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 discussa il
di fronte a una commissione esaminatrice composta da:

(presidente)

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 in 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 sulla traiettoria calcolate 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

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 necessita 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. [1]

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.

In ROS il software viene organizzato in package. 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 che si occupa di assegnare un nome e registrare ogni singolo nodo connesso al sistema 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.

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

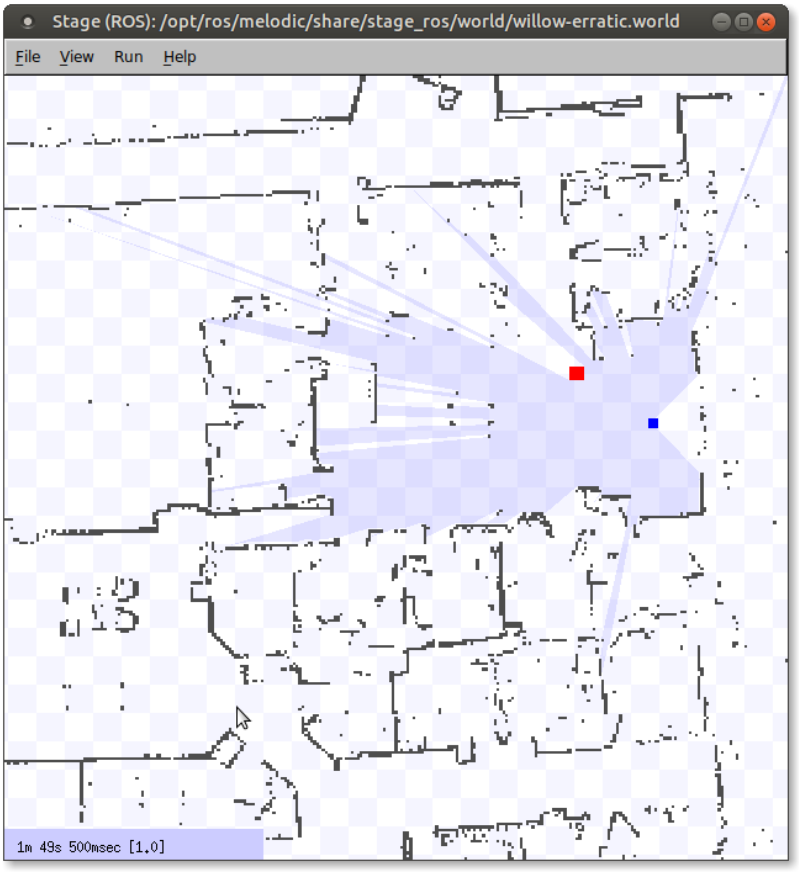
Questa modalità di comunicazione tra nodi è invece sincrona e usca la semantica Request/Response. Un nodo invia una richiesta a tutti i nodi che forniscono un determinato servizio. Da questi nodi riceverà una risposta. La struttura dati usata dai servizi è simile a quella dei messaggi, ma ha una sezione 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 responsabile della lettura della velocità di input e della scrittura della velocità di output.

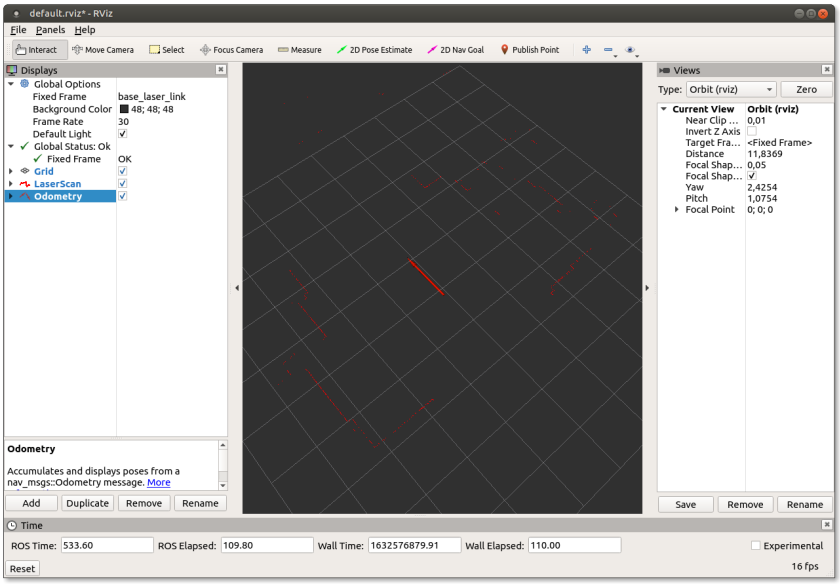
2.1.4 stage_ros

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



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:



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. 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 -∇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:

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

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 calcolata precedentemente. Partendo da una forza artificiale $\boldsymbol{f}_t = \begin{pmatrix} f_{t,x} & f_{t,y} & f_{t,\theta} \end{pmatrix}^T$ possiamo ottenere:

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

[2]

Capitolo 3

Descrizione del progetto

3.1 Obbiettivo

L'obbiettivo di questo progetto era 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.

3.2 Nodi, messaggi e servizi sviluppati

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

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 obstacle_detection

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 noto, 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)
```

3.2.2 cmd_vel_controller

Questo nodo si occupa di leggere la velocità di input e di scrivere in output la velocità ricalcolata in base alla forza repulsiva risultante calcolata da **obsatcle_detection**. 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 questo valore 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)
```

Bibliografia

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

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