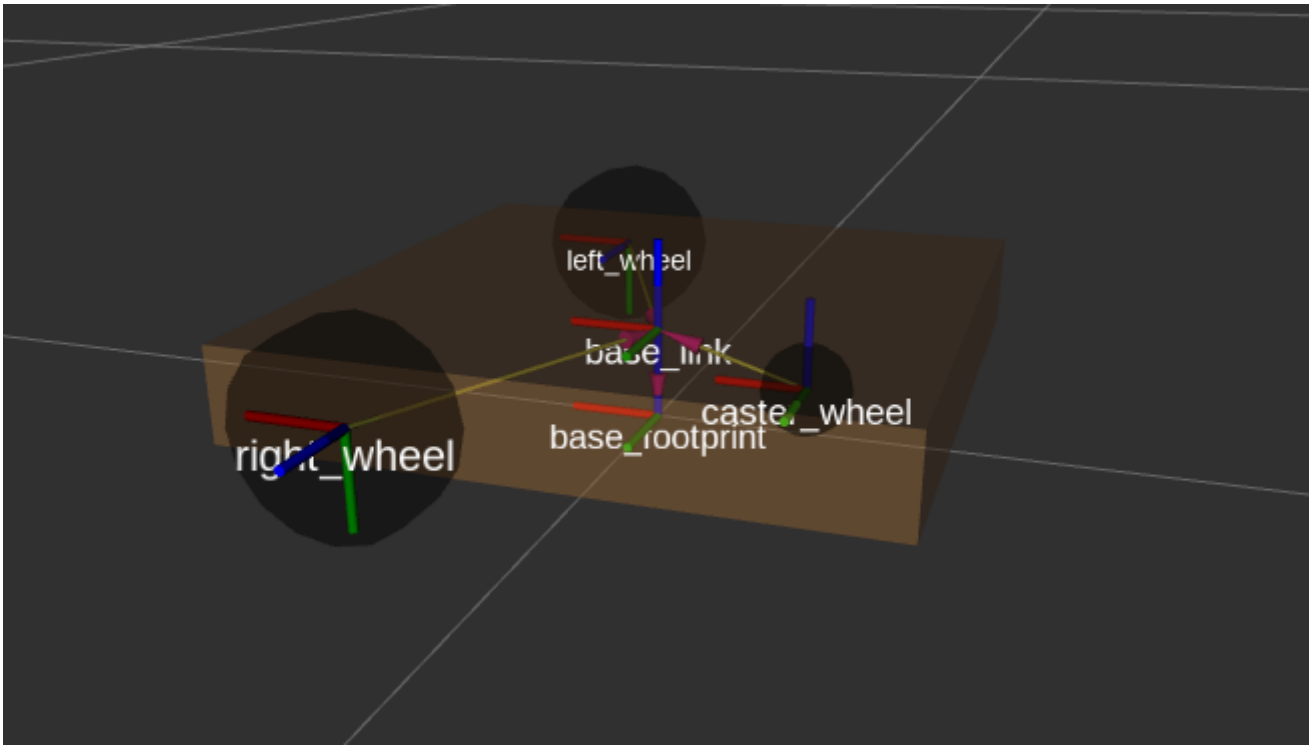
```
<node name="robot_state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
```

Un altro nodo, il `robot_state_publisher`, legge ed interpreta il file URDF dal parameter server ed è sottoscritto al topic `/joint_states`; il suo ruolo è quello di combinare le posizioni 1D di ogni giunto per calcolare l'albero delle trasformaioni che consentono di risalire alla posa nello spazio (6D) di ogni link rispetto agli altri: in altre parole, si occupa di risolvere la *cinematica diretta*. Questo albero di trasformazioni viene pubblicato attraverso messsaggi `tf2_msgs/TFMessage` sul topic `/tf` .

```
<node name="rviz" pkg="rviz" type="rviz" args="-d $(find
urdf_tutorial)/rviz/urdf.rviz" required="true" />
```

Infine viene eseguito `rviz` che legge il modello URDF ed è sottoscritto al topic `/tf`, consentendo così la corretta visualizzazione della posa di ogni singolo link del robot.
Una volta creato, salvare il file `*.launch` nella cartella `launch/` del package ed eseguire `rviz` digitando in un terminale:

```
$ roslaunch smr_pkg display_rviz.launch
```




Il modello URDF racchiude le informazioni cinematiche e grafiche del differential-drive, ma non ha conoscenza delle caratteristiche necessarie alla simulazione dinamica del veicolo. Affinché sia possibile effettuare dei test nell'ambiente simulativo **Gazebo**, è necessario introdurre nel file `*.urdf` due ulteriori tag per ciascun link del modello:

- `<collision>`: simile a `<visual>`, questo tag definisce la forma e le dimensioni di ciascun corpo, per determinare come questo interagisce con gli altri oggetti resenti nell'ambiente simulativo. La eometria di collisione spesso coincide con la geometria grafica, ma, in caso di geometrie mesh, è possibile utilizzare modelli a risoluzione minore o incapsulare la geometria all'interno di forme più semplici (scatole, cilindri, sfere) per diminuire l'onere computazionale della simulazione.
- `<inertial>`: questo tag definisce la massa e i momenti di inerzia di ciascun link, necessari a simulare correttamente i movimenti dei corpi in accordo alle principali leggi della meccanica classica. Ricavare tali dati per forme non elementari può diventare difficoltoso, spesso si può ricorrere a software specifici (ad esempio *MeshLab*) per ricavare una stima di tali parametri a partire dai file `*.stl`.

Nell'esempio corrente di modellazione di un semplice differential-drive è possibile introdurre tali informazioni dinamiche ricorrendo alle stesse geometrie usate per la visualizzazione grafica, e, essendo geometrie elementari, utilizzare le ben note formule elementari per calcolare le matrici di inerzia di ciascun link. Si assuma che il peso del telaio del robot abbia una massa di 1Kg, e ciascuna ruota (compresa la ruota sferica) di 0.1Kg. Utilizzando le formule per tali geometrie di base (cuboidi, cilindri e sfere) si ricavano i momenti di inerzia degli assi principali:

$$\begin{cases} I_{xx} = \frac{1}{12}m(dy^2 + dz^2) \\ I_{yy} = \frac{1}{12}m(dx^2 + dz^2) \\ I_{zz} = \frac{1}{12}m(dx^2 + dy^2) \end{cases} \quad \begin{cases} I_{xx} = \frac{1}{12}m(3r^2 + h^2) \\ I_{yy} = \frac{1}{12}m(3r^2 + h^2) \\ I_{zz} = \frac{1}{12}mr^2 \end{cases} \quad \begin{cases} I_{xx} = \frac{2}{5}mr^2 \\ I_{yy} = \frac{2}{5}mr^2 \\ I_{zz} = \frac{2}{5}mr^2 \end{cases}$$

Il file `smr.urdf.xacro` diventa

```
<?xml version="1.0"?>
<robot name="smr">

  <link name="base_footprint"/>

  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.3 0.2 0.03"/>
      </geometry>
      <material name="avana brown">
    <color rgba="0.6314 0.4275 0.2157 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <box size="0.3 0.2 0.03"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="1.0"/>
      <inertia ixx="3.408e-3"
               ixy="0"       iyy="7.575e-3"
               ixz="0"       iyz="0"       izz="1.083e-2"/>
    </inertial>
  </link>

  <joint name="base_joint" type="fixed">
    <parent link="base_footprint"/>
    <child link="base_link" />
    <origin xyz="0 0 0.03" rpy="0 0 0"/>
  </joint>

  <link name="right_wheel">
    <visual>
      <geometry>
        <cylinder length="0.02" radius="0.03"/>
      </geometry>
      <material name="black">
    <color rgba="0 0 0 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <cylinder length="0.02" radius="0.03"/>
```

```
      </geometry>
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="2.583e-5"
               ixy="0"       iyy="2.583e-5"
               ixz="0"       iyz="0"       izz="7.5e-6"/>
    </inertial>
  </link>

  <joint name="right_wheel_joint" type="continuous">
    <axis xyz="0 0 1"/>
    <parent link="base_link"/>
    <child link="right_wheel"/>
    <origin rpy="-1.5708 0 0" xyz="0.05 -0.11 0"/>
  </joint>

  <link name="left_wheel">
    <visual>
      <geometry>
        <cylinder length="0.02" radius="0.03"/>
      </geometry>
      <material name="black">
    <color rgba="0 0 0 1"/>
      </material>
    </visual>
    <collision>
      <geometry>
        <cylinder length="0.02" radius="0.03"/>
      </geometry>
    </collision>
    <inertial>
      <mass value="0.1"/>
      <inertia ixx="2.583e-5"
               ixy="0"       iyy="2.583e-5"
               ixz="0"       iyz="0"       izz="7.5e-6"/>
    </inertial>
  </link>

  <joint name="left_wheel_joint" type="continuous">
    <axis xyz="0 0 1"/>
    <parent link="base_link"/>
    <child link="left_wheel"/>
    <origin rpy="-1.5708 0 0" xyz="0.05 0.11 0"/>
  </joint>

  <link name="caster_wheel">
    <visual>
      <geometry>
        <sphere radius="0.015"/>
      </geometry>
      <material name="black">
    <color rgba="0 0 0 1"/>
      </material>
```

```
      </visual>
      <collision>
        <geometry>
          <sphere radius="0.015"/>
        </geometry>
      </collision>
      <inertial>
        <mass value="0.1"/>
        <inertia ixx="9e-6"
                 ixy="0"   iyy="9e-6"
                 ixz="0"   iyz="0"   izz="9e-6"/>
      </inertial>
    </link>

    <joint name="caster_wheel_joint" type="fixed">
      <axis xyz="0 1 0"/>
      <parent link="base_link"/>
      <child link="caster_wheel"/>
      <origin rpy="0 0 0" xyz="-0.1 0 -0.015"/>
    </joint>

  </robot>
```

È possibile inoltre definire per ciascun link ulteriori parametri intrinseci del corpo:

| ***Name*** | ***Type*** | ***Description*** |
|---|---|---|
| `material` | value | Material of visual element |
| `gravity` | bool | Use gravity |
| `dampingFactor` | double | Exponential velocity decay of the link velocity - takes the value and multiplies the previous link velocity by (1-dampingFactor) |
| `maxVel` | double | maximum contact correction velocity truncation term |
| `minDepth` | double | minimum allowable depth before contact correction impulse is applied |
| `mu1`, `mu2` | double | Friction coefficients μ for the principal contact directions along the contact surface as defined by the Open Dynamics Engine (ODE) (see parameter descriptions in ODE's user guide) |
| `fdir1` | string | 3-tuple specifying direction of mu1 in the collision local reference frame |
| `kp`, `kd` | double | Contact stiffness k_p and damping k_d for rigid body contacts as defined by ODE (ODE uses erp and cfm but there is a mapping between erp/cfm and stiffness/damping) |
| `selfCollide` | bool | If true, the link can collide with other links in the model |
| `maxContacts` | int | Maximum number of contacts allowed between two entities. This value overrides the max_contacts element defined in physics |
| `laserRetro` | double | intensity value returned by laser sensor |

## Simulazione

per simulare il robot è necessario introdurre un meccanismo di attuazione dei vari giunti e ricevere le informazioni di posizioni da ciascuno di essi per poter calcolare la posizione corrente del robot. È necessario dunque che il modello sia correlato di un'architettura software che esporti un'interfaccia di topic del tipo `cmd_vel/odom`. Su un robot reale tale interfaccia viene realizzata da una serie di nodi che si interfacciano direttamente con l'hardware del veicolo fungendo da driver; come nell'esempio dell'architettura proposta, una serie di nodi implementati su Raspberry sarebbero in grado di esportare tale interfaccia. In ambiente simulativo vengono usati dei **Gazebo plugins** per simulare sensori, attuatori o addirittura robot completi (i plugin sono librerie precompilate, alcune di esse possono essere ritrovate nella cartella di installazione di ROS: `/opt/ros/melodic/lib`). Tali plugin simulano attuatori e sensori e l'interfacciamento con essi, fungendo da driver che esportano direttamente i topic ROS necessari al loro utilizzo. Il *differential-drive plugin* esporta una interfaccia `cmd_vel/odom` che consnte il controllo di un differential-drive mediante messaggi di tipo `Twist` sul topic `/cmd_vel`, trasformando i comandi di velocità del robot in velocità di attuazione per le due ruote; al contempo esporta le informazioni sulla posa corrente del robot sul topic `/odom`.
Per aggiungere un plugin è necessario aggiungere alcuni tag nel file `smr.urdf.xacro`

```
<gazebo>
  <plugin name="differential_drive_controller"
filename="libgazebo_ros_diff_drive.so">
    <commandTopic>cmd_vel</commandTopic>
    <odometryTopic>odom</odometryTopic>
    <leftJoint>left_wheel_joint</leftJoint>
    <rightJoint>right_wheel_joint</rightJoint>
    <robotBaseFrame>base_link</robotBaseFrame>
    <wheelSeparation>0.2</wheelSeparation>
    <wheelDiameter>0.06</wheelDiameter>
    <wheelTorque>10</wheelTorque>
    <publishWheelJointState>true</publishWheelJointState>
  </plugin>
</gazebo>
```

Per prima cosa è necessario aprire un tag `<gazebo>` all'interno del tag `<robot>` del modello, successivamente

```
<plugin name="differential_drive_controller"
filename="libgazebo_ros_diff_drive.so">
```

viene caricato il plugin associato alla libreria `libgazebo_ros_diff_drive.so` per la simulazione dell'intero robot. Tale libreria richiede la specifica di alcuni campi per la creazione del robot virtuale, tra cui

```
<commandTopic>cmd_vel</commandTopic>
<odometryTopic>odom</odometryTopic>
```

per specificare i nodi dei topic sul quale ricevere i comandi di velocità ed esportare l'odometria del robot. I frame di riferimento per il centro di massa del robot e le due ruote

```
<leftJoint>left_wheel_joint</leftJoint>
<rightJoint>right_wheel_joint</rightJoint>
<robotBaseFrame>base_link</robotBaseFrame>
```

e i parametri fisici necessari ai calcoli cinematici

```
<wheelSeparation>0.2</wheelSeparation>
<wheelDiameter>0.06</wheelDiameter>
<wheelTorque>10</wheelTorque>
```

Infine, tramite

```
<publishWheelJointState>true</publishWheelJointState>
```

si abilita il plugin a publicare i messaggi sul topic `joint_states/`, per la posizione corrente delle ruote. Per caricare il modello del robot in Gazebo si possono seguire differenti strade, di seguito si riporta l'avvio della simulazione attraverso file `*.launch`, il quale avrà il compito si caricare il modello URDF del robot sul parameter server, eseguire Gazebo in un mondo virtuale di simulazione (inizialmente un *empty world*), infine richiamare un servizio per deporre (*spawn*) un'istanza del robot in Gazebo interpretando il file di descrizione URDF.

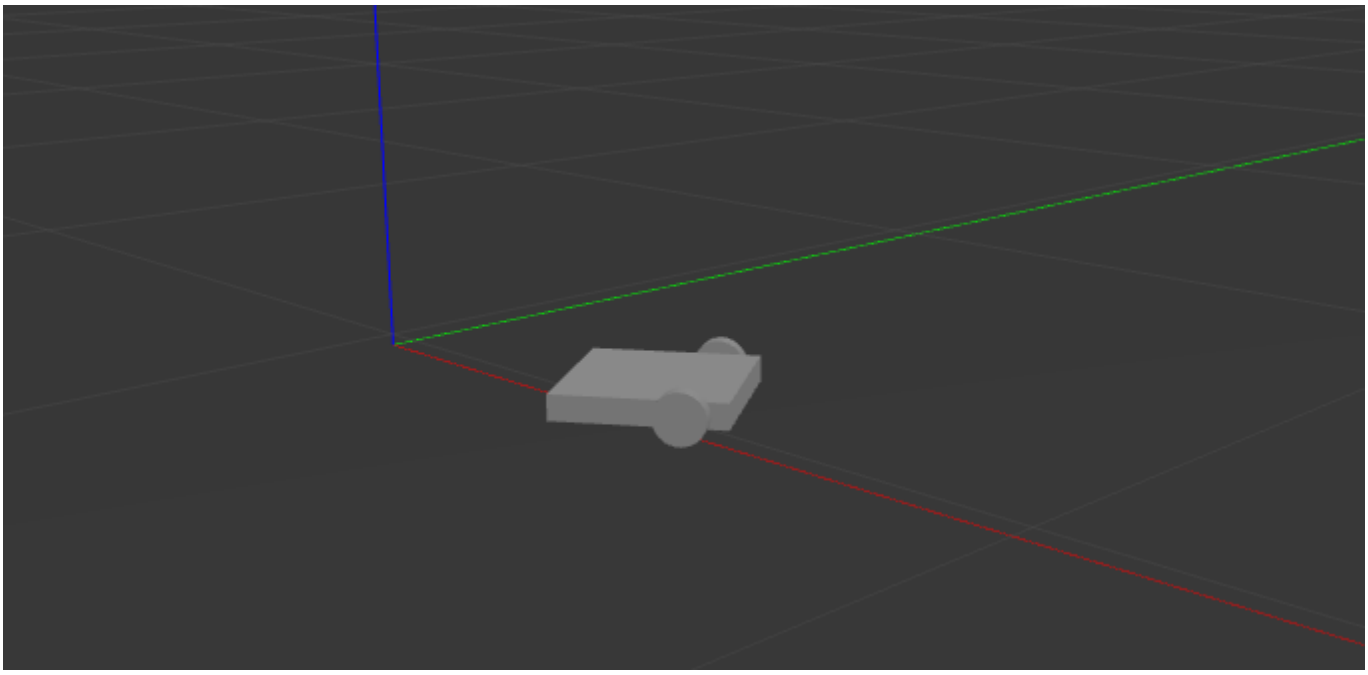
```
<launch>
  <!-- Load the URDF model into the parameter server -->
  <param name="robot_description" textfile="$(find
smr_pkg)/description/urdf/smr.urdf.xacro" />

  <!-- Start Gazebo with an empty world -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch"/>

  <!-- Spawn a smr in Gazebo, taking the description from the
  parameter server -->
  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
  args="-param robot_description -urdf -model smr" />
</launch>
```

Salvare il file `gazebo_sim.launch` nella cartella `launch/` ed eseguire in un terminale

```
$ roslaunch smr_pkg gazebo_sim.launch
```

A questo punto l'ambiente di simulazione Gazebo verrà aperto e un'istanza del robot sarà posizionata nell'origine del mondo virtuale.
Il modo più semplice per verificare che il robot funzioni correttamente con il plugin `diff_drive` è eseguire il nodo `teleop_node` e provare a teleoperare il robot nel mondo virtuale:

```
$ roslaunch turtlebot3_teleop turtlebot3_teleop.launch
```



> **GAZEBO PROBLEM**
>
> Potrebbe capitare che all'avvio di Gazebo venga sollevato l'errore:
>
> > [Err] [REST.cc:205] Error in REST request libcurl: (51) SSL: no alternative certificate subject name matches target host name 'api.ignitionfuel.org'
>
> Tale errore è dovuto all'aggiornamento del repository a cui Gazebo scarica i propri modelli. Per risolvere il problema basta modificare il file `~/.ignition/fuel/config.yaml` sostituendo la stringa `api.ignitionfuel.org` con il nuovo URL `api.ignitionrobotics.org`.

A un primo sguardo Gazebo e Rviz possono sembrare molto simili: entrambi mettono a disposizione una visualizzazione 3D del robot e permettono di visualizzare il robot nel suo ambiente virtuale. Tuttavia i due software giocano ruoli totalmente diversi: Gazebo simula effettivamente il robot all'interno di un mondo virtuale, mentre Rviz permette di visualizzare il robot e le sue conoscenze sull'ambiente circostante (processamento delle informazioni provenienti dai sensori). Gazebo è dunque un sostituto del robot reale in un ambiente naturale, ed esegue la simulazione dell'evoluzione del sistema implementando tutte le leggi della fisica e gli effetti delle forze che si generano dall'interazione del robot con gli altri elementi virtuali e con l'ambiente stesso, inoltre sono presenti plugin che permettono la generazione di dati sintetici a partire da modelli di sensori virtuali. Alla luce di ciò, Gazebo implementa diverse tipologie di plugin per simulare diversi aspetti del mondo reale: vi sono plugin che aggiungono al mondo virtuale specifiche caratteristiche fisiche (aereodinamica, idrodinamica, ecc), altri che modellano e permettono di controllare oggetti presenti nell'ambiente simulativo (robot o altri oggetti), altri, ancora, che permettono di simulare l'acquisizione di

dati da parte di sensori, implementando gli effetti fisici su cui sono basati gli elementi sensibili (camere, laser, GPS, IMU, ecc).
Dall'altro lato, Rviz permette di visualizzare lo *stato* del robot e le sue conoscenze dell'ambiente circostante, e che sia il mondo reale, e che sia l'ambiente simulativo. Le informazioni provenienti dai sensori a bordo costituiscono le *conoscenze* del robot, queste sono elaborate opportunamente e rappresentano cosa il robot *pensa* del mondo circostante (a paritre dalle informazioni di odometria e da quelle ottenute a partire da un sensore laser, ad esempio, il robot è in grado di costruire una mappa dell'ambiente e sapere dove si trova in ogni momento).
Dunque Rviz può essere usato in sinergia con Gazebo per visualizzare le percezioni del robot nell'ambiente simulato, in vista di essere trasferito, successivamente nel mondo reale.
Per visualizzare correttamente il robot, Rviz richiede di conoscere la struttura dell'intero robot e il suo stato corrente, in termini delle posizioni di tutte le sue variabili di giunto. È necessario dunque pubblicare tali informazioni sul topic `/joint_states` utilizzando il plugin `joint states publisher plugin` per publicare lo stato dei giunti non pubblicati gia da altri plugin (nel caso del differential-drive, il `diff-drive` pluin non pubblica le informazioni di giunto della ruota ausiliaria).

```
<plugin name="joint_state_publisher"
filename="libgazebo_ros_joint_state_publisher.so">
<jointName>caster_wheel_joint</jointName>
</plugin>
```

È possibile inoltre aggiungere l'esecuzione contestuale del nodo `robot_state_publisher` aggiungendo nel file `*.launch` le seguenti linee di codice:

```
  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher"/>
```

A questo punto è possibile eseguire l'ambiente di lavoro virtuale messo a disposizione da Gazebo:

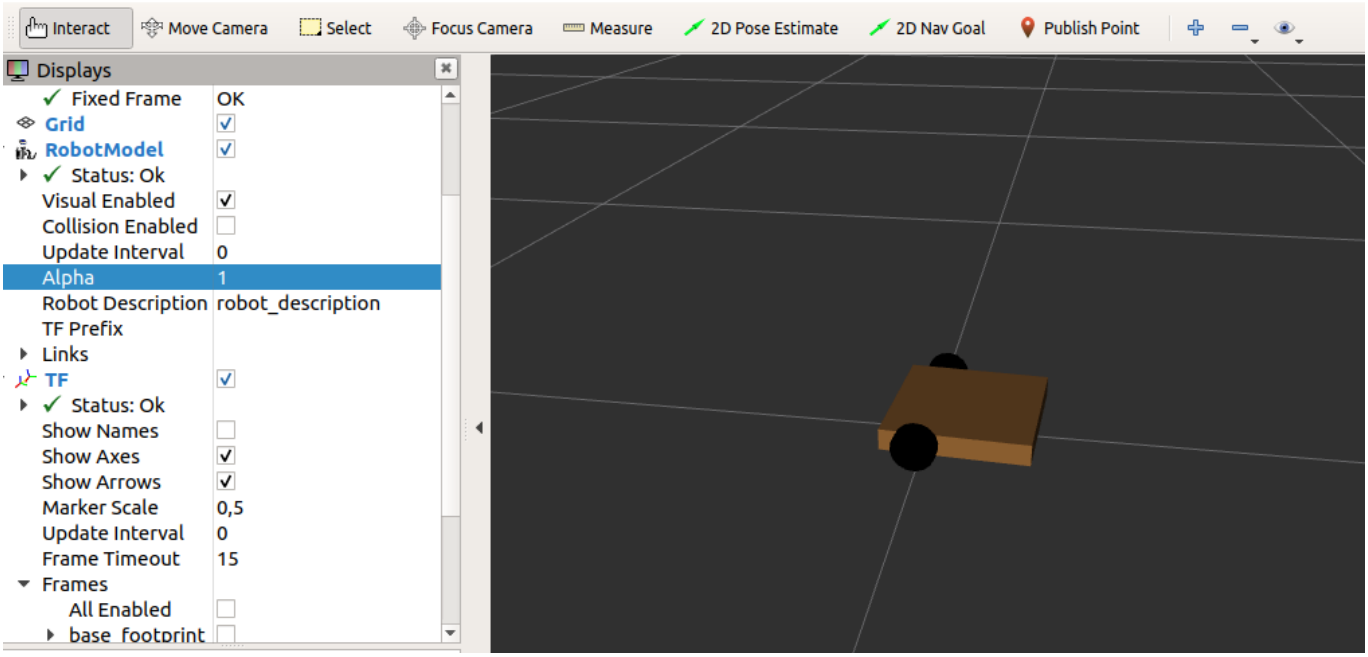
```
$ gazebo_sim.launch
```

e, in un nuovo terminale, lanciare Rviz:

```
$ rviz
```

Una volta lanciato Rviz, sarà necessario modificare alcune impostazioni di visualizzazione nel pannello laterale:

- In `Displays` → `Global Options`, impostare `fixed_frame` su `odom`; questo permette di visualizzare il robt muoversi intorno al punto iniziale (origine dell'origine dell'odometria).
- In `Displays`, utilizzare il bottone `Add` per inserire un nuovo `Robot Model`; questo consente a Rviz di leggere il file `*.urdf.xacro` dal parameter server e visualizzarlo in 3D.




È possibile verificare che tutto stia funzionando correttamente andando a utilizzare i comandi base di ROS, come `rostopic echo` seguito dal nome del topic`/tf`, per assicurarsi che tutte le trasformazioni tra i corpi siano pubblicate correttamente, o `/joints_states` per vedere visualizzate le informazioni di giunto.

## Mondo Virtuale Personalizzato

Gazebo permette di modificare e personalizzare il mondo virtuale entro cui il robot si potrà muovere. Oltre a vere e proprie ambientazioni del mondo reale, è possibile personalizzare il mondo base inserendo forme di base (quale cubi, cilindri, sfere, ecc) o oggetti più sofisticati già presenti nelle librerie di sistema (i modelli sono salvati su un server remoto, è necessario, al primo utilizzo, scaricare in locale i modelli che si intendono utlizzare).
I modelli degli oggetti possono essere trascinati nello spazio vuoto e il mondo può essere salvato in un file`*.world` nella cartella `worlds/` del package. Per caricare un mondo personalizzato precedentemente salvato è necessario specificare come argomento `world_name`, nel file `*.launch`, il percorso e il file `*.world`

```
<launch>
  <!-- Load the URDF model into the parameter server -->
  <param name="robot_description" command="$(find xacro)/xacro $(find
smr_pkg)/description/urdf/smr.urdf.xacro" />

  <!-- Start Gazebo with a custom world -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find smr_pkg)/worlds/test.world"/>
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="debug" value="false"/>
  </include>

  <!-- Spawn a smr in Gazebo, taking the description from the parameter
server -->
```

```
  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param
robot_description -urdf -model smr" />

  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher"/>

</launch>
```

## Sensori Virtuali

Come già accennato è possibile aggiungere dei sensori virtuali al robot attraverso l'uso dei plugin di Gazebo. A titolo d'esempio è possibile provare ad equipaggiare il differential-drive con un **laser range-finders**. Sensori laser di tale famiglia consentono di registrare accurate informazioni sull'ambiente circostante, sfruttando il tempo di volo di un fascio luminoso per calcolare la distanza dagli oggetti circostanti.
Per includere il sensore nel modello del robot è necessario aggiungere un link URDF che rappresenterà il sensore fisico, questo sarà collegato al resto della struttura per mezzo di un giunto.Si supponga di modellare il sensore laser con un cilindro montato sulla parte anteriore del telaio:

```
<link name="laser_link">
  <visual>
    <geometry>
      <cylinder length="0.04" radius="0.05"/>
    </geometry>
    <material name="gray">
      <color rgba="0.1 0.1 0.2 1"/>
    </material>
  </visual>
  <collision>
    <geometry>
      <cylinder length="0.04" radius="0.05"/>
    </geometry>
  </collision>
  <inertial>
    <mass value="0.4"/>
    <inertia ixx="3.03e-4"
             ixy="0"       iyy="3.03e-4"
             ixz="0"       iyz="0"       izz="8.3e-5"/>
  </inertial>
</link>

<joint name="laser_joint" type="fixed">
  <axis xyz="0 0 0"/>
  <parent link="base_link"/>
  <child link="laser_link"/>
  <origin rpy="0 0 0" xyz="0.05 0 0.035"/>
</joint>
```

Successivamente è necessario specificare a Gazebo che a tale link è associato un sensore; ciò viene fatto attraverso il tag `<sensor>` per specificare il tipo di sensore associato e i parametri intrinseci del sensore:

```
<gazebo reference="laser_link">
  <sensor type="ray" name="laser_sensor">
    <pose>0 0 0 0 0 0</pose>
    <visualize>false</visualize>
    <update_rate>40</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>0.015</resolution>
      </range>
      <noise>
        <type>gaussian</type>
        <mean>0.0</mean>
        <stddev>0.01</stddev>
      </noise>
    </ray>
    <plugin name="laser_sensor" filename="libgazebo_ros_laser.so">
      <topicName>scan</topicName>
      <frameName>laser_link</frameName>
    </plugin>
  </sensor>
</gazebo>
```

I punti chiave sono riassunti come segue:

- Creazione di un sensore di tipo `ray` ad agganciarlo al `laser_link`.
- Configurare i parametri base del sensore in accordo alle specifiche di accuratezza del sensore reale che si intende simulare; nell'esempio la frequenza di acquisizione dati è pari a 40 Hz, con uno spettro di 720 campioni su un campo di visione di 180 gradi, risoluzione di 1 un grado e accuratezza di 0.1m fino a 10m di distanza massima. È inoltre possibile aggiungere un tag `<noise>` con le opportune proprietà di disturbo, in modo da simulare un sensore reale.
- Infine, agganciare il plugin implementato nella libreria `libgazebo_ros_laser.so`, il quale pubblica le informazioni acquisite dal sensore virtuale attraverso messaggi di tipo `sensor_msgs/LaserScan` sul topic `/scan`.
  Salvando tale modello nel file `smr_equip.urdf.xacro` è possibile lanciare una nuova simulazione all'interno di un ambiente virtule personalizzato in cui è presente un cilindro difronte al robot.

```
<launch>
  <!-- Load the URDF model into the parameter server -->
  <param name="robot_description" command="$(find xacro)/xacro $(find
smr_pkg)/description/urdf/smr_equip.urdf.xacro" />

  <!-- Start Gazebo with a custom world -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" value="$(find smr_pkg)/worlds/test.world"/>
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="true"/>
    <arg name="gui" value="true"/>
    <arg name="headless" value="false"/>
    <arg name="debug" value="false"/>
  </include>

  <!-- Spawn a smr in Gazebo, taking the description from the parameter
server -->
  <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param
robot_description -urdf -model smr" />

  <node name="robot_state_publisher" pkg="robot_state_publisher"
type="robot_state_publisher"/>

</launch>
```

Eseguire Gazebo con il robot nel nuovo ambiente personalizzato:

```
$ roslaunch smr_pkg gazebo_custom_sim.launch
```

successivamente avviare un'istanza di Rviz

```
$ rviz
```

ed eseguire i seguenti passi:

- In `Displays` → `Global Options`, impostare `odom` come `fixed_frame`.
- In `Displays`, usare il bottone `Add` per inserire come `Robot Model` il modello contenuto nel parameter server.
- In `Displays`, utilizzare il bottone `Add` per abilitare la visualizzazione grafica di informazioni di tipo `LaserScan` provenienti dal topic `/scan`.
- Eseguire il nodo `turtlebot3_teleop` e provare a muoversi intorno al cilindro.