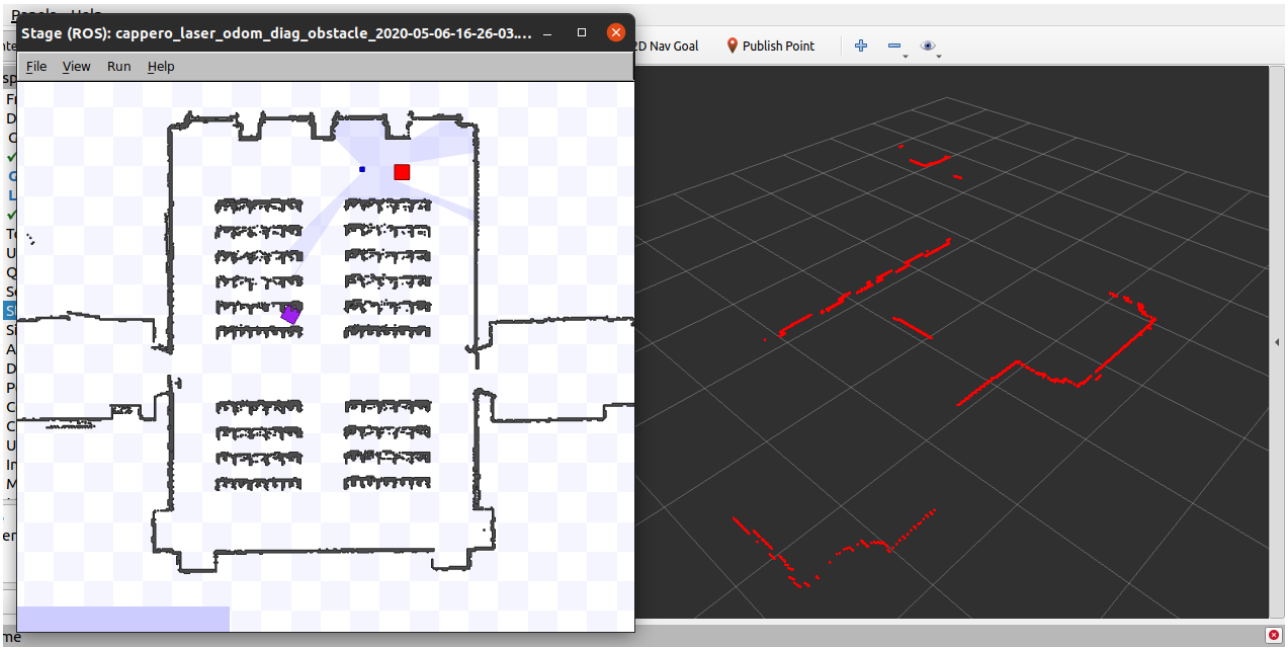
Collision Avoidance

Progetto finale per il Laboratorio di Intelligenza Artificiale e Grafica Interattiva, a.a. 2020-2021.

1. Generalità

Il problema della Collision Avoidance è di importanza cruciale in robotica e si colloca nel contesto del Local Motion Planning: il robot, tramite un sensore montato su di sé, deve essere in grado di percepire ciò che ha intorno e prendere una decisione istantanea sulla direzione in cui muoversi in modo da evitare eventuali ostacoli. Questo è fondamentale perché il robot possa operare in sicurezza, a priori dall'ambiente in cui si muove. In effetti, la Collision Avoidance si applica alla navigazione autonoma anche e soprattutto in ambienti inizialmente sconosciuti, cioè di cui il robot non possiede alcuna conoscenza pregressa. L'unico strumento a sua disposizione è il sensore montato a bordo.



> *Simulazione del robot in movimento con Stage, a sinistra. Display contemporaneo del laser scan con Rviz, a destra.*

Il progetto ha lo scopo di implementare un nodo ROS che realizzi la Collision Avoidance.

Nel nostro caso il nodo ROS è costituito da un programma in C++, che prende in input:

- Un comando di velocità iniziale, dato dall'utente;

- Le informazioni in possesso del robot sull'ambiente in cui si trova, date dal sensore laser montato a bordo.

> *ROS topic di interesse: /avoid_cmd_vel (vedi paragrafo 2.1.) e /base_scan.*

E produce in output:

- Un comando di velocità modificato, che detta lo spostamento effettivo del robot.

> *ROS topic di interesse /cmd_vel.*

Il programma deve mettersi nelle condizioni di ricevere gli input, elaborarli opportunamente ed emettere un output che permetta al robot di muoversi nel modo che vogliamo, senza collidere con gli ostacoli circostanti. Dunque il comando di velocità effettiva sarà sempre in buona sostanza quella che l'utente ha dato in input, ma filtrato dal programma e modificato al nostro scopo.

2. Funzionamento del programma

2.1. Approccio al problema

Dovendo gestire la deviazione della traiettoria del robot in funzione della distanza dall'ostacolo di volta in volta più vicino, si è scelto di pensare al mondo di interesse come ad un campo di forze repulsive. Il robot è rappresentato come una particella che si muove in uno spazio soggetto a delle "forze" prodotte dagli ostacoli circostanti, forze che di fatto identifichiamo con i vettori delle distanze degli ostacoli dal robot. Calcoliamo queste distanze e le prendiamo con segno opposto, in modo tale che più il robot si avvicina, più sarà indotto a rallentare e/o a girarsi.

2.2. Interazione tra componenti ROS

Nodi

Il nostro programma genera un nodo ROS chiamato `collision_avoidance`. Esso dovrà iscriversi a due topic per ricevere gli input di velocità e di laser scan, nonché dichiarare un altro topic su cui pubblicherà il suo output. Gli input vengono a loro volta pubblicati da altri nodi ROS, che nello specifico sono `teleop_twist_keyboard` (se si sta usando questo package come controller remoto) e `stageros` (relativo al programma di simulazione). È possibile verificare quanti e quali nodi sono attivi durante l'esecuzione con il comando:

```
$ rosnode list
/collision_avoidance
/teleop_twist_keyboard
/stageros
(altri nodi relativi al ros master)
```

Topic

Nonostante `collision_avoidance` lavori con messaggi di velocità dallo stesso tipo, si è scelto di di usare due topic separati per evitare di sovrascrivere un comando con l'altro: `/avoid_cmd_vel` è dedicato ai messaggi di input (`teleop_twist_keyboard` scrive su questo topic, `collision_avoidance` legge), mentre `cmd_vel` è per i messaggi di output (`collision_avoidance` scrive su questo topic, `stageros` legge). È possibile verificare quanti e quali topic sono attivi durante l'esecuzione con il comando:

```
$ rostopic list
/avoid_cmd_vel
/base_scan
/cmd_vel
(altri topic relativi al ros master e a stageros)
```

Più specificamente:

- ***/avoid_cmd_vel*** è il topic su cui l'utente pubblica il messaggio di velocità di input e sui cui il nostro programma si mette in ascolto. Se si sta usando il package `teleop_twist_keyboard`, che pubblica di default sul topic `/cmd_vel`, è necessario fare un remapping al topic giusto con il comando seguente: `$ rosrun teleop_twist_keyboard teleop_twist_keyboard.py cmd_vel:=avoid_cmd_vel`.

  → vedasi Figura 1.

- ***/base_scan*** è il secondo topic su cui `collision_avoidance` si mette in ascolto e su cui vengono pubblicati da `stageros` i messaggi di tipo `sensor_msgs/LaserScan` provenienti dal sensore montato sul robot;

- ***/cmd_vel*** è il topic su cui `collision_avoidance` pubblica il messaggio di velocità di output, che detta il movimento effettivo del robot e gli permette di non andare a sbattere contro gli ostacoli. A questo topic si iscrive `stageros`.

  → vedasi Figura 2.

Figura 1.

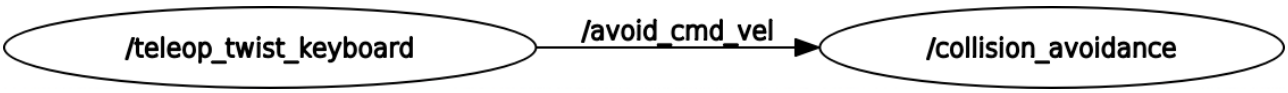
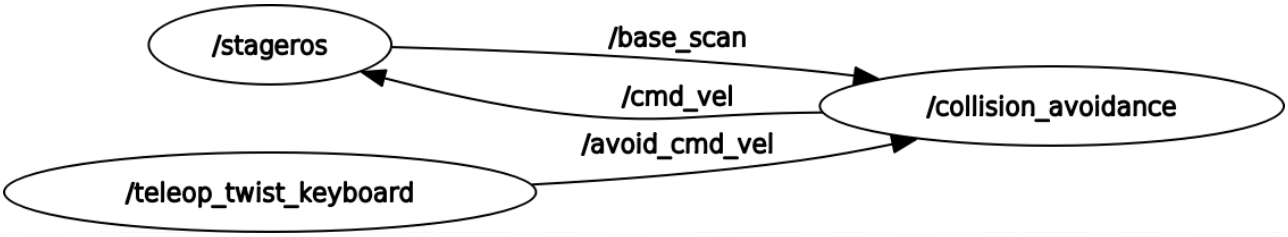


Figura 2.

Grafo computazionale complessivo dei componenti ROS durante l'esecuzione:



> *Diagrammi ottenuti con il tool rqt_graph.*

3. Analisi del programma

3.1. Main

Il metodo principale deve inizializzare correttamente il nodo e instaurare le connessioni con i topic di interesse, di cui ha bisogno per ricevere ed inviare informazioni. Dunque conterrà innanzitutto le istruzioni di base necessarie al funzionamento di programma ROS, e poi le funzioni con cui notifica al ROS master di volersi iscrivere ai topic /avoid_cmd_vel, /base_scan e di voler pubblicare sul topic /cmd_vel.

```
1  int main(int argc, char **argv) {
2
3      ros::init(argc, argv, "collision_avoidance");
4
5      ros::NodeHandle n;
6      ros::Rate loop_rate(10);
7
8      ROS_INFO("Ho avviato il nodo...");
9
10     //creo il subscriber per ricevere comandi laser scan
11     ros::Subscriber sub1 = n.subscribe("base_scan", 1000,
   laserScanCallback);
12
13     //creo il subscriber per ricevere comandi di velocità
```

```
14      ros::Subscriber sub2 = n.subscribe("avoid_cmd_vel", 1000,
    cmdVelCallback);
15
16      //dico al publisher di inviare i comandi di velocità
17      cmd_vel_pub = n.advertise<geometry_msgs::Twist>("cmd_vel",
    1000);
18
19      ROS_INFO("Inviare al topic /avoidance_cmd_vel il comando per
    far muovere il robot");
20      ros::spin();
21
22      return 0;
23  }
```

- `ros::NodeHandle n;` alloca l'oggetto che inizializza pienamente il nodo;
- `ros::init(argc, argv, "collision_avoidance");` deve essere necessariamente invocata per poter utilizzare il sistema ROS nel programma e definisce il nome del nodo;
- `ros::Subscriber sub1 = n.subscribe("base_scan, 1000, laserScanCallback");` notifica al ROS master che vogliamo ricevere messaggi su questo topic. I messaggi ricevuti vengono passati a una funzione di callback, qui `laserScanCallback`;
- `ros::Subscriber sub2 = n.subscribe("avoid_cmd_vel", 1000, cmdVelCallback);` notifica al ROS master che vogliamo ricevere messaggi su questo topic. I messaggi ricevuti vengono passati a una funzione di callback, qui `cmdVelCallback`;
- `cmd_vel_pub = n.advertise<geometry_msgs::Twist>("cmd_vel", 1000);` notifica al ROS master che vogliamo pubblicare sul topic `cmd_vel` e ritorna un oggetto Publisher che ci permette effettivamente di publicare informazioni (tramite la funzione `publish()`, vedi paragrafo 3.5.).
- `ros::spin();` processa messaggi dalla coda delle callback. Senza questa istruzione, le callback non verrebbero mai chiamate.

3.2. Callback del topic cmd_vel

Le funzioni di callback racchiudono il cuore del programma. Esse si occupano di recepire i dati di input sui topic cui il nodo è iscritto, processarli e costruire il messaggio finale. Questo viene poi pubblicato sul topic `/cmd_vel` e detta gli effettivi spostamenti del robot. In particolare, la callback relativa al topic `/avoid_cmd_vel` si occupa semplicemente di controllare se è stato ricevuto un messaggio di velocità e di salvare i dati di interesse in opportune variabili. Infine, stampa a terminale il loro valore.

```
1  void cmdVelCallback(const geometry_msgs::Twist::ConstPtr& msg) {
2      //ho ricevuto il comando di velocità
3      cmd_received = true;
4      vel_received = *msg;
5      vel_received_x = msg->linear.x;
6      vel_received_y = msg->linear.y;
7      vel_received_angular = msg->angular.z;
8      ROS_INFO("Ho ricevuto il comando!\nlinear_x = %f, linear_y =
   %f, angular = %f", vel_received_x, vel_received_y,
   vel_received_angular);
9  }
```

- `cmd_received = true;` setta la variabile flag `cmd_received` a true, poiché se la callback viene eseguita vuol dire che è stato ricevuto un messaggio;
- `vel_received = *msg;` salva in `vel_received` il comando di velocità ricevuto.

3.3. Callback del topic laser_scan

La callback del topic `/base_scan` ha tre compiti:

1. Controllare di aver ricevuto un comando di velocità e, in caso positivo, resettare la variabile di flag;
2. Invocare le funzioni dei package `tf` e `laser_geometry` per convertire il laser scan in input in una nuvola di punti 3D (riga 13). Si ricorda la struttura dei diversi tipi di messaggio:

SENSOR_MSGS/LASERSCAN	SENSOR_MSGS/POINTCLOUD
Laser Scan	Point Cloud
std_msgs/Header header float32 angle_min float32 angle_max float32 angle_increment float32 time_increment float32 scan_time float32 range_min float32 range_max float32[] ranges float32[] intensities	std_msgs/Header header [geometry_msgs/Point32] points [sensor_msgs/ChannelFloat32] channels

3. Ricavare l'isometria che permette di passare dal sistema di riferimento del sensore laser a quello del robot (righe 15-28). Il blocco *try-catch* estrae la trasformata tra i due sistemi di riferimento e gestisce eventuali eccezioni, mentre la funzione `convertPose2D` rende la trasformata un'effettiva isometria in forma di matrice 2x2. La nuvola di punti e l'isometria vengono poi passate come parametri ad una funzione ausiliaria `transformOperations()`, cui si è cercato di dare un nome autoesplicativo e che si occuperà di trattare questi dati da un punto di vista geometrico.

```
1  void laserScanCallback(const sensor_msgs::LaserScan::ConstPtr&
   msg) {
2      //devo ricevere il comando di velocità
3      if (!cmd_received)
4          return;
5      cmd_received = false;
6
7      tf::TransformListener listener;
8      tf::StampedTransform transform_obstacle;
9      laser_geometry::LaserProjection projector;
10     sensor_msgs::PointCloud cloud;
11
12     //funzione che converte un messaggio di tipo laser scan ad
   un messaggio di tipo point cloud
13     projector.transformLaserScanToPointCloud("base_laser_link",
   *msg, cloud, listener);
14
15     try {
16     //aspetto di avere una trasformata disponibile
17     //parametri: sistema di riferimento di arrivo, sistema di
   riferimento di partenza, tempo, timeout
18     listener.waitForTransform("base_footprint",
   "base_laser_link", ros::Time(0), ros::Duration(5.0));
19     //estraggo la trasformata tra i due sistemi di riferimento e
   la memorizzo in transform_obstacle
20     listener.lookupTransform("base_footprint",
   "base_laser_link", ros::Time(0), transform_obstacle);
21     }
22     catch (tf::TransformException &e) {
23         ROS_ERROR("%s", e.what());
24         ros::Duration(1.0).sleep();
25         return;
26     }
27
```

```
28      Eigen::Isometry2f laser_matrix =
    convertPose2D(transform_obstacle); //converto la trasformata in
    matrice 2D
29
30      transformOperations(cloud, laser_matrix);
31  }
```

3.4. Funzione transformOperations

Per le operazioni geometriche e relative alla deviazione della traiettoria del robot, si è cercato di modularizzare il codice incapsulandolo in due funzioni separate. La prima, `transformOperations()`, definisce le variabili che identificano le relative posizioni del robot e dell'ostacolo, per poi passarle alla seconda, `avoidanceOperations()`, cui è delegato il compito specifico di prendere provvedimenti per non mandare il robot a sbattere. È qui che si applica la strategia di pensare allo spazio in cui si muove il robot come ad un campo di forze repulsive che vengono esercitate dagli ostacoli circostanti. Così, il vettore distanza tra robot e ostacolo può essere interpretato come vettore che rappresenta la "forza" sussistente tra i due (o meglio, la direzione della forza repulsiva che l'ostacolo esercita sul robot). Una volta calcolata tale distanza, possiamo prenderla con segno opposto per pensare di frenare il robot.

Per prima cosa, `transformOperations()` fa un ciclo su tutti i punti della nuvola contenente le informazioni del laser scan; per ognuno di essi salva le coordinate x e y in nel vettore `obstacle_position`, che calcola nel sistema di riferimento del robot grazie all'isometria. Successivamente calcola la norma della distanza tra l'ostacolo e il robot, e definisce due "forze" in modulo che altro non sarebbero che le coordinate degli ostacoli normalizzate sulla distanza al quadrato. Alla fine, `force_x` e `force_y` saranno le forze risultanti (o meglio, le componenti x ed y delle forze risultanti) dalla somma (normalizzata) dei vettori distanza da tutti gli ostacoli. In conclusione, le si prende con segno opposto per poterne sfruttare il carattere "repulsivo" e utilizzarle nella deviazione della traiettoria del robot. Poi viene invocata la funzione che si occupa effettivamente della deviazione.

```
1   void transformOperations(sensor_msgs::PointCloud c,
    Eigen::Isometry2f lm) {
2
3       Eigen::Vector2f obstacle_position;
4
5       float force_x = 0.0;
6       float force_y = 0.0;
7       float obstacle_distance;
8
```

```
9       for (auto& point: c.points) { //ciclo su tutti i punti della
    nuvola
10
11          /*
12              p_i (x,y): posa ostacolo   ->   obstacle_position[2]
13              t (x,y): posa robot
14              t - p_i: direzione forza risultante
15              1/norm(t_i - p_i): modulo forza risultante
16          */
17
18          obstacle_position(0) = point.x;
19          obstacle_position(1) = point.y;
20          obstacle_position = lm * obstacle_position; //posizione
    ostacolo nel robot frame
21
22          obstacle_distance = sqrt(point.x * point.x + point.y *
    point.y);
23
24          force_x += (obstacle_position(0) / obstacle_distance) /
    obstacle_distance;
25          force_y += (obstacle_position(1) / obstacle_distance) /
    obstacle_distance ;
26      }
27
28      //prendiamo le forze uguali in modulo ma con verso opposto
    per far fermare il robot
29      force_x = -force_x;
30      force_y = -force_y;
31
32      avoidanceOperations(force_x, force_y, obstacle_distance);
33  }
```

3.5. Funzione avoidanceOperations

È la funzione che si occupa di far deviare la traiettoria del robot quando si trova troppo vicino ad un ostacolo. Il corpo di `avoidOperations()` definisce, elabora e pubblica il messaggio di velocità finale da pubblicare su `cmd_vel`. È ovvio che lo spostamento dovrà dipendere in una certa misura dalla vicinanza del robot all'ostacolo. Si è deciso di prendere una frazione di questa grandezza (riga 3) e di mischiarla alla componente y della forza per definire la nuova velocità angolare, sempre dipendente anche da quella iniziale ricevuta in input. Analoghi calcoli vengono svolti per le componenti della velocità lineare. Infine, si usa l'oggetto `ros::Publisher cmd_vel_pub` dichiarato globalmente per

pubblicare sul topic `cmd_vel` il messaggio così costruito che permette al robot di non andare a sbattere.

```
void avoidanceOperations(float fx, float fy, float ob_dist) {

    geometry_msgs::Twist msg_final;

    float d = ob_dist/20;

    //rotazione robot
    msg_final.angular.z = d * fy/60;
    fx *= abs(vel_received_x)/600;

    if (fx < 0) {
        msg_final.linear.x = fx + vel_received_x;
    }
    else {
        msg_final.linear.x = -fx + vel_received_x;
    }
    msg_final.angular.z += vel_received_angular;
    msg_final.linear.y = fy + vel_received_y;

    cmd_vel_pub.publish(msg_final);
}
```

- `cmd_vel_pub.publish(msg_final);` istruzione che pubblica su `/cmd_vel` il comando di velocità le cui componenti sono state costruite nelle righe di codice precedenti: `msg_final` è il messaggio in base al quale il robot effettivamente si sposta.