SAPIENZA
UNIVERSITÀ DI ROMA

Sviluppo di un sistema di Collision Avoidance tramite il metodo dell'Artificial Potential Field

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea in Ingegneria Informatica e Automatica

Candidato

Davide Albano
Matricola 1708530

Relatore

Prof. Giorgio Grisetti

Anno Accademico 2020/2021

Tesi non ancora discussa

Sviluppo di un sistema di Collision Avoidance tramite il metodo dell'Artificial Potential Field

Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Versione:

Email dell'autore: albano.1708530@studenti.uniroma1.it

Sommario

L'obbiettivo di questo lavoro è l'implementazionie di un sistema che permetta a un robot di muoversi all'interno di un ambiente evitando eventuali ostacoli. Il robot preso in considerazione non conosce l'ambiente in cui dovrà muoversi e può quindi basarsi soltanto sulle rilevazioni fatte tramite un laser scanner.

Per l'implementazione è stato usato ROS (Robot Operating System), un framework open source usato per gestire le operazioni e le cominucazioni dei vari nodi di un robot. Per la simulazione è stato usato il pacchetto stage_ros, mentre per la visualizzazione delle informazioni dei vari sensori e della traiettoria calcolata dal robot è stato usato il tool RVIZ.

Data l'assenza di informazioni iniziali riguardo alla mappa è stata necessario scegliere un algoritmo di local motion planning. In particolare è stato scelto il metodo dell'Artificial Potential Field, che è spesso usato per la sua semplicità sia dal punto di vista dello sviluppo sia dal punto di vista computazionale.

Indice

1 Introduzione **5**

1.1 Formulazione problema 5

1.2 Importanza della Collision Avoidance 5

1.2.1 Settore dei trasporti 5

1.2.2 Industry 4.0 6

1.3 Obbiettivo di questo progetto 6

2 Strumenti necessari e teoria **7**

2.1 ROS 7

2.1.1 Caratteristiche principali 7

2.1.2 Publisher/Subscriber 8

2.1.3 Services 8

2.1.4 stage_ros 9

2.1.5 RVIZ 10

2.2 Artificial Potential Field 10

2.2.1 Teoria 10

2.2.2 Applicazione per i robot su ruote 12

3 Descrizione del progetto **13**

3.1 Sistema di riferimento 13

3.2 Nodi 13

3.2.1 twist_generator 14

3.2.2 obstacle_detection 15

3.2.3 cmd_vel_controller 19

3.3 Messaggi 21
3.3.1 geometry_msgs/Twist.msg 21
3.3.2 sensor_msgs/LaserScan.msg 22
3.3.3 visualization_msgs/Marker.msg 22
3.4 Servizi 24
3.4.1 Force.srv 24

4 Esecuzione e test dell'algoritmo 25
4.1 Descrizione dell'esecuzione 25
4.2 Analisi dei test e scelta dei parametri 28

5 Conclusioni 29

Capitolo 1

Introduzione

1.1 Formulazione problema

Lo sviluppo di robot mobili a navigazione autonoma si suddivide in due parti: global path planning e local motion control. Il global path planning utilizza le informazioni che si hanno sulla mappa per trovare il percorso più corto per andare dal punto di partenza all'obbiettivo.

Tuttavia può capitare che le informazioni sull'ambiente in cui il robot dovrà muoversi siano scarse, non aggiornate o mancanti. Questo porta alla necessità di sviluppare algoritmi di local motion control, che permettono al robot di calcolare in tempo reale una traiettoria in grado di evitare gli ostacoli incontrati durante la navigazione. [4]

1.2 Importanza della Collision Avoidance

In un mondo che tende sempre di più all'automazione, il problema della Collision Avoidance diventa sempre di maggiore rilievo. L'utilizzo principale di questo tipo di algoritmi si può trovare principalmente nel settore dei trasporti e nell'Industry 4.0.

1.2.1 Settore dei trasporti

All'interno del settore dei trasporti sono in corso molti studi dal punto di vista dei trasporti via terra. Sono infatti molte le aziende automobilistiche (tra cui spiccano

principalmente Tesla, Waymo e Pony.ai) che stanno investendo nello sviluppo di sistemi di guida autonomi. Questi sistemi devono dunque essere in grado di reagire a tutti i possibili imprevisti che si potrebbero incontrare durante la guida, compresa ovviamente la presenza improvvisa di ostacoli lungo il tragitto.

Anche per gli spostamenti via mare e via aria sono in corso numerosi studi riguardanti la navigazione autonoma, con particolare interesse per i droni sottomarini (UUV) e aerei (UAV).

1.2.2 Industry 4.0

Per quanto riguarda il settore industriale, sempre più aziende stanno passando all'Industry 4.0, che prevede una automazione avanzata all'interno degli impianti produttivi. In fabbriche dove questi robot mobili lavorano anche a contatto con gli operatori diventa essenziale che essi riconoscano la presenza di altri robot e soprattutto di umani, per garantire una maggiore sicurezza. In questo caso entrano in campo gli algoritmi di collision avoidance, perché anche se la mappa generale può essere ben nota, gli ostacoli possono muoversi liberamente all'interno dell'area.

1.3 Obbiettivo di questo progetto

L'obbiettivo di questo progetto è quello di sviluppare un sistema di Collision Avoidance basato su un laser scanner con le seguenti caratteristiche:

Input Il sistema prende in input lo scan del laser e un comando di velocità

Output Il sistema produce un comando di velocità che previene la collisione con gli oggetti rilevati nel contorno, deflettendo la traiettoria del robot verso uno spazio libero da collisioni.

Quindi, a differenza della definizione classica del problema, nel caso analizzato non avremo un obbiettivo da raggiungere ma una velocità iniziale del robot, su cui si andrà a intervenire affinché il robot non vada a sbattere.

Capitolo 2

Strumenti necessari e teoria

2.1 ROS

2.1.1 Caratteristiche principali

ROS (Robot Operating System) è un framework open source utilizzato per lo sviluppo di applicazioni per la robotica. Mette infatti a disposizione strumenti e librerie utili per aiutare gli sviluppatori software nella realizzazione di applicazioni robotiche a partire dalla scrittura fino all'esecuzione e al debugging del codice.

ROS presenta inoltre alcune caratteristiche di un sistema operativo (gestione di processi, di pacchetti e delle loro dipendenze e astrazione di dispositivi hardware a basso livello) e di un middleware perché permette la comunicazione tra processi/macchine diverse.

Infine costituisce un'architettura distribuita in cui è possibile gestire in maniera asincrona un insieme di moduli software che possono essere scritti in vari linguaggi, tra cui C++ e Python.

Uno dei principali punti di forza di ROS è la sua modularità, che gli permette di essere compatibile con robot che hanno caratteristiche molto diverse tra loro. La modularità rende inoltre più facile riutilizzare il codice e permette di integrare ROS con altri framework.

ROS include anche alcuni tool che permettono di simulare hardware e di salvare i dati ottenuti dai vari sensori per poi analizzarli.

Il software viene organizzato in packages. Ogni package può contenere più eseguibili, chiamati nodi. I nodi possono comunicare tra loro in due modi, tramite il meccanismo Publisher/Subscriber oppure tramite il meccanismo dei Services.

ROS è strutturato intorno ad un nodo master che permette ai vari nodi di essere a conoscenza della presenza di altri nodi e di comunicare. Il Master è un nodo unico all'interno dell'architettura di ROS e si occupa di assegnare un nome a ogni nodo connesso al sistema e di registrarlo eventualmente come Publisher, Subscriber o Service Provider. I vari nodi usano una libreria (ROS client library) per poter usufruire delle funzionalità di ROS, attraverso linguaggio C++ (roscpp) o Python (rospy). Al master viene assegnato un well-known XML-RPC URI in modo che qualsiasi nodo creato sia sempre in grado di comunicare con esso.[7]

2.1.2 Publisher/Subscriber

È uno dei due meccanismi usati dai vari nodi ROS per comunicare, ed è una modalità di comunicazione asincrona. La scrittura di un messaggio avviene su di un topic dai nodi di tipo Publisher e tutti i nodi di tipo Subscriber che desiderano ricevere tale messaggio possono iscriversi a quel topic. Per uno stesso topic possono esserci più Publisher e più Subscriber.

Il messaggio è una struttura dati con diversi campi che possono essere di tipi diversi (sono supportati tutti i tipi standard primitivi integer, floating point, boolean, array e costanti). All'interno di un topic è possibile scrivere o leggere solo un tipo di messaggio. La definizione dei vari tipi di messaggi viene memorizzata in file .msg.

Nello sviluppo successivamente illustrato questo meccanismo di comunicazione è stato utilizzato per leggere le informazioni inviate dal laser scanner e la velocità in input e per scrivere in output la velocità ricalcolata in base agli ostacoli presenti.

2.1.3 Services

Questa modalità di comunicazione tra nodi è invece sincrona e usa la semantica Request/Response. Un nodo invia una richiesta a tutti i nodi che forniscono un determinato servizio. Da questi nodi riceverà una risposta.

La struttura dati usata dai servizi è simile a quella dei messaggi, ma ha una sezione per la Request e una per la Response, separate da una riga contenente i caratteri "---". La definizione dei vari tipi di servizi viene memorizzata in file .srv.

Nel caso qui presentato questa modalità è stata utilizzata per fornire il risultato dei calcoli effettuati sugli ostacoli circostanti al nodo che si occuperà poi di determinare la velocità e la direzione di output.

2.1.4 stage_ros

Questo package è stato utilizzato per la simulazione del robot e dell'ambiente circostante, compresi i vari ostacoli:

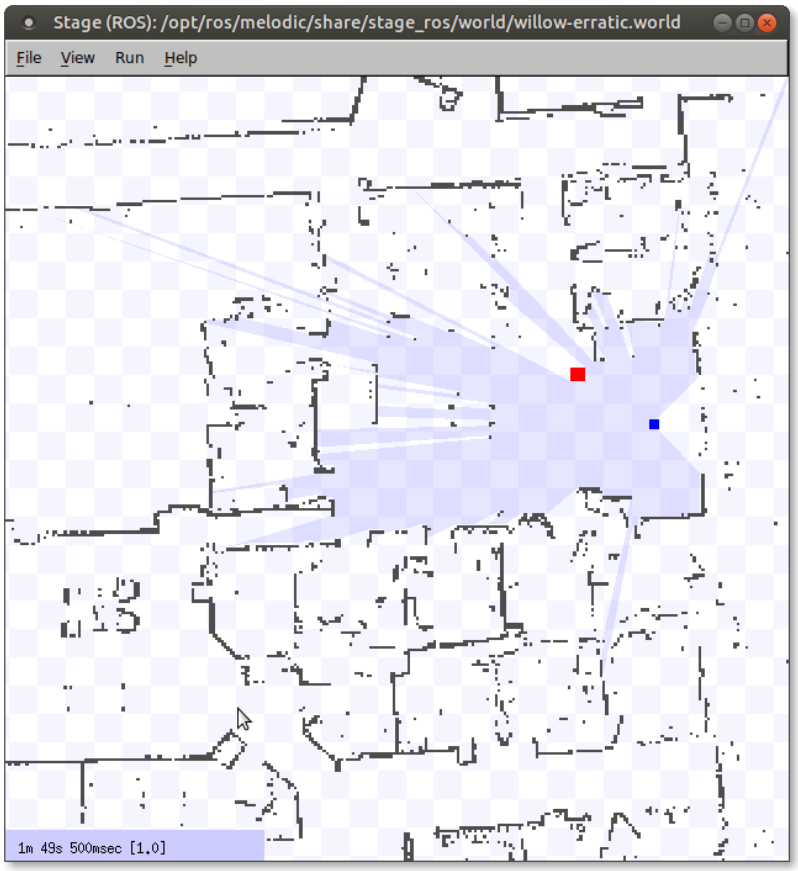

Figura 2.1. Schermata di partenza di stage_ros, in cui si possono vedere il robot (il quadrato blu), il raggio d'azione dello scanner (l'area blu chiaro), gli ostacoli (i tratti grigi) e l'obbiettivo (il quadrato rosso)

2.1.5 RVIZ

RVIZ è stato molto utile per visualizzare in tempo reale le informazioni presenti sui vari topic, come ad esempio la direzione del robot o i rilevamenti effettuati dallo scanner:

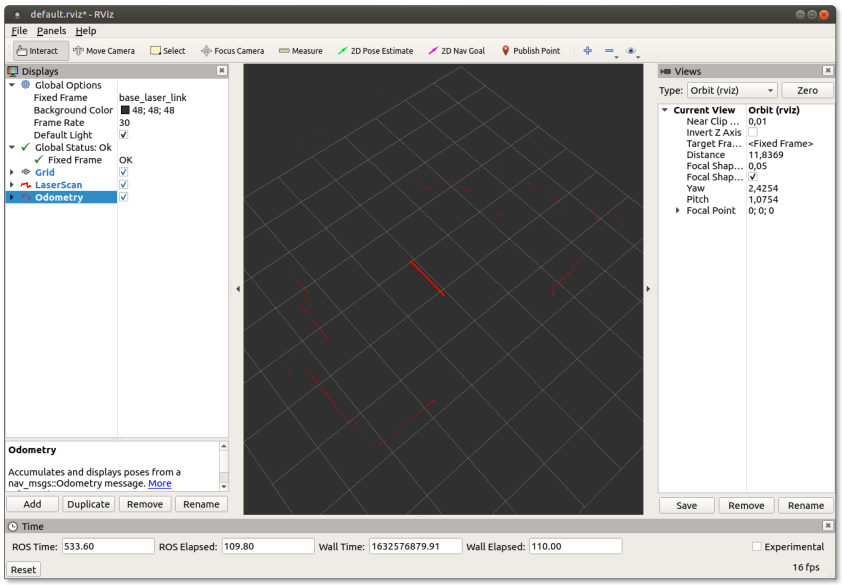

Figura 2.2. Schermata di RVIZ in cui sono state caricate le informazioni dello scanner (i tratti rossi) e della direzione del robot (la freccia rossa)

2.2 Artificial Potential Field

2.2.1 Teoria

Per quanto riguarda il local motion control, ci sono vari approcci ed algoritmi che possono essere utilizzati, come ad esempio il Vector Field Histogram, il Dynamic Window Approach e l'Artificial Potential Field[4]. Quest'ultimo metodo è stato scelto per l'implementazione del progetto.

Il metodo dell'Artificial Potential Field si basa sulla seguente idea: l'obbiettivo del robot si comporta come una forza attrattiva, mentre gli ostacoli si comportano come forze repulsive. Il potenziale totale U sarà la somma delle varie forze attrattive e repulsive e il gradiente negativo $-\nabla U(q)$ indicherà la traiettoria più sicura.

Per quanto riguarda le forze attrattive si combinano due profili di attrazione in base alla distanza del robot dall'obbiettivo: conico quando il robot è lontano, paraboidale quando è vicino. In questo modo la formula per il calcolo della campo attrattivo diventa:

$$U_a(\boldsymbol{q}) = \begin{cases} \frac{1}{2}k_a\|\boldsymbol{e}(\boldsymbol{q})\|^2 & \text{if} \quad \|\boldsymbol{e}(\boldsymbol{q})\| \leq \rho \\ k_b\|\boldsymbol{e}(\boldsymbol{q})\| & \text{if} \quad \|e(\boldsymbol{q})\| > \rho \end{cases}$$

dove $\boldsymbol{e} = \boldsymbol{q}_g - \boldsymbol{q}$, $k_a > 0$ e $k_b > 0$. Per la continuità della funzione è necessario inoltre che:

$$k_a e(\boldsymbol{q}) = k_b \frac{\boldsymbol{e}(\boldsymbol{q})}{\|\boldsymbol{e}(\boldsymbol{q})\|} \quad \text{for} \quad \|\boldsymbol{e}(\boldsymbol{q})\| = \rho$$

e quindi $k_b = \rho k_a$.

Per quanto riguarda invece il campo repulsivo, la formula è la seguente:

$$U_{r,i}(\boldsymbol{q}) = \begin{cases} \frac{k_{r,i}}{\gamma}\left(\frac{1}{\eta_i(\boldsymbol{q})} - \frac{1}{\eta_{0,i}}\right)^{\gamma} & \text{if } \eta_i(\boldsymbol{q}) \leq \eta_{0,i} \\ 0 & \text{if } \eta_i(\boldsymbol{q}) > \eta_{0,i} \end{cases}$$

dove $k_{r,i} > 0$ e $\gamma = 2, 3, \ldots$; $\eta_{0,i}$ è il raggio di influenza, mentre $\eta_i(\boldsymbol{q})$ è il margine ed è calcolato come segue:

$$\eta_i(\boldsymbol{q}) = \min_{\boldsymbol{q}' \in \mathcal{CO}_i} \|\boldsymbol{q} - \boldsymbol{q}'\|$$

dove $\mathcal{CO}_i$ è la obstacole region.

Otteniamo così una somma dei campi totali:

$$U_t(\boldsymbol{q}) = U_a(\boldsymbol{q}) + U_r(\boldsymbol{q})$$

da cui possiamo ricavare la formula della forza totale:

$$\boldsymbol{f}_t(\boldsymbol{q}) = -\nabla U_t(\boldsymbol{q}) = \boldsymbol{f}_a(\boldsymbol{q}) + \sum_{i=1}^{p} \boldsymbol{f}_{r,i}(\boldsymbol{q})$$

[5]

2.2.2 Applicazione per i robot su ruote

Essendo soggetti a determinati vincoli di movimento, nel caso dei robot su ruote non è possibile applicare direttamente la forza risultante appena calcolata. Partendo da una forza artificiale $\boldsymbol{f}_t = \begin{pmatrix} f_{t,x} & f_{t,y} & f_{t,\theta} \end{pmatrix}^T$ possiamo tuttavia ottenere:

$$v = f_{t,x} \cos\theta + f_{t,y} \sin\theta$$
$$\omega = f_{t,\theta}$$

dove θ è l'angolo formato tra l'asse sagittale del robot e l'asse x del sistema di riferimento [6], mentre $f_{t,\theta}$ può essere calcolato nel seguente modo[5]:

$$f_{t,\theta} = k_\theta \left(\operatorname{atan} 2 \left(f_{t,y}, f_{t,x}\right) - \theta\right)$$

Capitolo 3

Descrizione del progetto

3.1 Sistema di riferimento

Data la semplicità del caso in esame è stato possibile scegliere come sistema di riferimento il sistema di riferimento mobile del robot. Questo ha portato a una semplificazione delle formule. Infatti in questo modo la posizione del robot è sempre $(x, y) = (0, 0)$, mentre l'angolo θ definito nel capitolo precedente è sempre 0.

Le formule prima presentate diventano quindi:

$$v = \|(f_{t,x}, f_{t,y})\|$$
$$\omega = \operatorname{atan2}(f_{t,y}, f_{t,x})$$

Infatti tramite ω facciamo ruotare il robot nella direzione della forza risultante, e poi lo facciamo muovere linearmente impostando v uguale al modulo della forza.

3.2 Nodi

Per questo progetto sono stati sviluppati tre nodi ROS:

twist_gnerator Questo nodo si occupa di simulare una serie di comandi di velocità in input.

obstacole_detection Questo nodo si occupa di analizzare le informazioni provenienti dallo scanner e memorizzare i calcoli effettuati su tali informazioni.

cmd_vel_controller Questo nodo si occupa di intercettare i comandi di velocità in input e calcolare la nuova traiettoria basandosi sulle informazioni ricevute da obstacole_detection.

3.2.1 twist_generator

```
#! /usr/bin/env python

import rospy
from geometry_msgs.msg import Twist
from random import random
from time import sleep

class twistGenerator:
    def __init__(self):
        self.pub = rospy.Publisher('/cmd_vel_input', Twist, queue_size
            =10)
        self.move()

    def move(self):
        vel_msg = Twist()
        vel_msg.linear.x = 0
        vel_msg.linear.y = 0
        vel_msg.linear.z = 0
        vel_msg.angular.x = 0
        vel_msg.angular.y = 0
        vel_msg.angular.z = -5 + random() * 10
        self.pub.publish(vel_msg)
        vel_msg.angular.z = 0
        while not rospy.is_shutdown():
            vel_msg.linear.x = random() * 10
            self.pub.publish(vel_msg)



def main():
    rospy.init_node('twist_generator', anonymous=True)
    twist = twistGenerator()
    try:
```

```
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down")

if __name__ == '__main__':
    main()
```

Per poter scrivere sul topic di input della velocità **cmd_vel_input**, questo nodo deve innanzitutto registrarsi come Publisher su quel topic:

```
self.pub = rospy.Publisher('/cmd_vel_input', Twist, queue_size=10)
```

Il nodo andrà quindi a chiamare la funzione **move**, che scriverà sul topic un messaggio con una velocità angolare casuale e poi inizierà a inviare una serie di messaggi con velocità lineare casuale:

```
def move(self):
    vel_msg = Twist()
    vel_msg.linear.x = 0
    vel_msg.linear.y = 0
    vel_msg.linear.z = 0
    vel_msg.angular.x = 0
    vel_msg.angular.y = 0
    vel_msg.angular.z = -5 + random() * 10
    self.pub.publish(vel_msg)
    vel_msg.angular.z = 0
    while not rospy.is_shutdown():
        vel_msg.linear.x = random() * 10
        self.pub.publish(vel_msg)
        self.pub = rospy.Publisher('/cmd_vel_input', Twist, queue_size
            =10)
```

In questo modo il robot partirà con una direzione iniziale sempre diversa e poi continuerà ad andare avanti con velocità diverse.

3.2.2 obstacle_detection

```
#! /usr/bin/env python
```

```
import rospy
from sensor_msgs.msg import LaserScan
import time
import math
from collision_avoidance.srv import Force, ForceResponse
from visualization_msgs.msg import Marker
from tf.transformations import quaternion_from_euler

class obstacleDetection:
    def __init__(self):
        self.sub = rospy.Subscriber('/base_scan', LaserScan, self.
            callback)
        self.pub = rospy.Publisher('marker', Marker, queue_size=10)
        self.srv = rospy.Service('force_service', Force, self.
            force_service)
        self.force = ForceResponse(0, 0)

    def callback(self, msg):
        self.force = ForceResponse(0, 0)
        for i, val in enumerate(msg.ranges, start=0):
            if val >= 1.25:
                continue
            magnitude = 1 / val
            temp_angle = msg.angle_min + (i * msg.angle_increment)
            x = magnitude * math.cos(temp_angle)
            y = magnitude * math.sin(temp_angle)
            angle = math.atan2(-y, -x)
            force = ForceResponse(magnitude, angle)
            self.set_net_force(force)
        self.show_force()

    def set_net_force(self, force):
        x1 = self.force.magnitude * math.cos(self.force.angle)
        y1 = self.force.magnitude * math.sin(self.force.angle)
        x2 = force.magnitude * math.cos(force.angle)
        y2 = force.magnitude * math.sin(force.angle)
        x_total = x1 + x2
        y_total = y1 + y2
```

```
        total_magnitude = math.sqrt(x_total ** 2 +  y_total ** 2)
        total_angle = math.atan2(y_total, x_total)
        self.force = ForceResponse(total_magnitude, total_angle)

    def show_force(self):
        self.marker = Marker()
        self.marker.header.frame_id = '/base_link'
        self.marker.header.stamp = rospy.get_rostime()
        self.marker.ns = 'robot'
        self.marker.id = 0
        self.marker.type = 0
        self.marker.action = 0
        self.marker.pose.position.x = 0
        self.marker.pose.position.y = 0
        self.marker.pose.position.z = 0
        q = quaternion_from_euler(0, 0, self.force.angle)
        self.marker.pose.orientation.x = q[0]
        self.marker.pose.orientation.y = q[1]
        self.marker.pose.orientation.z = q[2]
        self.marker.pose.orientation.w = q[3]
        self.marker.scale.x = self.force.magnitude
        self.marker.scale.y = 0.05
        self.marker.scale.z = 0.05

        self.marker.color.r = 0.0
        self.marker.color.g = 1.0
        self.marker.color.b = 0.0
        self.marker.color.a = 1.0

        self.marker.lifetime = rospy.Duration(0)

        self.pub.publish(self.marker)

    def force_service(self, request):
        return self.force

def main():
    rospy.init_node('obstacle_detection', anonymous=True)
```

```
    od = obstacleDetection()
    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down")

if __name__ == '__main__':
    main()
```

La parte principale di questo nodo è l'analisi delle informazioni che lo scanner scrive nel topic **base_scan**. Per poter leggere queste informazioni il nodo come prima cosa si iscrive al topic, specificando la callback che deve essere chiamata appena viene scritto un nuovo messaggio:

```
self.sub = rospy.Subscriber('/base_scan', LaserScan, self.callback)
```

Questo nodo dovrà inoltre reigstrarsi come provider del servizio **force_service** e inizializzare la variabile **force**. La forza risultante agente sul robot viene infatti calcolata da questo nodo, salvata nella variabile **force** e poi fornita a qualunque nodo la richieda:

```
    self.srv = rospy.Service('force_service', Force, self.force_service
        )
    self.force = ForceResponse(0, 0)
```

```
def force_service(self, request):
    return self.force
```

Ogni volta che lo scanner inserirà una nuova rilevazione sotto froma di messaggio nel topic **base_scan**, verrà quindi chiamata la callback, che filtrerà i dati rilevati in base a un parametro di soglia, ne calcolerà l'equivalente forza repulsiva e infine invocherà per ogni valore calcolato la funzione **set_net_force**:

```
def callback(self, msg):
    self.force = ForceResponse(0, 0)
    for i, val in enumerate(msg.ranges, start=0):
        if val >= 1.25:
            continue
        magnitude = 1 / val
```

```
        temp_angle = msg.angle_min + (i * msg.angle_increment)
        x = magnitude * math.cos(temp_angle)
        y = magnitude * math.sin(temp_angle)
        angle = math.atan2(-y, -x)
        force = ForceResponse(magnitude, angle)
        self.set_net_force(force)
```

La funzione **set_net_force** si occuperà quindi di calcolare progressivamente la risultante delle varie forze calcolate dalla callback e di salvare il risultato nella variabile **force**:

```
def set_net_force(self, force):
    x1 = self.force.magnitude * math.cos(self.force.angle)
    y1 = self.force.magnitude * math.sin(self.force.angle)
    x2 = force.magnitude * math.cos(force.angle)
    y2 = force.magnitude * math.sin(force.angle)
    x_total = x1 + x2
    y_total = y1 + y2
    total_magnitude = math.sqrt(x_total ** 2 +  y_total ** 2)
    total_angle = math.atan2(y_total, x_total)
    self.force = ForceResponse(total_magnitude, total_angle)
```

3.2.3 cmd_vel_controller

```
#! /usr/bin/env python

import rospy
from geometry_msgs.msg import Twist
from random import random
from time import sleep
from collision_avoidance.srv import Force, ForceResponse
import math

class cmdVelController:
    def __init__(self):
        self.pub = rospy.Subscriber('/cmd_vel_input', Twist, self.
            callback)
        self.pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)
```

```
    def callback(self, msg):
        if msg.linear.x == 0:
            self.pub.publish(msg)
            return
        rospy.wait_for_service('force_service')
        try:
            force_service = rospy.ServiceProxy('force_service', Force)
            force = force_service()
            vel_msg = Twist()
            x_linear = msg.linear.x + force.magnitude
            vel_msg.linear.x = x_linear if x_linear < 0.75 else 0.75
            vel_msg.linear.y = 0
            vel_msg.linear.z = 0
            vel_msg.angular.x = 0
            vel_msg.angular.y = 0
            vel_msg.angular.z = msg.angular.z + force.angle
            print(vel_msg)
            self.pub.publish(vel_msg)
        except rospy.ServiceException as e:
            print('Service call failed: %s' %e)


def main():
    rospy.init_node('cmd_vel_controller', anonymous=True)
    cmd = cmdVelController()
    try:
        rospy.spin()
    except KeyboardInterrupt:
        print("Shutting down")

if __name__ == '__main__':
    main()
```

Questo nodo si occupa di leggere la velocità di input e di scrivere in output la velocità ricalcolata in base alla forza repulsiva risultante. Per fare ciò deve registrarsi come Subscriber al topic **cmd_vel_input** e come Publisher al topic **cmd_vel**:

```
    self.pub = rospy.Subscriber('/cmd_vel_input', Twist, self.callback)
```

```
    self.pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)
```

Nella funzione di callback chiamerà quindi il servizio **force_service** che gli fornirà il valore corrente della forza repulsiva risultante. Con il risultato della chiamata a questo servizio popolerà i valori della velocità lineare sull'asse x e di quella angolare sull'asse z. Infine invierà il messaggio sul topic **cmd_vel**, che corrisponde al topic di input del robot:

```
def callback(self, msg):
    if msg.linear.x == 0:
        self.pub.publish(msg)
        return
    rospy.wait_for_service('force_service')
    try:
        force_service = rospy.ServiceProxy('force_service', Force)
        force = force_service()
        vel_msg = Twist()
        x_linear = msg.linear.x + force.magnitude
        vel_msg.linear.x = x_linear if x_linear < 0.75 else 0.75
        vel_msg.linear.y = 0
        vel_msg.linear.z = 0
        vel_msg.angular.x = 0
        vel_msg.angular.y = 0
        vel_msg.angular.z = msg.angular.z + force.angle
        print(vel_msg)
        self.pub.publish(vel_msg)
    except rospy.ServiceException as e:
        print('Service call failed: %s' %e)
```

3.3 Messaggi

3.3.1 geometry_msgs/Twist.msg

La definizione del tipo Twist.msg è presente nella libreria fornita da ROS, ed è la seguente[1]:

```
# This expresses velocity in free space broken into its linear and
    angular parts.
```

```
Vector3  linear
Vector3  angular
```

I messaggi di tipo Twist sono quelli che sono usati sui topic **cmd_vel_input** e **cmd_vel**. Ogni variabile di tipo Vector3 è composta da 3 componenti (x, y e z), che in questo caso corrispondo alle componenti delle velocità lineare e angolare.

3.3.2 sensor_msgs/LaserScan.msg

Anche per quanto riguarda i messaggi inviati sul topic **base_scan** è stato usato un messaggio definito nella libreria ROS[2]:

```
# Single scan from a planar laser range-finder

Header header

float32 angle_min        # start angle of the scan [rad]
float32 angle_max        # end angle of the scan [rad]
float32 angle_increment  # angular distance between measurements [rad]

float32 time_increment   # time between measurements [seconds]
float32 scan_time        # time between scans [seconds]

float32 range_min        # minimum range value [m]
float32 range_max        # maximum range value [m]

float32[] ranges         # range data [m]
float32[] intensities    # intensity data [device-specific units].
```

3.3.3 visualization_msgs/Marker.msg

I messaggi di tipo Marker sono tra i più utilizzati per visualizzare informazioni su RVIZ, e sono definiti nel seguente modo[3]:

```
uint8 ARROW=0
uint8 CUBE=1
uint8 SPHERE=2
uint8 CYLINDER=3
```

```
uint8 LINE_STRIP=4
uint8 LINE_LIST=5
uint8 CUBE_LIST=6
uint8 SPHERE_LIST=7
uint8 POINTS=8
uint8 TEXT_VIEW_FACING=9
uint8 MESH_RESOURCE=10
uint8 TRIANGLE_LIST=11

uint8 ADD=0
uint8 MODIFY=0
uint8 DELETE=2
uint8 DELETEALL=3

Header header                       # time/frame information
string ns                           # namespace
int32 id                           # object ID
int32 type                         # Type of object
int32 action                        # add, modify or deletes objects
geometry_msgs/Pose pose                 # Pose of the object
geometry_msgs/Vector3 scale             # Scale of the object
std_msgs/ColorRGBA color            # Color [0.0-1.0]
duration lifetime
bool frame_locked

geometry_msgs/Point[] points

std_msgs/ColorRGBA[] colors

string text

string mesh_resource
bool mesh_use_embedded_materials
```

Nel progetto questo tipo di messaggio è stato usato per visualizzare su RVIZ la freccia verda che rappresentava la risultante delle forze artificiali che agiscono sul robot in ogni istante.

3.4 Servizi

3.4.1 Force.srv

Mentre per i messaggi è stato possibile usare dei tipi già definiti nella libreria ROS, per quanto riguarda il servizio **force_service** si è rivelato necessario definire un nuovo tipo di messaggio con le informazioni essenziali a rappresentare la forza risultante:

```
---
float64 magnitude
float64 angle
```

Come spiegato precedentemente, le due parti del file .srv separate dai caratteri "---" corrispondono rispettivamente alla Request e alla Responde del servizio. Infatti in questo caso specifico **force_service** non prende nessun parametro di input e restituisce un oggtto con due proprietà: il modulo della forza e l'angolo che essa forma con il robot.

Capitolo 4

Esecuzione e test dell'algoritmo

4.1 Descrizione dell'esecuzione

Per avere un'idea dei calcoli effettuati dall'algoritmo è stata aggiunta una parte di codice che ha la funzione di scrivere sul topic **marker** le informazioni sulla forza risultante calcolata. In questo modo è quindi possibile visualizzare su RVIZ la forza calcolata in ogni istante dell'esecuzione:

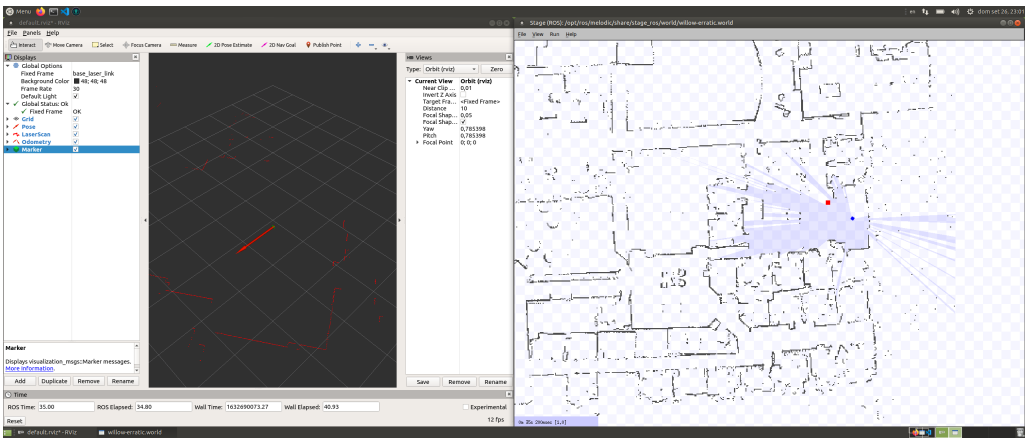

Figura 4.1. Come è possibile notare, nel test analizzato si parte da una situazione in cui la risultante delle forze (rappresentata dalla freccia verde) è nulla, poiché il robot è abbastanza distante da qualsiasi ostacolo.

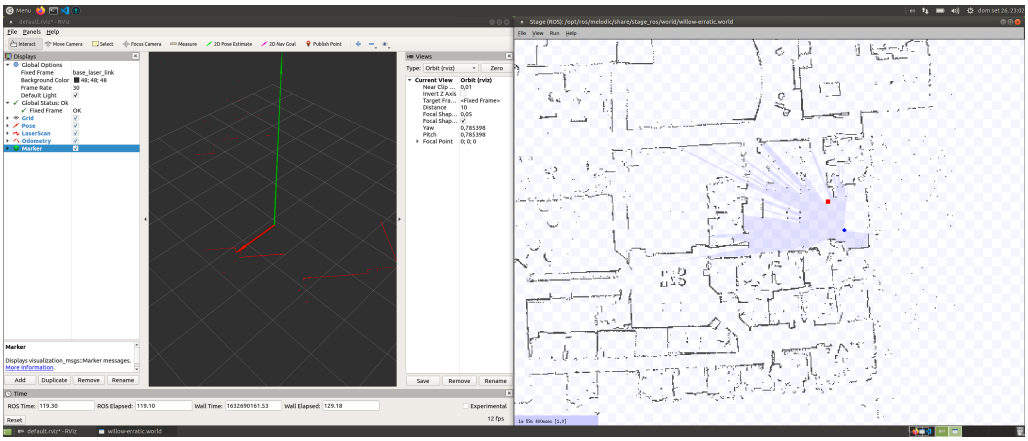

Figura 4.2. Quando si prova ad avvicinare il robot a un ostacolo, la forza risultante aumenta notevolmente.

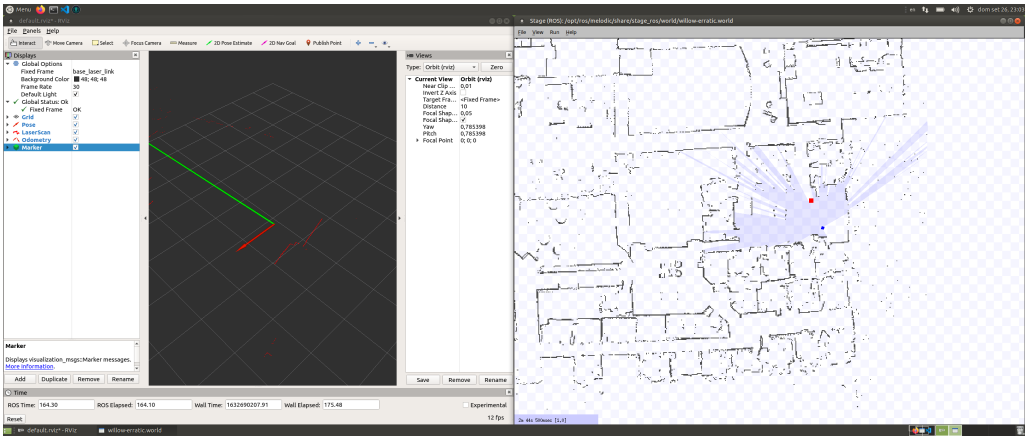

Figura 4.3. La direzione del robot (rappresentata dalla freccia rossa) viene quindi corretta in base alla forza calcolata.

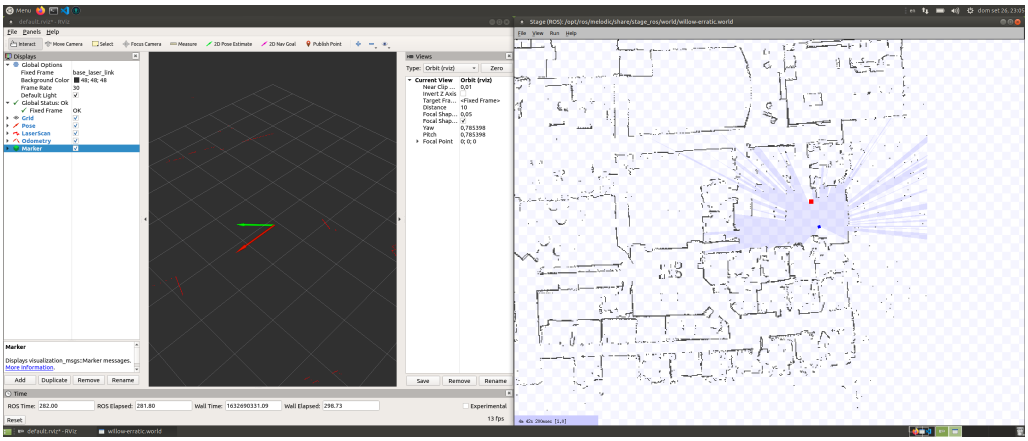

Figura 4.4. Dopo le dovute correzioni, il robot viene portato in una zona in cui la presenza di ostacoli vicini è minore.

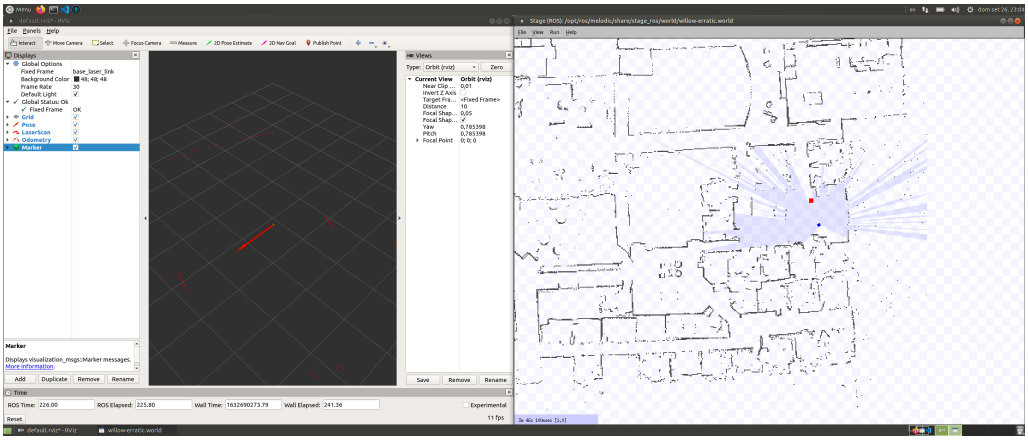

Figura 4.5. Infine, il robot si trova nuovamente in una zona in il rischio di collisioni è minimo.

4.2 Analisi dei test e scelta dei parametri

Durante la fase di test è stato necessario valutare i seguenti parametri:

Soglia misurazioni scanner Questo parametro serve a decidere quanto devono essere vicini gli ostacoli per costituire un possibile problema di collisione.

Coefficiente velocità angolare Questo parametro serve ad attenuare gli effetti della forza sulla velocità angolare.

Coefficiente velocità lineare Questo parametro serve ad attenuare gli effetti della forza sulla velocità lineare.

Dopo aver svolto alcune osservazioni si è giunti a una conclusione.

Per quanto riguarda il primo parametro, è stato scelto 1.25 come distanza di soglia; infatti è stato osservato che valori più bassi causano collisioni in alcune situazioni, mentre valori troppo alti limitano la libertà di manovra del robot, portandolo a volte a rimanere bloccato in un loop.

Per quanto riguarda il secondo ed il terzo parametro non è stato possibile trovare una combinazione di valori adeguata in quanto si è notato che valori troppo alti della velocità lineare di output potevano sempre portare a delle collisioni. Si è optato quindi per limitare la velocità lineare di output a valori inferiori a 0.75, scelta che rende quasi nulla la possibilità di collisione. Con questo valore massimo è stato possibile scegliere pari a 1 il coefficiente per la velocità angolare, che non verrà quindi attenuata.

Capitolo 5

Conclusioni

Durante lo svolgimento di questo progetto è stato possibile capire l'importanza che possono ricoprire nella robotica gli algoritmi di local motion control. Durante i test il robot è stato infatti messo in una condizione di totale assenza di informazioni riguardo alla mappa in cui era situato. Tutti i calcoli fatti dal robot si basavano quindi solo sulle informazioni che potevano essere raccolte in tempo reale. Questo evidenzia la capacità degli algoritmi di local motion control (come ad esempio quello dell'Artificial Potential Field,presentato in questo studio) di reagire prontamente a eventuali imprevisti.

È stato tuttavia messo in evidenza quanto sia importante la scelta dei parametri per il corretto funzionamento dell'algoritmo, che va effettuata a seguito di test e simulazioni. Questi parametri potrebbero infatti cambiare in base all'applicazione che si deve fare dell'algoritmo.

Nonostante l'importanza di questo tipo di algoritmi, è stato tuttavia possibile notare le loro limitazioni e la necessità di associarli con algoritmi di global path planning e SLAM (Simultaneus Localization and Mapping), per avere una accuratezza e delle prestazioni migliori.

Bibliografia

[1] http://docs.ros.org/en/noetic/api/geometry_msgs/html/msg/Twist.html. geometry_msgs/twist message.

[2] http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/LaserScan.html. sensor_msgs/laserscan message.

[3] http://docs.ros.org/en/noetic/api/visualization_msgs/html/msg/Marker.html. visualization_msgs/marker message.

[4] Prof. Marcello Chiaberge Lorenzo Galtarossa. Obstacle avoidance algorithms for autonomous navigation system in unstructured indoor areas. *Politecnico di Torino*, 2018.

[5] Prof. Giuseppe Oriolo. Autonomous and Mobile Robotics - Artificial Potential Fields. *Sapienza, Università di Roma.*

[6] Prof. Giuseppe Oriolo. Autonomous and Mobile Robotics - Motion Control of WMRs: Regulation. *Sapienza, Università di Roma.*

[7] Prof. Giorgio Grisetti Simone Mastrocola. Applicazione di recapito parcelle mediante robot mobili. *Sapienza, Università di Roma*, 2021.