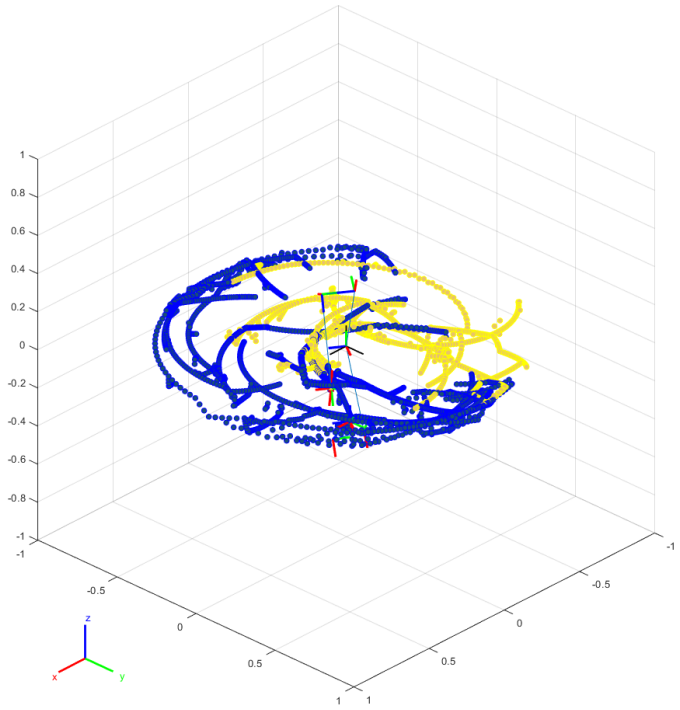
POLITECNICO DI BARI
Dipartimento di Ingegneria Elettrica e dell'Informazione

Laurea Magistrale Ingegneria dell'Automazione

Corso di Robotics – Industrial Handlings

CBiRRT



Professore:
Ing. Paolo Lino

Studenti:
Alessandro Quatela
Giuseppe Roberto

Anno Accademico 2021/2022

Indice

1 Introduzione

La vita quotidiana è piena di compiti che vincolano il nostro movimento. Portare una tazza di caffè, sollevare un oggetto pesante, o spostare una brocca di latte fuori dal frigorifero, sono esempi di compiti che comportano vincoli imposti al nostro corpo così come agli oggetti manipolati. Consentire ai robot autonomi di eseguire tali compiti comporta l'elaborazione di movimenti soggetti a vincoli multipli, spesso simultanei. Per permettere ai robot di eseguire compiti simili, è necessario un algoritmo di pianificazione del movimento che possa generare traiettorie che obbediscano ai vincoli delle articolazioni e facciano scorrere gli oggetti lungo le superfici di supporto quando necessario.

Il raggiungimento di tale obiettivo riscontra una serie di difficoltà da parte del pianificatore di traiettorie. Le configurazioni consentite del robot non sono note a priori da parte del pianificatore, ma l'algoritmo deve scoprire le configurazioni valide mentre pianifica.
Nel contesto della pianificazione del movimento, questa esplorazione viene fatta attraverso il campionamento. Tuttavia, campionare i vincoli di posa in modo efficiente e probabilisticamente completo è difficile. Non è chiaro come rappresentare tali vincoli in modo generale, come assicurare che l'esplorazione sia efficiente e come garantire la completezza probabilistica.

Per passare dalla configurazione iniziale a quella finale il pianificatore della traiettoria deve trovare un percorso adatto al robot rispettando determinati vincoli ed evitando gli ostacoli. Considerando lo spazio di configurazione di un manipolatore, è possibile definire alcune zone chiamate manifolds dove i vincoli sono rispettati. Per manipolatori aventi molti DOF il calcolo dei manifolds diventa particolarmente oneroso. Di conseguenza, il pianificatore non ha alcuna conoscenza preliminare della forma dei manifolds. La mancanza di conoscenza preventiva della forma dei manifolds collettore preclude l'utilizzo di un controllo nello spazio di lavoro.

Trovare una configurazione all'interno del manifold attraverso il campionamento casuale delle variabili di giunto è estremamente improbabile. Questo, preclude l'uso di pianificatori standard basati sul campionamento casuale, come l'RRT o il Probabilistic Road Map (PRM) che campionano direttamente lo spazio di configurazione.
Una soluzione a questo problema è l'utilizzo dell'algoritmo CBiRRT capace di esplorare lo spazio di configurazione rispettando i vincoli ed evitando gli ostacoli.

1.1 Stato dell'Arte

Il pianificatore Constrained Bi-directional Rapidly-Exploring Random Tree (CBiRRT), affronta il problema del campionamento su manifold vincolati. Il CBiRRT prima campiona nello spazio di configurazione e poi usa operazioni di proiezione per spostare i campioni sui manifolds quando necessario. Questa tecnica permette al pianificatore di esplorare efficientemente i manifolds e di costruire percorsi interni ad essi.

I vincoli di posa sono rappresentati tramite i Task Space Regions (TSRs). I TSR sono semplici da definire, possono essere campionati in modo efficiente, e la distanza dal TSR può essere calcolata rapidamente, rendendoli ideali per la pianificazione basata sul campionamento.
I TSR possono essere concatenati insieme per creare vincoli di posa più complessi per l'end-effector.

L'algoritmo CBiRRT si basa su diversi sviluppi nella pianificazione del movimento e nella ricerca sul controllo dei robot. Il BiRRT rappresenta la versione bidirezionale dell'algoritmo Rapidly-exploring Random Tree (RRT) capace di esplorare lo spazio di configurazione, dove due alberi, uno cresciuto dalla configurazione di partenza ed uno cresciuto dalla configurazione finale, esplorano alternandosi lo spazio e tentano di connettersi tra loro. Il CBiRRT sfrutta tale strategia di ricerca combinandola con metodi di proiezione che spostano i campioni dallo spazio di configurazione verso i manifolds, risultando efficace per i problemi di pianificazione del movimento che coinvolgono i vincoli.

Nella letteratura robotica, i metodi di proiezione sono sorti nel contesto della ricerca nei controlli e nella cinematica inversa. Gli algoritmi di cinematica inversa iterativa usano metodi di proiezione basati sulla pseudo-inversa o sulla trasposta dello Jacobiano per spostare iterativamente l'end-effector di un robot più vicino alla posa desiderata nello spazio di lavoro.

1.2 Obiettivo e Approccio

L'obiettivo di questo progetto è implementare su MATLAB l'algoritmo Constrained Bi-directional RapidlyExploring Random Tree (CBiRRT) per la pianificazione di percorsi nello spazio di configurazione con vincoli per robot manipolatori al fine di evitare gli ostacoli.
Il CBiRRT prima campiona nello spazio di configurazione e poi usa operazioni di proiezione (quando necessario) per spostare i campioni dallo spazio di configurazione verso il manifold (zona in cui i vincoli sono rispettati). In questo modo vengono costruiti i percorsi all'interno dei manifold.

Nell' implementazione sono stati definiti parametri, classi e funzioni al fine di ottenere un codice generale versatile e adattabile in base all'applicazione.

Il progetto è presentato spiegando le tappe percorse nell'implementazione:

- Definizione delle classi Node Tree e TSR e delle loro relative funzioni come prerequisiti all'algoritmo.
- Implementazione dello pseudocodice mostrato negli articoli di Dmitry Berenson riportati in bibliografia.
- Implementazione delle funzioni aggiuntive necessarie per il controllo dell'algoritmo CBiRRT e per la visualizzazione dei risultati.
- Presentazione di un esempio di utilizzo con conclusioni e risultati

I file del codice sono disponibili su GitHub al seguente link:
https://github.com/gmeidk/CBiRRT.

2 Prerequisiti

L'implementazione dell'algoritmo in MATLAB richiede l'installazione del Robotics System Toolbox.
L'algoritmo CBiRRT opera facendo crescere nello spazio di configurazione del robot due alberi (***Tree***), uno a partire dalla configurazione (***Node***) di partenza e uno dalla configurazione dell'obiettivo.
Gli alberi cercano di connettersi tra loro, esplorando lo spazio di configurazione e verificando le condizioni dei vincoli definiti nel ***TSR***.
È necessario dunque implementare le classi Node, Tree e TSR alla base della logica dell'algoritmo.

2.1 Classe Node

La classe `Node` è composta da 3 proprietà elencate nella seguente tabella:

NOME	TIPO DI DATO	DESCRIZIONE
`q`	array(float)	rappresenta la configurazione del robot nello spazio di configurazione.
`node_index`	int	rappresenta l'indice del nodo all'interno dell'albero (default 1, indica che è il nodo generatore dell'albero).
`parent_index`	int	indica il nodo dal quale è stato generato (default 0, indica il nodo generatore dell'albero), utile per ricostruire l'albero.

Nella seguente tabella sono elencate le funzioni utili correlate alla classe `Node`:

NOME	INPUT	OUTPUT	DESCRIZIONE
`Node`	q, node_index, parent_index	node	date le proprietà in ingresso, restituisce in uscita l'oggetto Nodo.
`node2config`	node, robot	config	estrae la configurazione del nodo relativa al robot in ingresso.
`showNode`	node, robot	figure	mostra in figura la configurazione del nodo relativa al robot in ingresso.
`nodeDistance`	node1, node2	distance	calcola la distanza (angdiff) tra le configurazioni dei due nodi in ingresso.
`path`	node, tree, isBackward	path	costruisce il percorso a ritroso dato nodo e albero in ingresso.
`directKin`	node, robot	ee_position	fornisce la matrice di trasformazione all'end-effector nelle coordinate di base, calcolando la cinematica diretta dati in ingresso il robot e il nodo.
`isWrapped`	node	isWrapped	dato il nodo in ingresso controlla che i valori delle variabili di giunto siano compresi nell'intervallo [-π, π].
`checkCol`	node, robot, false_collision	isCollided	controlla se la configurazione in ingresso presenta delle collisioni, ignorando quelle presenti nel vettore false_collision.
`config2node`	config, robot	node	data una configurazione di ingresso genera il realtivo nodo.
`tform2node`	robot, tform, weights	node	data una matrice di trasformazione e dei pesi in ingresso genera il nodo.

2.2 Classe Tree

La classe `Tree` è composta da 2 proprietà elencate nella seguente tabella:

NOME	TIPO DI DATO	DESCRIZIONE
`node_array`	array(Node)	vettore contenente i Nodi che compongono l'albero.
`isBackward`	bool	variabile booleana che specifica il tipo di albero: $\begin{cases} 1 & node_{goal} \in Tree \\ 0 & node_{start} \in Tree \end{cases}$

Nella seguente tabella sono elencate le funzioni utili correlate alla classe `Tree`:

NOME	INPUT	OUTPUT	DESCRIZIONE
`Tree`	root_node, isBackward	tree	date le proprietà in ingresso, restituisce in uscita l'oggetto Tree.
`addNode`	tree, node	tree, node_index	aggiunge un nodo all'albero in ingresso e ne restituisce anche la sua posizione.

2.3 Classe TSR

La classe `TSR` è composta da 14 proprietà elencate nella seguente tabella:

NOME	TIPO DI DATO	DESCRIZIONE
T_w^0	tform_matrix	matrice di trasformazione relativa all'oggetto w nelle coordinate di base.
T_e^w	tform_matrix	offset dell'end-effector relativo al sistema di coordinate dell'oggetto w.
x_{min}, x_{max}	float	vincoli relativi all'asse x.
y_{min}, y_{max}	float	vincoli relativi all'asse y.
z_{min}, z_{max}	float	vincoli relativi all'asse z.
R_{min}, R_{max}	float	vincoli relativi all'angolo di roll.
P_{min}, P_{max}	float	vincoli relativi all'angolo di pitch.
Y_{min}, Y_{max}	float	vincoli relativi all'angolo di yaw.

Nella seguente tabella sono elencate le funzioni utili correlate alla classe `TSR`:

NOME	INPUT	OUTPUT	DESCRIZIONE
`TSR`	T_w^0,T_e^w,$\mathrm{B_w}$	tsr	date le proprietà in ingresso ($\mathrm{B_w}$ matrice dei vincoli), restituisce in uscita l'oggetto TSR.
`print`	tsr	string	stampa nella Command Window le proprietà del TSR in ingresso.
`TSRDistance`	tsr, T_s^0	$\mathrm{D_w}$	restituisce la distanza dal TSR estratta dalla matrice $T_{s\prime}^w = (T_w^0)^{-1}[T_s^0(T_e^w)^{-1}]$
`displacement`	tsr, T_s^0	Δx	restituisce la distanza dal TSR tenendo conto del range dei vincoli.

3 Implementazione CBiRRT

L'algoritmo CBiRRT è implementato seguendo lo pseudocodice presente negli articoli riportati in bibliografia. Nella figura è riportato lo pseudocodice del CBiRRT.

Algorithm 1: CBiRRT(Q_s, Q_g)

```
1  T_a.Init(Q_s); T_b.Init(Q_g);
2  while TimeRemaining() do
3      q_rand ← RandomConfig();
4      q^a_near ← NearestNeighbor(T_a, q_rand);
5      q^a_reached ← ConstrainedExtend(T_a, q^a_near, q_rand);
6      q^b_near ← NearestNeighbor(T_b, q^a_reached);
7      q^b_reached ← ConstrainedExtend(T_b, q^b_near, q^a_reached);
8      if q^a_reached = q^b_reached then
9          P ← ExtractPath(T_a, q^a_reached, T_b, q^b_reached);
10         return SmoothPath(P);
11     else
12         Swap(T_a, T_b);
13     end
14 end
15 return NULL;
```

L'algoritmo CBiRRT riceve in ingresso il nodo di start e il nodo di goal dai quali costruirà i relativi alberi. Finchè non viene raggiunta la condizione di stop `TimeRemaining` (`max_iteration` nell'implementazione) l'algoritmo continua l'esecuzione. In ogni iterazione l'algoritmo genera una configurazione casuale del robot q_{rand} (un nodo n_{rand} nell'implementazione) attraverso la funzione `RandomConfig`. Tramite la funzione `NearestNeighbor` viene trovata la configurazione q^a_{near} (n^a_{near}) appartenente all'albero A più vicina a q_{rand} (n_{rand}). La funzione `ConstraintExtend` fa crescere l'albero A dalla configurazione q^a_{near} (n^a_{near}) verso q_{rand} (n_{rand}) attraverso un ramo che congiunge la q_{rand} (n_{rand}) alla q^a_{reach} (n^a_{reach}) la quale rispetta i vincoli del TSR.
Le funzioni `NearestNeighbor` e `ConstraintExtend` vengono utilizzate anche per espandere l'albero B che utilizza la q^a_{reach} (n^a_{reach}) al posto della q_{rand} (n_{rand}). Se l'albero B si connette all'albero A $q^a_{reach} = q^b_{reach}$ ($nodeDistance(n^a_{reach}, n^b_{reach}) \leq max_{step}$), viene restituito il percorso trovato, altrimenti gli alberi vengono scambiati e il processo viene ripetuto.

Segue l'implementazione MATLAB corrispettiva dello pseudocodice dell'algoritmo CBiRRT.

```
function [path, debug] =
CBiRRT(n_start,n_goal,robot,TSR,check_self_collision,max_step,eps,max_iteration)

try
    bar = waitbar(0,'Ricerca in corso...', 'Name','CBiRRT - Quatela, Roberto',
'CreateCancelBtn','setappdata(gcbf,''canceling'',1)');
    setappdata(bar,'canceling',0);

    % if check_self_collision is true calculate the false collision array
    if check_self_collision
        false_collision = FindFalseCollision(robot);
        disp('False Collision occured in HomeConfiguration, those will be neglected.');
    else
        false_collision = [];
    end

    % check correctness of CBiRRT input params

CheckInputParams(n_start,n_goal,robot,TSR,check_self_collision,max_step,eps,max_iteration,false_c
ollision);

    % algorithm initialization
    Ta = Tree(n_start, false);
    Tb = Tree(n_goal, true);

    iterations = 1;

    dist = zeros([1,max_iteration]);

    n_reach1_min = n_start;
    n_reach2_min = n_goal;
    min_dist = n_reach1_min.nodeDistance(n_reach2_min);
    n_reach1_min_back = false;


    time_it_old = 0.28;


    while true

        % stop condition
        if (iterations > max_iteration) | getappdata(bar,'canceling')
            if n_reach1_min_back == Ta.isBackward
                path_a = n_reach1_min.path(Ta, n_reach1_min_back);
                path_b = n_reach2_min.path(Tb, ~n_reach1_min_back);
            else
                path_a = n_reach1_min.path(Tb, n_reach1_min_back);
                path_b = n_reach2_min.path(Ta, ~n_reach1_min_back);
            end

            if ~n_reach1_min_back
                path = [path_a, path_b];
            else
                path = [path_b, path_a];
            end
            disp("Soluzione non trovata");
            break
        end
```

```
        tic

        n_rand = RandomConfig(robot);

        n_a_near = NearestNeighbor(Ta,n_rand);

               [Ta, n_a_reach] =
ConstraintExtend(Ta,n_a_near,n_rand,TSR,check_self_collision,false_collision,robot,max_step,Inf,e
ps);

        n_b_near = NearestNeighbor(Tb,n_a_reach);

        [Tb, n_b_reach] =
ConstraintExtend(Tb,n_b_near,n_a_reach,TSR,check_self_collision,false_collision,robot,max_step,In
f,eps);

        % save the calculated distance (debug.history)
        dist(iterations) = nodeDistance(n_a_reach,n_b_reach);

        % save the best result obtained
        if dist(iterations) < min_dist
            n_reach1_min = n_a_reach;
            n_reach1_min_back = Ta.isBackward;
            n_reach2_min = n_b_reach;
            min_dist = dist(iterations);
        end

        % end condition
        if nodeDistance(n_a_reach,n_b_reach) <= max_step
            path_a = n_a_reach.path(Ta,Ta.isBackward);
            path_b = n_b_reach.path(Tb,Tb.isBackward);
            if Tb.isBackward
                path = [path_a, path_b];
            else
                path = [path_b, path_a];
            end
            break
        else
            [Ta, Tb] = Swap(Ta,Tb);
        end

        % estimation of remaining time displayed on the waitbar
        iterations = iterations + 1;
        time_it = time_it_old * 0.9 + 0.1 * toc;
        time_it_old = time_it;
        time_est = floor(time_it*(max_iteration-iterations));
        waitbar(iterations/max_iteration,bar,strjoin(["Ricerca in corso... (",iterations," iter,
",time_est," sec)"]));
    end

    % save the debug structure and close the waitbar
    debug = struct('history', dist, 'Ta', Ta, 'Tb', Tb, 'iterations', iterations);
    delete(bar);

% catch the error and stop the algorithm
catch exception
    delete(bar);
    msgbox(exception.message);
    rethrow(exception);
end
end
```

3.1 RandomConfig

Implementazione MATLAB della funzione `RandomConfig`:

```
function n_rand = RandomConfig(robot)
    config_rand = randomConfiguration(robot);
    n_rand = Node.config2node(config_rand,robot);
end
```

Dato il `robot` in ingresso `RandomConfig` tramite la funzione `randomConfiguration` genera una configurazione casuale (`config_rand`) che viene convertita nel corrispettivo nodo (`n_rand`) tramite la funzione `config2node`.

3.2 NearestNeighbor

Implementazione MATLAB della funzione `NearestNeighbor`:

```
function n_near = NearestNeighbor(T,n_target)
    node_array = T.node_array;

    % matrix composed of the distances between the nodes of the tree and n_target
    distance_matrix = cell2mat(arrayfun(@(node) node.nodeDistance(n_target), node_array,
'UniformOutput', false));

    [min_dist, n_index] = min(distance_matrix);

    n_near = node_array(n_index);
end
```

Dato in ingresso l'albero `T` e un nodo `n_target`, la funzione `NearestNeighbor` trova il nodo n_{near} appartenente all'albero più vicino al nodo in ingresso. Il nodo n_{near} viene trovato selezionando la distanza minima (`min_dist`) tra tutti i nodi appartenenti all'albero e il nodo in ingresso.

3.3 ConstraintExtend

Nella figura è riportato lo pseudocodice della funzione `ConstrainedExtend` presente negli articoli in bibliografia.

Algorithm 2: ConstrainedExtend(T, q_{near}, q_{target})

```
1  q_s ← q_near; q_s^old ← q_near;
2  while true do
3      if q_target = q_s then
4          return q_s;
5      else if |q_target − q_s| > |q_s^old − q_target| then
6          return q_s^old;
7      end
8      q_s^old ← q_s;
9      q_s ← q_s + min(Δq_step, |q_target − q_s|) (q_target − q_s)/|q_target − q_s|;
10     q_s ← ConstrainConfig(q_s^old, q_s);
11     if q_s ≠ NULL and CollisionFree(q_s^old, q_s) then
12         T.AddVertex(q_s);
13         T.AddEdge(q_s^old, q_s);
14     else
15         return q_s^old;
16     end
17 end
```

La funzione `ConstrainedExtend` opera muovendosi iterativamente da una configurazione q_{near} (n_{near} nell'implementazione) verso una configurazione q_{target} (n_{target}) con una dimensione del passo di Δq_{step} (`max_step`). Dopo ogni passo verso q_{target} (n_{target}), la funzione controlla se la nuova configurazione q_s (n_s) ha raggiunto q_{target} (n_{target}) o se non sta facendo progressi verso q_{target} (n_{target}), in entrambi i casi la funzione termina. Se le condizioni non sono vere, allora l'algoritmo fa un passo verso q_{target} (n_{target}) e passa la nuova q_s (n_s) alla funzione `ConstrainConfig` (nell'implementazione questa funzione viene omessa e n_s viene passato a `ProjectConfig`). Se `ProjectConfig` è in grado di proiettare n_s nel manifold e non è in collisione, il nuovo nodo n_s viene aggiunto all'albero e il processo viene ripetuto. Altrimenti, `ConstrainedExtend` termina. `ConstrainedExtend` restituisce sempre l'ultimo nodo raggiunto dall'operazione di estensione.

Segue l'implementazione MATLAB corrispettiva dello pseudocodice della funzione `ConstrainedExtend`.

```
function [T, n_reach] =
ConstraintExtend(T,n_near,n_target,TSR,check_self_collision,false_collision,robot,max_step,max_it
eration,eps)

iterations = 1;
ns = n_near;
ns_old = n_near;

while true
    % stop condition
    if n_target.q == ns.q
        n_reach = ns;
        break
    elseif (n_target.nodeDistance(ns) > n_target.nodeDistance(ns_old))
        n_reach = ns_old;
        break
    elseif iterations > max_iteration
        n_reach = ns;
        break
    end

    ns_old = ns;
    ns.parent_index = ns_old.node_index;
    ns.q = wrapToPi(ns.q + min(max_step, n_target.nodeDistance(ns)) * (n_target.q - ns.q) /
n_target.nodeDistance(ns));

    % TODO ns = ConstrainConfig(ns_old, ns);

    [ns, found] = ProjectConfig(ns_old,ns,TSR,robot,eps,max_step);

    % adding found node to the Tree
    if found
        % collision check
        if check_self_collision
            if ~ns.checkCol(robot,false_collision)
                [T,ns.node_index] = T.addNode(ns);
            end
        else
            [T,ns.node_index] = T.addNode(ns);
        end
    else

        n_reach = ns_old;
        break
    end

    iterations = iterations + 1;

    % added stop condition
    if ns.nodeDistance(ns_old) < 5*eps
        n_reach = ns_old;
        break
    end
end
end
```

Nell'implementazione è stata aggiunta una ulteriore condizione di stop sulla distanza tra n_s e n_s^{old} per ottimizzare i tempi di esecuzione dell'algoritmo.

3.3.1 ProjectConfig

Nella figura è riportato lo pseudocodice della funzione `ProjectConfig` presente negli articoli in bibliografia.

Algorithm 4: ProjectConfig(q_s^{old}, q_s, $\mathbf{C}$, $\mathbf{T}_c^0$)

1 **while** true **do**
2 $\quad \Delta\mathbf{x} \leftarrow$ DisplacementFromConstraint($\mathbf{C}$, $\mathbf{T}_c^0$, q_s);
3 $\quad$ **if** $\|\Delta \boldsymbol{x}\| < \epsilon$ **then** return q_s;
4 $\quad \mathbf{J} \leftarrow$ GetJacobian(q_s);
5 $\quad \Delta q_{error} \leftarrow \mathbf{J}^T(\mathbf{J}\mathbf{J}^T)^{-1}\Delta x$;
6 $\quad q_s \leftarrow (q_s - \Delta q_{error})$;
7 $\quad$ **if** $\left|q_s - q_s^{old}\right| > 2\Delta q_{step}$ ***or*** OutsideJointLimit(q_s) **then** return NULL;
8 **end**

Segue l'implementazione MATLAB corrispettiva dello pseudocodice della funzione `ProjectConfig`.

```
function [ns,found] = ProjectConfig(ns_old,ns,TSR,robot,eps,max_step)

    % used to calculate the geometric jacobian
    endeffector_name = robot.Bodies(end);
    endeffector_name = endeffector_name{1}.Name;

    while true

        T0_s = directKin(ns, robot);
        delta_x = displacement(TSR, T0_s);
        if norm(delta_x) < eps
            found = true;
            break
        end
        J = geometricJacobian(robot,node2config(ns,robot),endeffector_name);
        delta_qerr = pinv(J)*delta_x;
        ns.q = wrapToPi(ns.q - delta_qerr');
        if nodeDistance(ns,ns_old) > 2 * max_step
            ns=0;
            found = false;
            break
        end
    end

end
```

3.4 Swap

Implementazione MATLAB della funzione `Swap`:

```
function [Tb,Ta] = Swap(Ta,Tb)

end
```

Dati in ingresso gli alberi `Ta` e `Tb` la funzione `Swap` restituisce i due alberi scambiati.

3.5 Funzioni Aggiuntive

In questo paragrafo sono descritte le funzioni aggiunte rispetto allo pseudocodice di riferimento per effettuare controlli sull'algoritmo CBiRRT.

3.5.1 CheckInputParams

Implementazione MATLAB della funzione `CheckInputParams`:

```
function
CheckInputParams(n_start,n_goal,robot,TSR,check_self_collision,max_step,eps,max_iteration,false_c
ollision)

if check_self_collision
    if ~strcmp(robot.DataFormat,'row')
        error("If check collision is true robot. DataFormat must be 'row'.");
    end

    if n_start.checkCol(robot,false_collision)
        error("If check collision is true start configuration must be without collision.");
    end

    if n_goal.checkCol(robot,false_collision)
        error("If check collision is true goal configuration must be without collision.");
    end

end

end
```

La funzione `CheckInputParams` effettua tre controlli se `check_self_collision` è vera:

- controlla che il formato delle configurazioni del robot sia riga
- controlla che il robot nella configurazione del nodo n_{start} non riporti auto-collisioni
- controlla che il robot nella configurazione del nodo n_{goal} non riporti auto-collisioni

3.5.2 FindFalseCollision

Implementazione MATLAB della funzione `FindFalseCollision`:

```
function false_collision = FindFalseCollision(robot)
    [isColliding, sepDist] = robot.checkCollision(robot.homeConfiguration);
    if isColliding
    [b1,b2] = find(isnan(sepDist));
    false_collision = [b1, b2];
    else
    false_collision = [];
    end
end
```

La funzione `FindFalseCollision` restituisce nel vettore `false_collision` le coppie di giunti `[b1,b2]` che collidono nella configurazione base del robot in ingresso.

4 Implementazione Visualizzazione Risultati

In questo capitolo sono riportate le funzioni utili per la visualizzazione dei risultati ottenuti dall'algoritmo.

4.1 ShowPath

Implementazione MATLAB della funzione `ShowPath`:

```
function ShowPath(path,robot,new_fig)

path_T = cell2mat(arrayfun(@(node) tform2trvec(node.directKin(robot)), path,...
    'UniformOutput', false));
path_T = reshape(path_T,3,[]);

if nargin == 3
    if new_fig
        figure(),
    end
else
    figure(),
end

axis([-1 1 -1 1 -1 1]), view([135,8]);

plot3(path_T(1,:),path_T(2,:),path_T(3,:),'-.o','Color','b','MarkerSize',6,...
    'MarkerFaceColor','#D9FFFF'), grid on;

end
```

La funzione `ShowPath` mostra in un grafico tridimensionale il percorso ottenuto attraverso la cinematica diretta di ogni nodo appartenente al path.

4.2 AnimatePath

Implementazione MATLAB della funzione `AnimatePath`:

```
function AnimatePath(path,robot,duration)

figure("WindowState","maximized"), hold on,

subplot(2,2,1), title('test'),
robot.show(node2config(path(1),robot),'PreservePlot',false,'FastUpdate',false);

subplot(2,2,3)
robot.show(node2config(path(length(path)),robot),'PreservePlot',false,...
    'FastUpdate',false);

subplot(2,2,[2,4]), hold on,
ShowPath(path,robot,false),

if nargin == 2
    duration = 3;
end

step = duration/length(path);

for i=1:length(path)
    node = path(i);
    subplot(2,2,[2,4]), hold on;
    robot.show(node2config(node,robot),'PreservePlot',false,'FastUpdate',false);
    pause(step)

end
end
```

La funzione `AnimatePath` simula graficamente l'esecuzione del percorso in ingresso da parte del robot.

4.3 ShowTree

Implementazione MATLAB della funzione `ShowTree`:

```
function ShowTree(robot,Ta,Tb)

a_list = cell2mat(arrayfun(@(node) tform2trvec(node.directKin(robot)),...
    Ta.node_array, 'UniformOutput', false));
a_list = reshape(a_list,3,[]);

show(robot,Ta.node_array(1).node2config(robot)); hold on


plot3(a_list(1,1),a_list(2,1),a_list(3,1),'p','Color','r','MarkerSize',8,...
    'MarkerFaceColor','#d62828'); hold on, grid on;
plot3(a_list(1,2:end),a_list(2,2:end),a_list(3,2:end),'o','Color','y',...
    'MarkerSize',5,'MarkerFaceColor','#e9c46a'); hold on,

if nargin == 3
    b_list = cell2mat(arrayfun(@(node) tform2trvec(node.directKin(robot)),...
        Tb.node_array, 'UniformOutput', false));
    b_list = reshape(b_list,3,[]);

    show(robot,Tb.node_array(1).node2config(robot)); hold on

    plot3(b_list(1,1),b_list(2,1),b_list(3,1),'p','Color','r',...
        'MarkerSize',8,'MarkerFaceColor','#d62828'); hold on,
    plot3(b_list(1,2:end),b_list(2,2:end),b_list(3,2:end),'o','Color','b',...
        'MarkerSize',5,'MarkerFaceColor','#184e77');
end
end
```

La funzione `ShowTree` mostra in un grafico tridimensionale tutti i nodi nello spazio di lavoro appartenenti agli alberi del nodo di start e del nodo di goal indicandoli con colori diversi.

5 Utilizzo, Risultati e Conclusioni

In questo capitolo è mostrato un esempio di utilizzo del codice su un caso di studio implementato con i relativi risultati. Successivamente saranno discusse le conclusioni e i possibili lavori futuri relativi al progetto.

5.1 example.m

Di seguito è riportato il codice presente nel file `example.m` relativo al caso di studio.

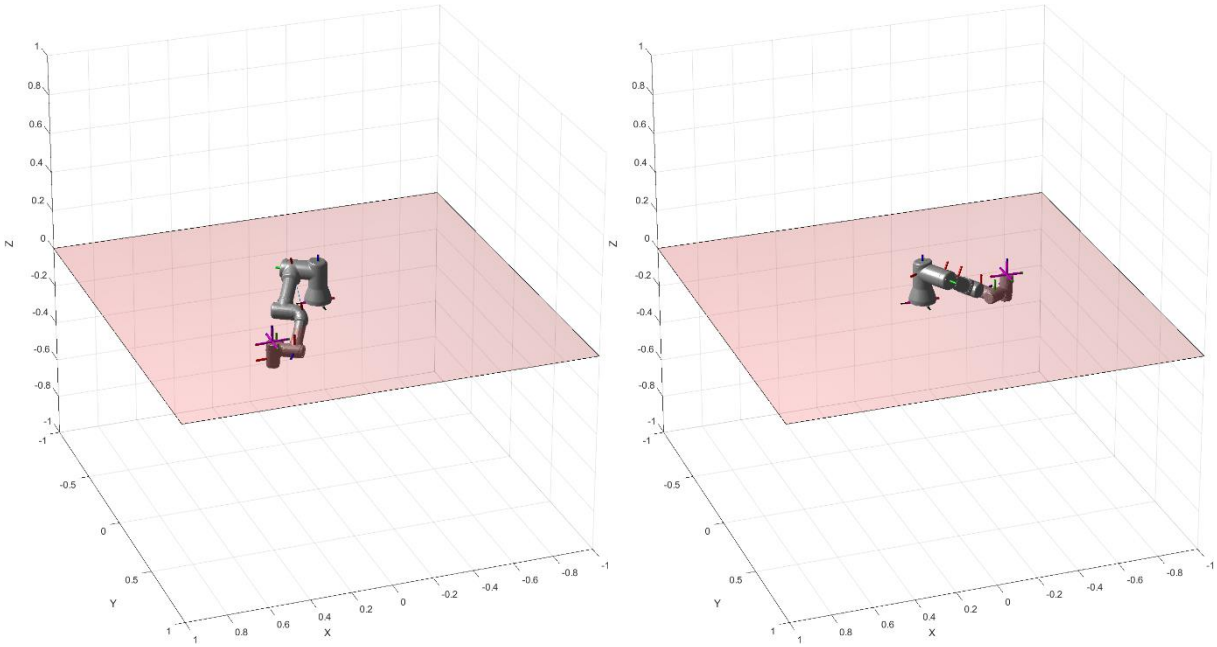
```
clc, clear all, close all;

robot = loadrobot("universalUR3");

robot.DataFormat = 'row';

n_start = Node.tform2node(robot,trvec2tform([0.366,0.366,0]),[0.1, 0.1, 0.1, 1, 1, 1000]);
n_start.directKin(robot)
n_goal = Node.tform2node(robot,trvec2tform([-0.5,-0.143,0]), [0.1, 0.1, 0.1, 1, 1, 1000]);
n_goal.directKin(robot)

% Parameter Definition

MAX_ITERATION = 200;
MAX_STEP = 0.1;
eps = 0.01;
CHECK_SELF_COLLISION = false;

% TSR Definition

Bw = [-Inf, Inf; -Inf, Inf; -0.027, 0.027; -Inf, Inf; -Inf, Inf; -Inf, Inf];
T0_w = trvec2tform([0, 0, 0]);
angle_z = 0;
Tw_e = [cos(angle_z)    sin(angle_z)    0       0
        -sin(angle_z)   cos(angle_z)    0       0
        0                   0           1       0
        0                   0           0       1];

tsr = TSR(T0_w,Tw_e,Bw);

%% CBiRRT Algorithm

[path, debug] = CBiRRT(n_start,n_goal,robot,tsr,CHECK_SELF_COLLISION,MAX_STEP,eps,MAX_ITERATION);

%% Data Visualization

AnimatePath(path,robot);
ShowTree(robot,debug.Ta,debug.Tb);

%% Distance History Visualization

figure(2)
plot(1:MAX_ITERATION,debug.history);
```

Nella seguente tabella sono mostrati i parametri di input configurabili per eseguire l'algoritmo CBiRRT:

PARAMETRO	TIPO DI DATO	DESCRIZIONE
`n_start`	Node	nodo di start.
`n_goal`	Node	nodo di goal.
`robot`	rigidBodyTree	modello del robot.
`TSR`	TSR	Task Space Region.
`check_self_collision`	bool	abilita il controllo dell'auto-collisione.
`max_step`	float	massimo passo di aggiornamento delle configurazioni.
`eps`	float	tolleranza rispetto ai vincoli del TSR.
`max_iteration`	int	condizione di stop.

i parametri di input configurabili per eseguire l'algoritmo CBiRRT:

Nel caso di studio analizzato è stato utilizzato il robot manipolatore universalUR3 (6 DOF).
Il braccio del robot a partire dalla configurazione iniziale deve riuscire a raggiungere la configurazione di goal in modo che l'end-effector resti vincolato (a meno di una tolleranza) sul piano z=0 (in rosso).



Le configurazioni iniziali e finali del robot sono calcolate tramite la cinematica inversa data la posa dell'end effector iniziale e finale.
Nella figura a sinistra è mostrata la configurazione iniziale del robot:

$$q_{start} = [-2.126, 2.610, 0, -1.069, -1.566, -2.585]$$

Nella figura a destra è mostrata la configurazione finale del robot:

$$q_{goal} = [-3.093, 0.508, 0, -1.072, -1.566, 1.062]$$

I vincoli all'end-effector sono definiti attraverso le proprietà del TSR:

$$T_w^0 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, T_e^w = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, B_w = \begin{bmatrix} -\infty & \infty \\ -\infty & \infty \\ -0.027 & 0.027 \\ -\infty & \infty \\ -\infty & \infty \\ -\infty & \infty \end{bmatrix}$$

La matrice T_w^0 rappresenta la trasformazione dal sistema di riferimento base al centro del piano, nel caso in questione è stata scelta una matrice identità.
La matrice T_e^w rappresenta l'offset tra il sistema di riferimento del piano e l'end-effector, nel caso in questione corrisponde a una matrice identità.

La matrice B_w contiene i vincoli degli assi e degli angoli che l'end-effector del robot deve rispettare. Nel caso analizzato l'unico vincolo imposto è sull'asse z affinché l'end-effector resti vincolato al piano con una tolleranza di $\pm 0.027\ m$.

I parametri relativi all'algoritmo sono impostati come segue:

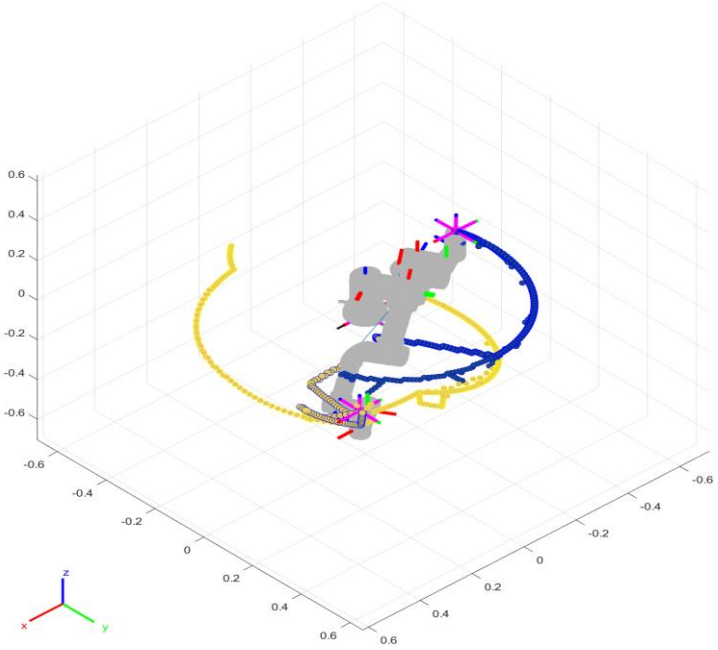
- `max_iteration = 200`
- `max_step = 0.1`
- `eps = 0.01`
- `check_self_collision = false`

Un numero massimo di iterazioni pari a 200 è sufficiente nel caso in questione per completare l'esplorazione. I valori di `max_step` ed `eps` sono specifici del problema. Nel caso analizzato non si è interessati al controllo dell'auto-collisione del robot.

5.2 Risultati

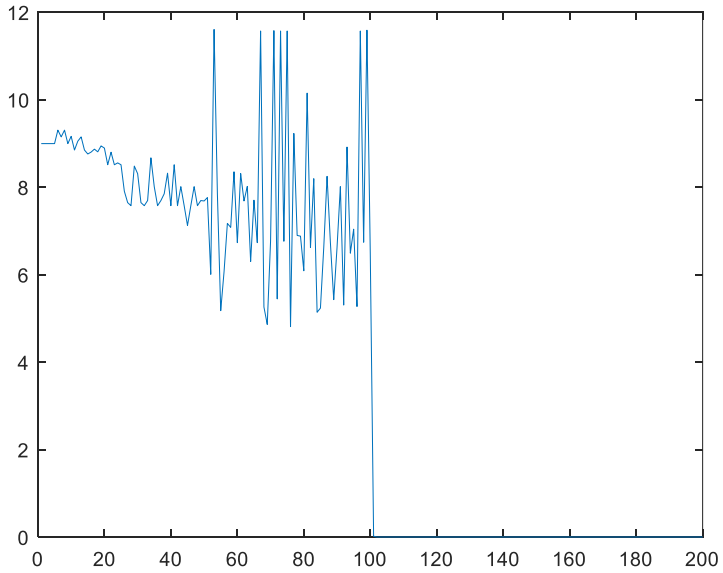
L'utilizzo del pianificatore di traiettoria basato sul campionamento casuale rende i risultati dipendenti dalle singole esecuzioni dell'algoritmo CBiRRT, seppure i suoi parametri di ingresso siano gli stessi.

Nella figura seguente è mostrato il robot nella configurazione iniziale e finale, il percorso ottenuto e i due alberi creati a partire dal nodo di start e di goal.



L'algoritmo implementato si occupa esclusivamente di ricercare un percorso valido senza effettuare alcun tipo di ottimizzazione.

Nella figura seguente è mostrato l'andamento della distanza tra i nodi n^a_{reach} e n^b_{reach} (indice della distanza minima tra gli alberi) in funzione del numero di iterazioni. Se la distanza raggiunge il valore 0 gli alberi T_a e T_b si connettono e l'algoritmo CBiRRT restituisce il percorso, altrimenti continua fino alla condizione di stop.



In questa specifica esecuzione il percorso è stato trovato dopo circa 100 iterazioni in circa 10 secondi.
Il numero di iterazioni e il tempo necessario per trovare il percorso dipendono in maniera considerevole dalle singole esecuzioni dell'algoritmo del CBiRRT.

È stata effettuata un'analisi parametrica sull'efficienza dell'algoritmo valutando il numero medio di iterazioni e il tempo medio impiegati dall'algoritmo per restituire il percorso. Di seguito è riportata la tabella con i risultati ottenuti effettuando 10 prove per ogni caso parametrico.

`max_step`	`eps`	n° medio di iterazioni	tempo medio $[sec]$	successo [%]
0.1	0.01	12	6.22	100
0.1	0.005	17	7.59	100
0.05	0.01	62	27.82	100
0.025	0.005	-	-	0

Quando i parametri `max_step` e `eps` diventano troppo stringenti l'algoritmo non riesce a trovare il percorso entro le condizioni di stop.

5.3 Conclusioni e Lavori Futuri

È stato presentato l'algoritmo CBiRRT per la pianificazione di percorsi nello spazio di configurazione rispettando i vincoli, combinando l'esplorazione dello spazio di configurazione con metodi di proiezione verso il manifold.
Il vantaggio principale di questo algoritmo è la generalità nella rappresentazione dei vincoli. L'algoritmo è in grado di affrontare una vasta gamma di problemi senza ricorrere a tecniche altamente specializzate. Questo è evidenziato in parte dal basso numero di parametri in ingresso e dal fatto che, nonostante la vasta gamma di robot compatibili, i valori dei parametri restano gli stessi.
In conclusione, l'obiettivo di questo progetto è stato raggiunto, implementando su MATLAB l'algoritmo CBiRRT. L'implementazione ha riguardato la definizione delle classi Node, Tree e TSR e ha seguito il flusso dello pseudocodice presente negli articoli presenti in bibliografia. L'algoritmo implementato non sempre riesce a trovare nelle condizioni di stop un percorso valido soprattutto se sono imposti vincoli o parametri troppo stringenti. In tal caso l'algoritmo mostrerà comunque il percorso migliore, composto dai rami degli alberi aventi i nodi più vicini connessi da un salto lineare.

Come lavori futuri per migliorare l'algoritmo è consigliato:

- Migliorare l'efficienza dell'algoritmo.
- Implementare la funzione `ConstraintConfig` e la catena TSR per l'implementazione di più TSR.
- Implementare la funzione `AddRoot` per utilizzare il TSR anche nelle configurazioni di start.
- Implementare la funzione `SmoothPath` per ottimizzare il percorso trovato.

Bibliografia

[1] Berenson, D., Srinivasa, S. and Kuffner, J. (2011) 'Task Space Regions: A framework for pose-constrained manipulation planning', The International Journal of Robotics Research, 30(12), pp. 1435–1460. doi: 10.1177/0278364910396389.

[2] Berenson, Dmitry & Srinivasa, Siddhartha & Ferguson, Dave & Kuffner, James. (2009). Manipulation planning on constraint manifolds. 625-632. 10.1109/ROBOT.2009.5152399.