Modellazione e Sviluppo dei task PickPlace e Trajectory Tracking su Arduino Tinkerkit Braccio

Laboratorio di Automatica

Angelo D'Amante
angelo.damante@stud.unifi.it

Lorenzo Falai
lorenzo.falai@stud.unifi.it

A.A. 2021/2022

Sommario

In questo lavoro presentiamo un manipolatore antropomorfo a 5 gradi di libertà (DOF) per eseguire due semplici task: *Pick&Place* e *Trajectory Tracking*. Partendo dalla simulazione e giungendo all'effettiva implementazione hardware usando la piattaforma *Arduino Tinkerkit Braccio* è stata progettata la cinematica diretta e inversa usando Matlab e Simulink. Tutto il progetto si trova nella repository su Github [5].

Indice

Capitolo 1

Introduzione

Un robot **manipolatore antropomorfo** è un sistema meccanico in grado di muovere un oggetto afferrandolo in genere con una pinza, posta alla sua estremità, detta End-Effector (EE). Il sistema è composto da corpi rigidi, detti **link**, e articolazioni, dette **giunti**; ogni giunto che compone il robot, fino all'EE escluso, fornisce al robot un grado di libertà in più.

Obiettivo

L'obiettivo di questo elaborato è far eseguire ad un robot antropomorfo con 5 gradi di libertà (degree of freedom - DOF) due task:

- **Trajectory Tracking**: Seguire una traiettoria, fornendo i vari waypoint di interesse.
- **Pick&Place**: Molto utile nella produzione industriale, consiste nello spostare un oggetto da un punto ad un altro nella maniera più rapida e automatica possibile.

Workflow

Il workflow seguito consiste di tre fasi:

1. **Modellazione**: Descrizione del modello del robot a 5 DOF con il metodo di Denavit Hartenberg (DH) e definizione del workspace del modello.
2. **Simulazione**: Far eseguire i due compiti prima descritti in un ambiente simulato per verificare la correttezza dei sistemi e delle soluzioni ideate. Ciò è stato possibile usando `VR toolbox` e `Robotics Toolbox` di Simulink.
3. **Implementazione**: Scrivere l'effettiva soluzione sulla piattaforma hardware. Questa fase presenta il pericolo che i vari test di soluzione possano danneggiare in modo permanente il robot; per evitare ciò, sono stati utilizzati sistemi di `unit test` e `rescue model`.

Risultati

Infine, vengono analizzati e confrontati i risultati ottenuti nella fase di **Simulazione** con quelli di **Implementazione**, per garantirne la correttezza e l'affidabilità nell'esecuzione dei task.

Hotspot del Progetto

Le parti più critiche, ovvero i *"punti caldi"* del progetto, si sono rivelate essere le seguenti:

- **KIN e IKIN**: Scrivere al livello hardware il modulo per la cinematica diretta (KIN) e inversa (IKIN). L'idea è quella di fornire una soluzione in forma chiusa e modulare per qualsiasi robot antropomorfo dotato di n DOF partendo semplicemente dai parametri di Denavit Hartenberg.
- **Stateflow e Tracking**: Determinare il **tempo di campionamento** per gestire la scansione dei vari waypoint e mandare i comandi ai giunti per il task *Trajectory Tracking*. L'uso di una chart con **stateflow** in questa fase è stato provvidenziale.

Layout Directories

In [5] si trova la repository con tutto il progetto, di cui riportiamo le directory:

```
|-- arm-manipulator-5dof            # Progetto
    |-- +classes                    # Variabili Object System
    |-- +functions                  # Script Modulari
    |-- +unittests                  # Unit Test Preliminari
    |-- data                        # Oggetti e Contenitori
    |-- docs                        # Documentazioni Usate
    |-- graphics                    # File .x3d
    |-- Tinkerkit_model             # File STL
    |-- models                      # File Simulink
    |   |-- simulation                # Fase di Simulazioni
    |   |-- hardware                  # Fase di Implementazione HW
    |   |-- kinematics                # Implementazione delle cinematiche
    |   |-- tasks                     # Modellazione dei Task
```

Per poter avviare i modelli per la simulazione si deve avviare preventivamente lo script `initSim.m` locato nella directory principale. Per i modelli che girano sull'hardware, invece, si avvia preventivamente lo script `initHW.m` locato nella stessa directory.

Capitolo 2

Modellazione

La prima fase è quella di modellazione del robot su cui si basa il progetto, così da ottenere la tabella dei parametri di Denavit Hartenberg.

2.1 Manipolatore Tinkerkit Braccio

Abbiamo scelto il **Robot Antropomorfo Tinkerkit Braccio a 5 DOF** prodotto da Arduino [1], mostrato in figura 2.1

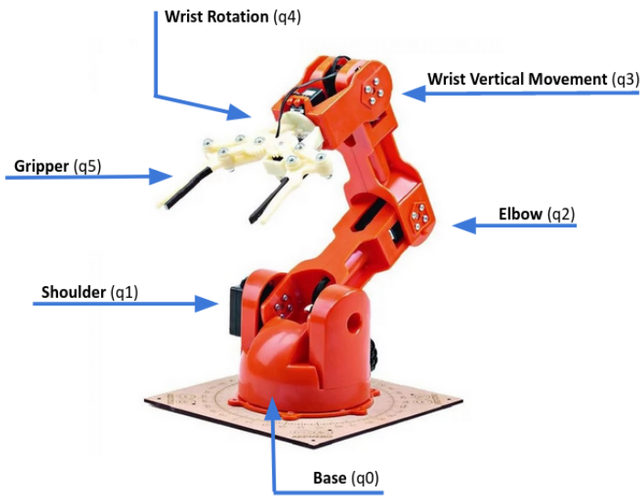

Figura 2.1: Tinkerkit Braccio

2.1.1 Hardware

Il robot è dotato di 6 giunti rotoidali, ma solo 5 concorrono al conteggio dei gradi di libertà; infatti, il giunto che si occupa dell'apertura della pinza non rappresenta uno di questi. [16].

Servomotori

In tutte le articolazioni dalla **base** al **polso** sono usati servi di tipo **SR 431** con riduttore in metallo. La velocità di rotazione è di circa $0.2s/60°$, la coppia di stallo è di circa $13kg{\cdot}cm$, l'angolo di rotazione massima è di 180° e con una larghezza d'impulso di $1500\mu s$ si ottiene la posizione intermedia, le dimensioni sono $42 \times 20.5 \times 39.5\ mm$ di altezza e il peso è di $62\ g$. [16]

Nella **pinza** e nella rotazione del **polso** sono stati usati motori di tipo **SR 311** con riduttore in metallo. La velocità di rotazione senza carichi è di circa $0.13s/60°$, la coppia di stallo è di circa $3.5kg \cdot cm$, anche in questo caso, l'angolo di rotazione massima è di 180° con la stessa larghezza d'impulso del SR 431, le dimensioni sono $31.3 \times 16.5 \times 28.6\ mm$ di altezza, il peso è 27 g. [16]

Notiamo che, per come è montato il Braccio, la graduazione angolare incisa dal costruttore sul supporto di base originale è antioraria, mentre la rotazione della base per angoli crescenti risulta oraria.

Shield

Lo shield del robot in figura 2.2 fornisce l'alimentazione di potenza che Arduino non sarebbe in grado di fornire. I connettori da M1 a M6 sono destinati alla connessione dei servo a cui forniscono l'alimentazione e i segnali di controllo PWM.

Ogni connessione dispone di un fusibile auto-ripristinante di protezione:

- **M1 a M4** vengono limitati a 1.1 A.
- **M5 e M6** vengono limitati a 750 mA.

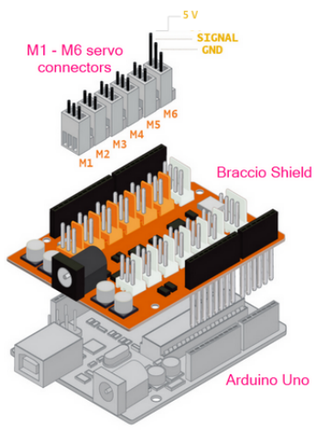

Figura 2.2: Shield

Le connessioni sono riportate nella tabella seguente [15],

Utilizzo	Pin Arduino	Connettore
base	11	M1
shoulder	10	M2
elbow	9	M3
pitch	6	M4
wrist	5	M5
grip	3	M6
voltage	12	-

Limiti fisici dei Servo

É importante notare che, per quanto i servo descritti in precedenza possano tranquillamente arrivare ai 180°, i limiti imposti dalla struttura [2] risultano i seguenti:

- Il motore **M1** della base ha un range $[15°, 65°]$;

- I motori **da M2 a M5** hanno un range $[0°, 180°]$;
- Il motore **M6** della pinza ha un range $[10°, 70°]$.

2.1.2 Hello Braccio

La libreria `Braccio`, fornita da Arduino [2] dà la possibilità di testare con una demo il corretto funzionamento del robot, nonchè il primo approccio con esso. La demo chiamata `simpleMovements` in [15] consiste di due semplici comandi

```
1  void loop(){
2      //(step delay, M1, M2, M3, M4, M5, M6);
3      Braccio.ServoMovement(20, 0, 15, 180, 170, 0, 73);
4      Braccio.ServoMovement(20, 180, 165, 0, 0, 180, 10);
5  }
```

Il nostro obiettivo però è quello di interfacciare il robot con Simulink usando le connessioni della Shield. Quindi, in riferimento alla repository della libreria in [4], si può notare come il funzionamento del metodo `ServoMovement()` consiste nel raggiungere l'angolo desiderato fornito in input incrementando l'attuale angolo di ogni motore di un grado per volta, fino al completo raggiungimento dell'input fornito.

A tale scopo abbiamo scritto in Matlab la funzione `nextStep` come segue:

```
1  function v = nextStep(current, desired)
2      if (current < desired)
3          current = current + 1;
4      end
5
6      if (current > desired)
7          current = current - 1;
8      end
9
10     v = current;
11 end
```

Questa funzione viene usata in una chart per simulare il `loop()` di sistema di Arduino. Quindi, l'esempio `simpleMovement` è del tutto equivalente al modello in figura 2.3. Nel dettaglio, la chart è riportata in figura 2.4.

Possiamo notare l'uso della libreria di Arduino in Simulink, in particolare:

- Standard Servo Write: Per scrivere valori (angoli interi positivi da 0 a 180) nei servo SR 311 e SR 431.
- Standard Servo Read: Per leggere valori dai PIN (PWM) connessi ai servo.
- Digital Output: Per collegare l'alimentazione della Shield.

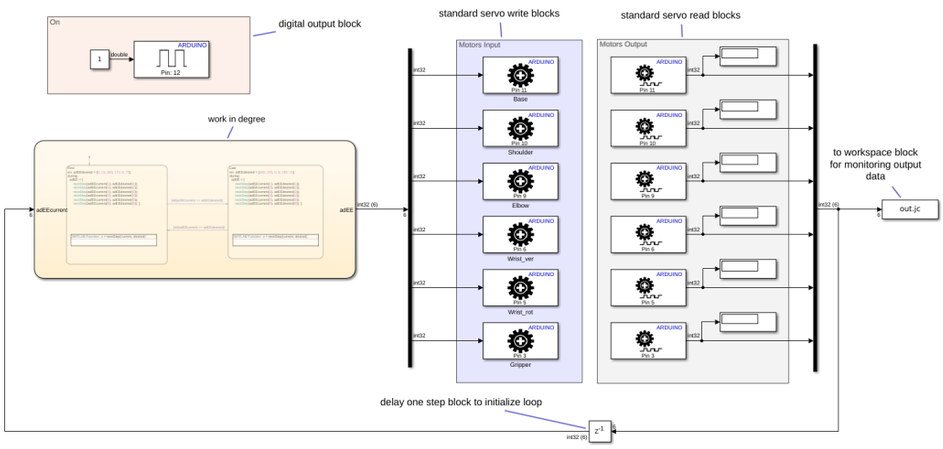

Figura 2.3: mHelloBraccio.slx

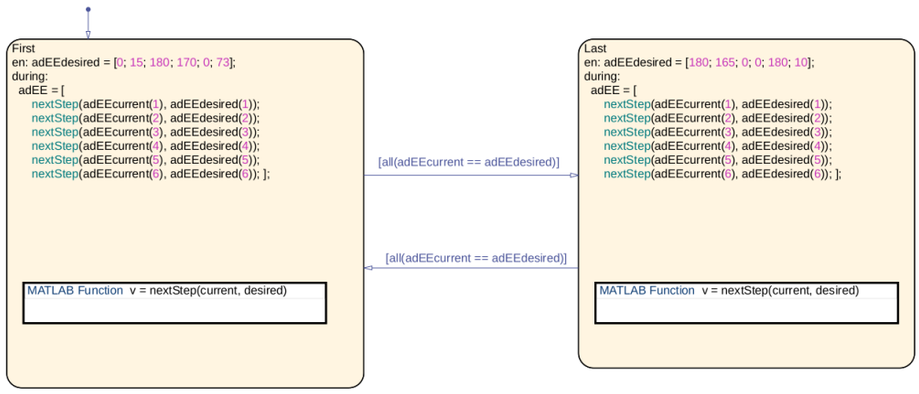

Figura 2.4: chart presente nel modello mHelloBraccio.slx

2.2 Workspace dell'Arduino Braccio

Il manipolatore presenta le seguenti caratteristiche:

- Altezza (max): 52cm;
- Larghezza della base: 14cm;
- Ampiezza pinza: 90mm;
- Peso: 792g.

È fondamentale tracciare lo spazio di lavoro del robot, così da tener conto solo dei punti raggiungibili e salvaguardare le funzioni dalle singolarità. Generalmente un manipolatore seriale a catena aperta con la quasi totalità dei giunti con asse di rotazione paralleli tra loro, presenta un **workspace sferico**. Dobbiamo però tener conto di alcuni aspetti:

- Il corpo che contiene il giunto della base non concorre al calcolo dello spazio di lavoro.
- Fissando la base ad un piano di appoggio, si è scelto per semplicità di test di non poter andare al di sotto di tale superficie; così facendo, si tiene in considerazione solo la semisfera superiore.

- Il limite fisico imposto al giunto della base ($[15°, 65°]$) fa si che i punti lungo la verticale del robot siano irrangiungibili, se non piegandosi completamente. Questo porta ad escludere un cilindro di raggio 100 mm e altezza 220 mm.

Quindi il workspace (figura 2.5) risulta una semisfera di raggio 400 mm con un'insenatura cilindrica.

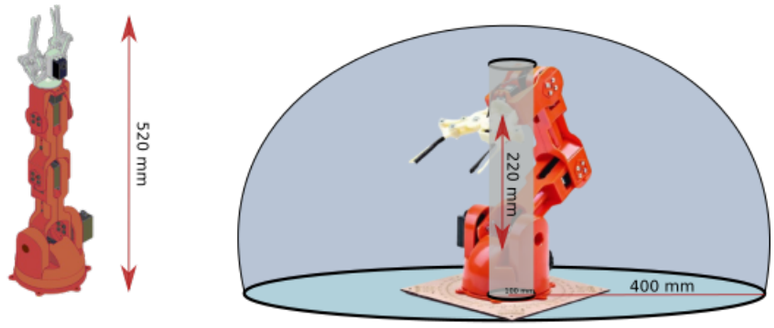

Figura 2.5: Workspace

makeWS.m

La definizione del workspace ha un'elevata importanza in fase di testing. Infatti, lo script `makeWS.m` [1] si occupa di generare tanti punti casuali (figura 2.6) che riempiano la semisfera [2] definita dall'utente. Lo scopo è quello di far funzionare tutto il progetto per ogni punto generato da questo script, ovvero non si deve presentare alcun fallimento in tutto il workspace.

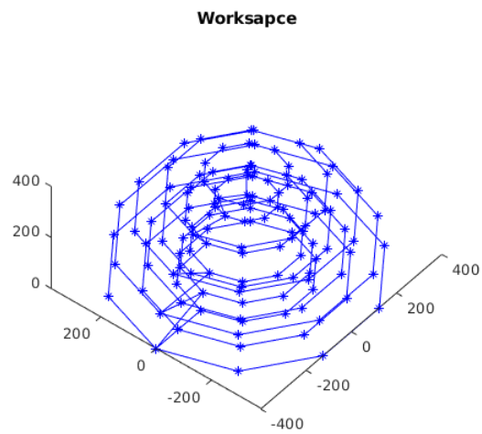

Figura 2.6: Punti generati dallo script

2.3 Mappatura Cinematica Diretta

Seguendo il metodo sistematico di Denavit Hartenberg per manipolatori con catena cinematica aperta [14], scegliamo di enumerare i link e i giunti come mostrato in figura 2.7. La matrice di trasformazione omogenea tra il giunto i e il giunto $i-1$ è

[1] Codice presente nel github del progetto [5].
[2] Per semplicità la chiameremo semisfera, anche se presenta l'insenatura cilindrica.

una composizione di due rotazioni, una intorno all'asse z e una intorno all'asse x, con le relative traslazioni.

$$T_{i+1}^{i}(\vartheta) = \begin{bmatrix} \cos\vartheta_i & -\sin\vartheta_i\cos\alpha_i & \sin\vartheta_i\sin\alpha_i & a_i\cos\vartheta_i \\ \sin\vartheta_i & \cos\vartheta_i\cos\alpha_i & -\cos\vartheta_i\sin\alpha_i & a_i\sin\vartheta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Per scrivere la matrice di trasformazione omogenea eseguiamo le rotazioni in terna corrente, ottenendo

$$T_5^0 = T_1^0 \cdot T_2^1 \cdot T_3^2 \cdot T_4^3 \cdot T_5^4.$$

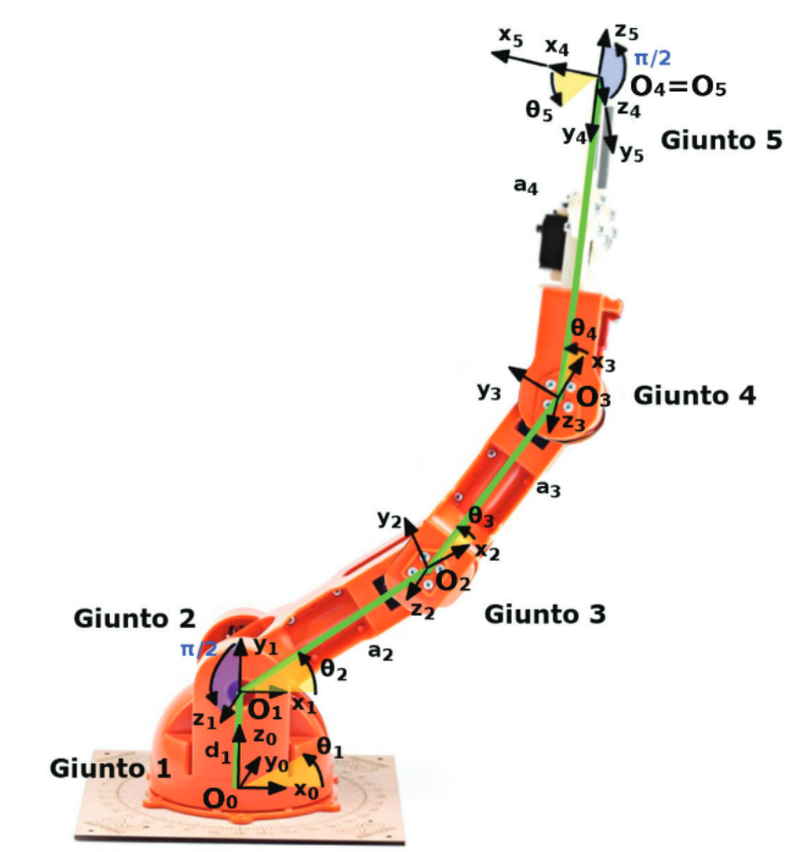

Figura 2.7: Cinematica Diretta del Tinkerkit Braccio

Otteniamo la seguente tabella dei parametri di DH [16]:

Link	a_i	α_i	d_i	ϑ_i
1-Base	0	$\frac{\pi}{2}$	d_1	ϑ_1
2-Forearm	a_2	0	0	ϑ_2
3-Arm	a_3	0	0	ϑ_3
4-Hand	a_4	0	0	ϑ_4
5-EE	0	$\frac{\pi}{2}$	0	ϑ_5

.

Capitolo 3

Trajectory Tracking

Il task di **Trajectory Tracking** consiste nel far seguire una determinata traiettoria alla mano del manipolatore.

3.1 Robotics Toolbox

Il principale strumento utilizzato è il `Matlab Robotics Toolbox`, che comprende funzioni per il calcolo della cinematica diretta ed inversa dei manipolatori e permette la visualizzazione di modelli di robot. Per importare un modello del **Braccio** ci sono due opzioni:

- Creare un modello usando `Simscape Multibody`, che è un ambiente di sviluppo 3D per parti meccaniche;
- Usare un file **URDF**.

In questo caso è stato scelto di usare la seconda opzione. Un file `URDF` (Unified Robotic Description Format) non è altro che un file `XML`, più spesso usato in `ROS`, che contiene una descrizione completa del robot. In particolare comprende:

- una descrizione geometrica di ogni link;
- una descrizione di ogni giunto (tipologia, origine della terna, ecc...);
- quali link sono connessi tra di loro e attraverso quali giunti;
- vincoli fisici di ogni giunto;
- può richiamare file *.stl* per la visualizzazione 3D del modello.

Vediamo un estratto del codice del file per capire meglio:

```
1       <link name="link1">
2      <visual>
3        <geometry>
4          <mesh filename="./base.stl"/>
5        </geometry>
6        <material name="orange"/>
7      </visual>
8    </link>
9
```

```
10      <joint name="base" type="revolute">
11      <axis xyz="0 0 -1"/>
12      <limit effort="1000.0" lower="0.0"
13          upper="3.141592653589793" velocity="4.0"/>
14      <origin rpy="0 0 0" xyz="0 0 0"/>
15      <parent link="base_link"/>
16      <child link="link1"/>
17    </joint>
18
19     <joint name="shoulder" type="revolute">
20      <axis xyz="1 0 0"/>
21      <limit effort="1000.0" lower="0.2617993877991494"
22          upper="2.8797932657906435" velocity="4.0"/>
23      <origin rpy="-1.5707963267948966 0 0" xyz="0 0 0.0505"/>
24      <parent link="link1"/>
25      <child link="link2"/>
26    </joint>
```

Nella prima sezione del codice viene definito il primo link, che è la base dell'**Arduino Braccio**, e vi viene collegato il relativo file *.stl*. Nelle due sezioni successive si individuano quali giunti sono connessi a questo link e se ne definisce il tipo (in questo caso abbiamo due giunti di rotazione), i limiti fisici e le origini delle terne di giunto.

Quello che si ottiene importando un modello *.urdf* in Matlab è il modello 3D mostrato in figura 3.1, visualizzabile attraverso il comando $show(robot, config)$.

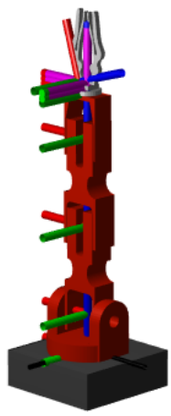

Figura 3.1: Modello di Arduino Braccio

Nel caso del manipolatore **Arduino Braccio** non tutti i giunti servono per la risoluzione del problema cinematico inverso; infatti gli ultimi due giunti non cambiano la posizione dell'end effector, ma solo l'orientazione. Per questo motivo i giunti di rotazione del polso e di comando della mano sono stati rimossi ed è stato aggiunto un'end effector virtuale al modello, collegato alla base della mano e che punta nel centro della stessa; quello che si ottiene è il modello di figura 3.2.

In questo modo possiamo risolvere il problema cinematico inverso sulla base della posizione dell'end effector virtuale e determinare le configurazioni dei primi quattro giunti del robot.

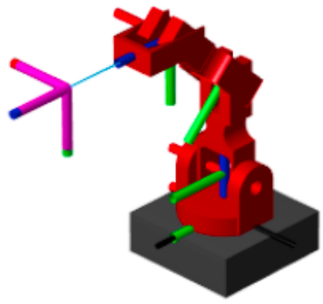

Figura 3.2: Modello di Arduino Braccio con End Effector Virtuale

3.2 Simulazione Matlab

3.2.1 Generazione della Traiettoria

Il primo fondamentale passo nell'inseguimento di una traiettoria è la generazione della traiettoria stessa. A tale scopo sono stati seguiti i seguenti passaggi:

- Si definisce un insieme di waypoints, ossia di punti dai quali il manipolatore deve passare;
- A partire dai waypoints, si genera una traiettoria che li attraversa.

Per la definizione dei waypoints sono stati usati due criteri diversi:

1. Un insieme di waypoints di tipo **Pick&Place**

```
wayPoints=[punto_iniziale; punto_palla;
           punto_iniziale; punto_scatola;
           punto_iniziale];
```

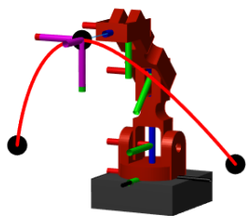

Figura 3.3: Traiettoria di Tipo **Pick&Place**

2. Un insieme di waypoints generati casualmente nel workspace

```
wayPoints=[punto_iniziale; ...
    (rand*0.15)+0.05 ((rand*4)-2)/10 (rand*0.35)+0.05; ...
    (rand*0.15)+0.05 ((rand*4)-2)/10 (rand*0.35)+0.05; ...
    (rand*0.15)+0.05 ((rand*4)-2)/10 (rand*0.35)+0.05; ...
    (rand*0.15)+0.05 ((rand*4)-2)/10 (rand*0.35)+0.05; ...
    punto_iniziale];
```

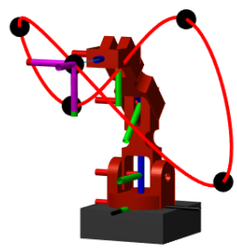

Figura 3.4: Esempio di Traiettoria Generata Casualmente

Una volta definiti i waypoints, la generazione della traiettoria è immediata grazie ad un comando del `Curve Fitting Toolbox` di Matlab

```
traj=cscvn(wayPoints');
fnplt(traj,'r',2);
```

Il comando `cscvn` esegue un'interpolazione dei punti usando spline cubiche e ritorna una struttura che rappresenta la traiettoria. Il comando `fnplt` ne permette invece la visualizzazione. L'ultimo passo è il campionamento della traiettoria: vogliamo passare da una funzione continua ad un numero finito di punti, che sono quelli da cui faremo effettivamente passare l'end effector. All'aumentare del numero di punti migliora l'approssimazione della traiettoria, ma aumenta la complessità di calcolo: noi abbiamo scelto di usare 20 punti per ogni waypoint.

```
[n,~]=size(wayPoints);
totalPoints=n*20;
x=linspace(0,traj.breaks(end),totalPoints);
eePos=ppval(traj,x);
```

Questo ci permette di scegliere un numero finito di punti equispaziati sulla traiettoria.

3.2.2 Cinematica Inversa

Per ognuno dei punti selezionati sulla traiettoria da inseguire si deve risolvere il problema cinematico inverso. Per fare ciò possiamo ancora utilizzare il `Robotics Toolbox` e impostare un problema ricorsivo a partire da una condizione iniziale dei giunti:

```
config = homeConfiguration(robot);
ik = robotics.InverseKinematics('RigidBodyTree',robot);
ik.SolverAlgorithm = 'LevenbergMarquardt';
weights = [0 0 0 1 1 1];
initialguess = config;
for idx = 1:size(eePos,2)
    tform = trvec2tform(eePos(:,idx)');
    configSoln(idx,:) = ik('end_effector',tform,weights,initialguess);
    initialguess = configSoln(idx,:);
end
```

Ad ogni iterazione si risolve il problema cinematico inverso per il punto selezionato utilizzando come condizione iniziale la soluzione determinata all'iterazione precedente.

Quello che si ottiene in uscita è la matrice `ConfigSoln`: ogni riga rappresenta la configurazione dei giunti necessaria per raggiungere uno dei punti della traiettoria. Visualizzando come configurazione del robot una riga alla volta, mantenendo fissati gli ultimi due giunti che in questa applicazione non vengono utilizzati, possiamo osservare l'inseguimento della traiettoria da parte del manipolatore

```
title('Waypoints Tracking')
config2(5).JointPosition=0;
config2(6).JointPosition=73*pi/180;
for idx = 1:size(eePos,2)
    config2(1).JointPosition=configSoln(idx,1).JointPosition;
    config2(2).JointPosition=configSoln(idx,2).JointPosition;
    config2(3).JointPosition=configSoln(idx,3).JointPosition;
    config2(4).JointPosition=configSoln(idx,4).JointPosition;
    show(robot2,config2, 'PreservePlot', false,'Frames','off');
    hold on
    if idx==1
        fnplt(traj,'r',2);
        plot3(wayPoints(:,1),wayPoints(:,2),wayPoints(:,3),'.','MarkerSize',
            40, 'MarkerEdgeColor','k');
    end
    pause(0.1)
end
```

3.3 Simulazione Simulink

Prima di procedere all'implementazione, il task è stato testato in simulazione con Simulink e la cinematica è stata simulata usando i blocchi **Get Transform** e **Inverse Kinematic** forniti dal *Robotics Toolbox* [9]. Inoltre, per vedere il risultato a video, abbiamo utilizzato il blocco **VR Sink** fornito dal toolbox *Simulink 3D Animation* [10].

Il modello Simulink completo è riportato in figura 3.5 e notiamo che in ingresso alla simulazione viene fornito il vettore di waypoint chiamato `adWaypoints` $\in \mathbb{R}^{3\times 100}$, costituito da 100 punti. I vari waypoint vengono processati dalla chart mostrata in figura 3.6, che scandisce un waypoint per volta valutandone la vicinanza con l'attuale posizione dell'end-effector con una certa tolleranza.

Per ogni waypoint, si calcola il vettore di coordinate dei giunti con la cinematica inversa e la configurazione risultante va in ingresso alla cinematica diretta per calcolare la coordinata dell'end-effector, da dare in pasto al visualizzatore grafico.

Si nota immediatamente che in linea teorica non occorrerebbe il passaggio dalla cinematica diretta per ottenere la simulazione desiderata. Infatti, le coordinate dell'end-effector calcolate dal blocco sarebbero riportate in `adWaypoints`. Bisogna però tener conto della tolleranza: infatti, il risolutore della cinematica inversa non riporta un risultato in forma chiusa, ma risolve un problema di ottimizzazione che dipende dai parametri configurati. Questo comportamento non deterministico giustifica quindi l'utilizzo del blocco di cinematica diretta.

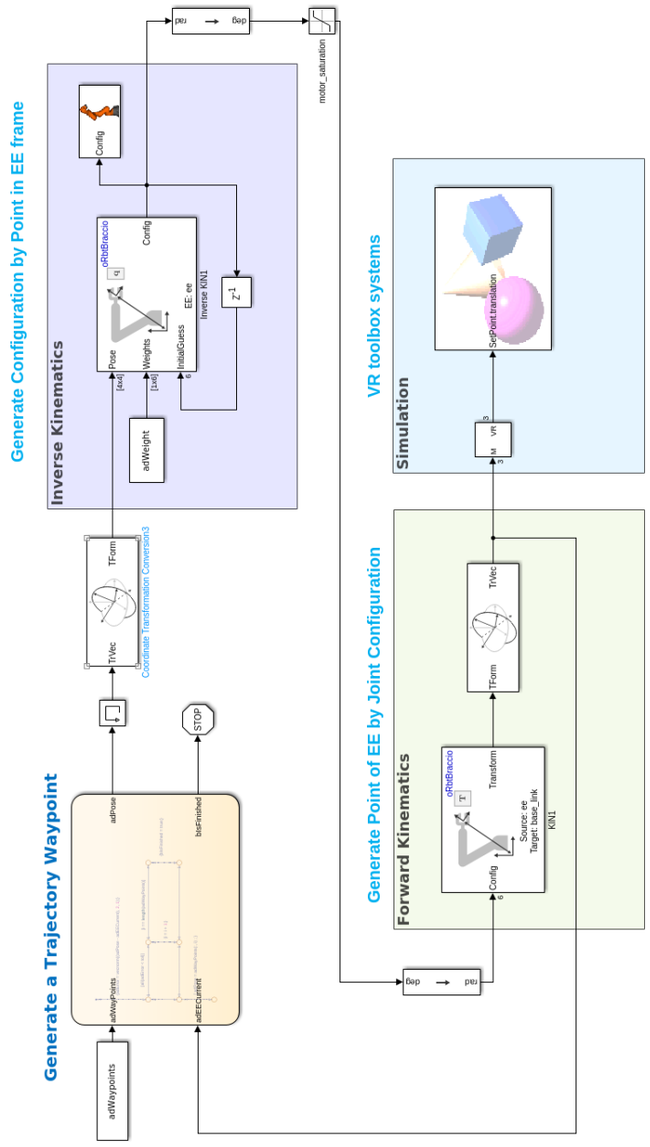

Figura 3.5: mBraccioSimulation.slx

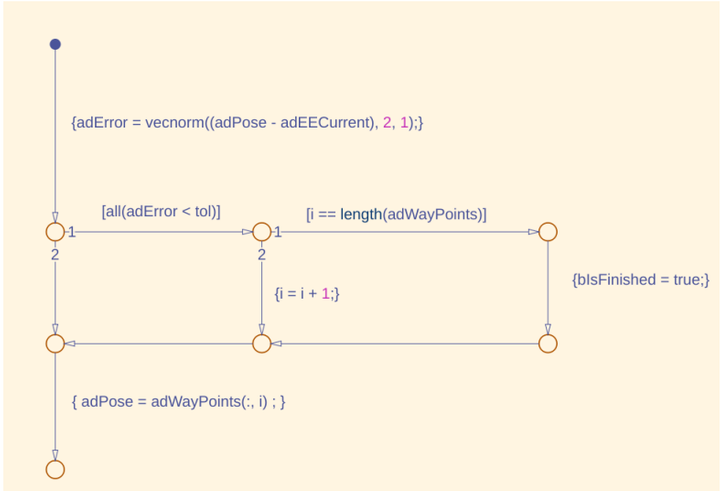

Figura 3.6: Chart che scandisce i vari waypoint

3.3.1 KIN e IKIN con il Toolbox

Vediamo brevemente i blocchi usati per simulare la cinematica del modello:

- **Get Transform** [7]: restituisce la matrice di trasformazione omogenea tra i vari *body frames* del `RigidBodyTree` del robot. Questa trasformazione è usata per convertire le coordinate dal corpo `source` al corpo `target`.

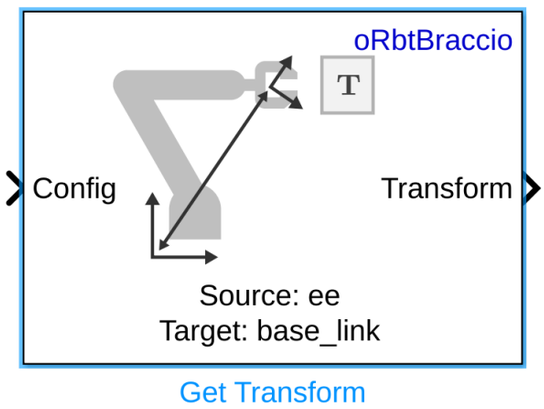

Figura 3.7: Blocco di Cinematica Diretta fornito dal Toolbox

Analizzando la figura 3.7 notiamo i seguenti ingressi e uscite:

  - `Config`: vettore di configurazione dei giunti $\in \mathbb{R}^n$ espresso in radianti;
  - `Transform`: matrice di trasformazione omogenea che mappa, nel nostro caso, il frame *ee* (impostato come `source`) nel frame *base* (impostato come `target`).

- **Inverse Kinematic** [8]: usa un solutore di cinematica inversa per calcolare il vettore di configurazione dei giunti per una posa desiderata dell'end-effector basata sul modello `RigidBodyTree` specificato.

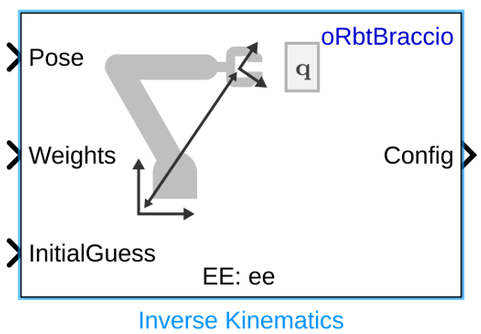

Figura 3.8: Blocco di Cinematica Inversa fornito dal Toolbox

Analizzando la figura 3.8 notiamo i seguenti ingressi e uscite:

  - `Pose`: la posa desiderata dell'end-effector;
  - `Weights`: i pesi sulla tolleranza della posa;
  - `InitialGuess`: configurazione iniziale che cambia andando avanti con le varie iterazioni, (cambiamento reso possibile utilizzando un `delay one step`);
  - `Config`: configurazione dei giunti finale in uscita.

3.3.2 VR Sink

Per controllare e animare un mondo virtuale si utilizza il blocco **VR SINK** [13] riportato in figura 3.9 e in questo progetto è stato utilizzato tale ambiente configurando un mondo virtuale molto basilare per visualizzare il robot in ambiente simulato e rendere semplice la comprensione di quanto implementato.

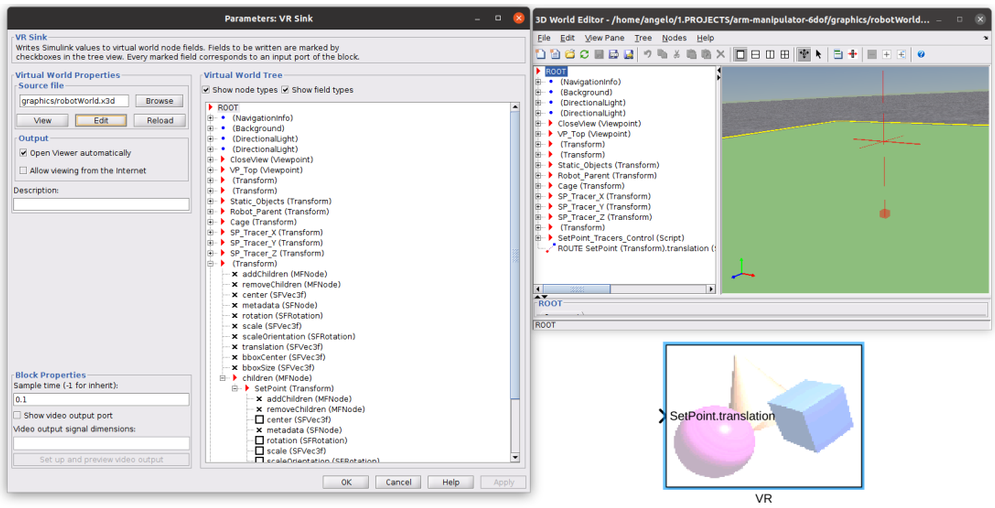

Figura 3.9: Blocco di VR Sink con 3D World Editor

Il blocco ha in ingresso le coordinate xyz di un punto dello spazio e lo rappresenta nel mondo virtuale (file `robotWorld.x3d`) impostato e configurato secondo le proprie esigenze. In linea teorica, per far muovere il robot nel mondo virtuale occorrerebbe creare gli oggetti fisici con tutte le caratteristiche geometriche dei vari giunti e link, ma il blocco `VR RigidBodyTree` [12] rappresentato in figura 3.10 inserisce il modello `rigidBodyTree` direttamente nel mondo virtuale.

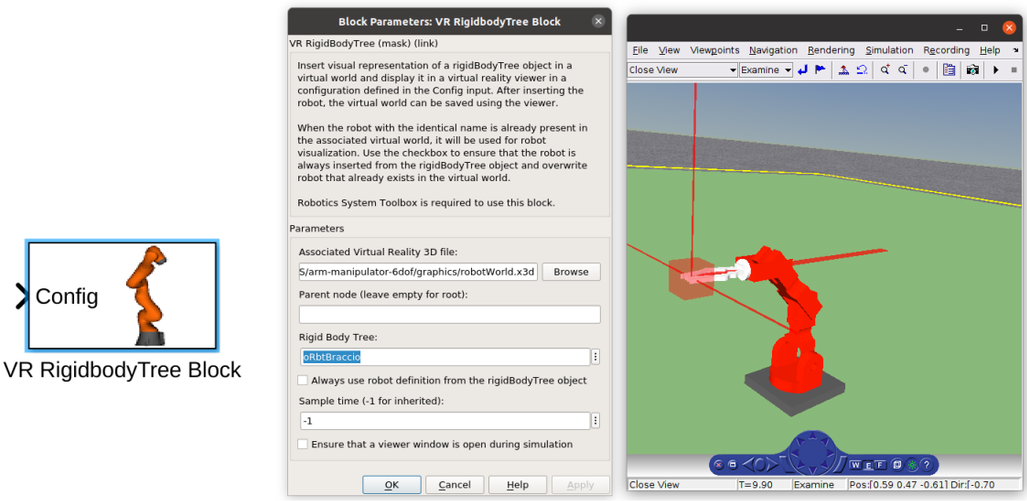

Figura 3.10: Blocco VR RigidBodyTree

Quindi, in definitiva, il blocco `VR Sink` impostato con il mondo virtuale nel file `robotWorld.x3d` configurato con il modello del robot `oRbtBraccio` fornisce una rappresentazione visuale completa ed esaustiva dei movimenti del robot dando in input il solo punto dell'end-effector.

3.4 Implementazione

Oltre che in simulazione, l'inseguimento della traiettoria è stato testato anche su hardware. Per ottenere un confronto rispetto alla soluzione adottata nell'operazione di **Pick&Place**, in questa fase non è stata utilizzata la funzione `NextStep`, ma è stato definito un orizzonte temporale di 20 secondi e sono stati creati sei segnali, uno per ogni giunto. Ogni segnale è costruito definendo una struttura che contiene i valori di posizione del giunto e i valori temporali

```
tot=20;
step=tot/totalPoints;
time=0:step:tot;

figure
base.time=time';
base.signals.values=JointCommandsDeg(:,1);
subplot(2,2,1);
plot(time,JointCommandsDeg(:,1)');
title('Base Motor Signal')
grid on

shoulder.time=time';
shoulder.signals.values=JointCommandsDeg(:,2);
subplot(2,2,2);
plot(time,JointCommandsDeg(:,2)');
title('Shoulder Motor Signal')
grid on

elbow.time=time';
elbow.signals.values=JointCommandsDeg(:,3);
subplot(2,2,3);
plot(time,JointCommandsDeg(:,3)');
title('Elbow Motor Signal')
grid on

wrist.time=time';
wrist.signals.values=JointCommandsDeg(:,4);
subplot(2,2,4);
plot(time,JointCommandsDeg(:,4)');
title('Wrist Motor Signal')
grid on
```

Il segnale definito nel workspace Matlab viene poi usato in Simulink come ingresso dei blocchi dei servo motori di arduino

I principali problemi che sono stati riscontrati in questa applicazione sono due:

1. L'applicazione non può essere eseguita in real-time, per problematiche che verranno esposte in seguito (Capitolo 4). Si deve quindi prima eseguire uno script Matlab per la generazione della traiettoria, la risoluzione della cinematica inversa e la definizione dei segnali e in seguito lanciare il modello Simulink (Figura 3.11).

2. L'inseguimento della traiettoria non è smooth.

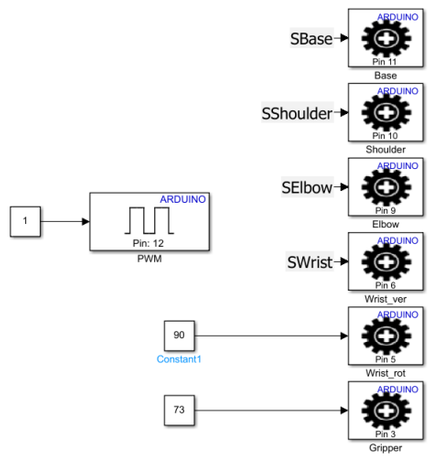

Figura 3.11: Modello Simulink per l'Inseguimento della Traiettoria

Capitolo 4

Pick&Place

L'obiettivo di questa implementazione è quello di far eseguire al robot il task in modo del tutto *stand-alone*, definendo un software *modulare* che si adatta via via alla struttura del robot e del suo workspace. In questo contesto notiamo fin da subito che vi è la sola fase di implementazione, in quanto quella di simulazione è stata discussa nel Capitolo 3 e rimpiazzata da una fase di testing molto accurata.

Descrizione dell'applicazione

Il task **Pick&Place** consiste nel far prendere un oggetto da un **pick point** e portarlo a destinazione nel **place point**. Quindi, per eseguire tale operazione, le fasi sono le seguenti:

1. Dal **pick point** ricavare il vettore di coordinate di giunto da applicare al robot.
2. Applicare tale vettore al robot attraverso la cinematica diretta e verificare dove effettivamente l'end-effector punta.
3. Prendere l'oggetto con una semplice routine di apertura e chiusura della mano.
4. Ricavare il nuovo vettore di coordinate di giunto dal **place point**.
5. Applicare tale vettore al robot come al punto 2.

Problematiche di generazione di codice

Il punto **1** e il punto **4** mostrano come sia necessario un blocco di cinematica inversa e chiaramente anche uno di cinematica diretta. Nella fase simulativa descritta nel Capitolo 3 il blocco di **KIN** (Figura 3.7) e quello di **IKIN** (Figura 3.8) sono forniti dal *Robotics Toolbox*. In fase implementativa, però, non è possibile usare tali blocchi, in quanto il **code generator** non supporta la compilazione per **l'external mode**. Infatti il nostro obiettivo è quello di calcolare la cinematica diretta e inversa a run-time, a differenza del task *Trajectory Tracking* in cui la computazione avveniva preventivamente con uno script Matlab.

Problematiche di smoothing

Per rendere i punti **2** e **5** più smooth possibili si usa la funzione `nextStep`, descritta nel Capitolo 2. Questo permette di contenere gli spike di accelerazione eccessivi e diminuisce il rischio di compromettere la struttura del robot.

Problematiche di iterazione

Il punto **3** viene gestito con una *chart*; per non rischiare di far eseguire a vuoto il task, vi sarà un controllo che tiene conto dei vari punti di **pick** e **place** precedenti, così da non avviare le routine quando non necessario.

Analisi Hardware

Anche se ancora non è stato definito alcun modello per questo task, è semplice intuire da tutte le problematiche esposte che i vari modelli occuperanno molta memoria. In riferimento allo spazio di memoria delle schede disponibili:

- **Arduino UNO**: Equipaggiato con ATmega328P e 32k byte di memoria flash disponibile.
- **Arduino Mega**: Equipaggiato con ATmega2560 e 256k byte di memeoria flash disponibile.

Durante i test è stato superato lo spazio disponibile della scheda di **Arduino UNO**. Quindi, la scelta obbligata è ricaduta su **Arduino Mega**, per la quale è garantita la compatibilitá con la shield del Tinkerkit.

4.1 Implementazione Moduli di Utility

Prima di addentrarci nell'implementazione vera e propria del modello che esegue il task del **Pick&Place**, è bene introdurre delle funzioni locate nella directory `+functions` che verranno utilizzate dai modelli, dagli oggetti e dalle unità di test che descriveremo. Per motivi di spazio, riportiamo solo la *signature* delle funzioni. Il codice integrale si trova in [5].

computeT : Funzione che prende in ingresso i parametri di Denavit Hartenberg ϑ_i e rende in uscita la matrice di trasformazione omogenea $T^i_{i+1}(\vartheta) \in R^{4\times4}$.

```
1: procedure COMPUTET(dD, dQ, dA, dB)
2:     adT = T_i^{i+1}
3:     return adT
4: end procedure
```

carnotRule : Esegue la regola del coseno [17] prendendo in ingresso le lunghezze dei lati del triangolo e calcolando l'angolo tra il lato l_2 e il lato l_3. Inoltre, la funzione restituisce un flag negativo nel caso in cui il coseno dell'angolo fosse maggiore di 1 o minore di -1.

```
1: procedure CARNOTRULE(l_1, l_2, l_3)
2:     C_23 = (l_2^2 + l_3^2 − l_1^2)/(2 l_2 l_3)
3:     ϑ = acos(C_23)
4:     return ϑ
5: end procedure
```

isInsideWorkspace : Verifica se il punto `adPoint` appartiene al workspace dato in igresso. Nella fase di testing (descritta nelle sezioni successive) il flag ritornato viene esaminato come controllo preventivo per avviare le procedure di calcolo della cinematica inversa.

```
1: procedure ISINSIDEWORKSPACE(adPoint, oWorkspace)
2:     verify if adPoint ∈ oWorkspace object
3:     return true or false
4: end procedure
```

4.2 Implementazione KIN e IKIN

Per implementare i blocchi **Get Transform** e **Inverse Kinematics** del *Robotics Toolbox* sono state seguite due strade (Figura 4.1).

1. Usando i blocchi della libreria `Simulink` [6], in particolare `Commonly Used Block`, `Math Operations`, `Ports Subsystem` e `Sink`.

2. Usando i `System Objects` [11], implementando l'interfaccia fornita dalla classe `matlab.System`.

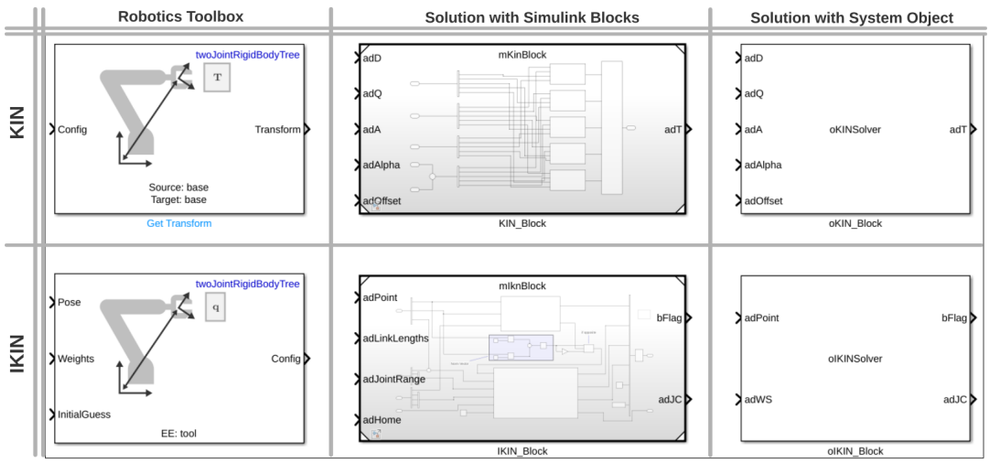

Figura 4.1: Le due strade seguite per l'implementazione

Brevemente, elenchiamo i pro e contro delle due implementazioni:

- L'implementazione **1** risulta più performante rispetto alla **2**, in quanto sfrutta meglio il code generator di Simulink.

- L'implementazione **2** presenta una maggiore modularità, in quanto rispetta il principio di Open-Closed di buona programmazione[1].

- La difficoltà nella realizzazione delle due implementazioni è simile; la **1** risulta più verbosa della **2** a causa dell'elevato numero di connessioni e blocchi.

[1] Aperto all'estensione e chiuso alla modifica

4.2.1 Algoritmo

Per entrambe le soluzioni, gli algoritmi di cinematica diretta **KIN** e inversa **IKIN** implementati sono gli stessi.

KIN

L'algoritmo **KIN** implementato, di cui riportiamo lo pseudocodice in 1, prende in ingresso i parametri di Denavit Hartenberg e restituisce la matrice di trasformazione omogenea con l'ausilio della funzione `computeT`, descritta in precedenza.

Algorithm 1 Algoritmo KIN

```
1: procedure KIN(adD, adQ, adA, adAlpha, adOffset)
2:     adJC = adQ + adOffset
3:     adT = eye(4,4)
4:     for i in [1, nDOF ] do
5:         d = adD[i], q = adJC[i], a = adA[i], alpha = adAlpha[i]
6:         adT = adT * functions.computeT(d, q, a, alpha)
7:     end for
8:     return adT
9: end procedure
```

IKIN

L'algoritmo **IKIN** mostrato nello pseudocodice 2 ha in ingresso il punto dello spazio a cui si vuole puntare e il workspace del robot. In uscita si hanno un flag che indica se tutto è andato a buon fine e il vettore di configurazione dei giunti.

Algorithm 2 Alogritmo IKIN

```
1: procedure IKIN(adPoint, adWS)
2:     x = adPoint(1), y = adPoint(2), z = adPoint(3)
3:     adJC initialization with Home Config
4:     If (adPoint ∉ workspace) then return [false, adJC]
5:     bFlag, base = computeBaseAngle(x, y)
6:     radius = √(x² + y²)
7:     If !bFlag then radius = -radius
8:     z = z - legthOfBaseLink
9:     for φ = −2π → 2π  do
10:        bFlag, shoulder, elbow, wrist = solvePlanar(radius, z, phi)
11:        If bFlag then break
12:    end for
13:    adJC = [base, shoulder, elbow, wrist]
14:    return [bFlag, adJC]
15: end procedure
```

Notiamo che in ingresso alla funzione che risolve il problema planare (riga 10) vi è anche l'angolo di orientamento dell'end-effector $\varphi = q_1 + q_2 + q_3$, ma dato che in questo contesto non è definito a priori, si fa ciclare tale angolo (riga 9-11) nel range $[-2\pi, 2\pi]$ fino a quando il flag di ritorno risulta *true*. Se ciò non dovesse mai accadere, chiaramente, l'algoritmo fallisce.

Fasi dell'algoritmo IKIN

L'algoritmo IKIN è diviso in due fasi

- `computeBaseAngle`: Calcolo dell'angolo del giunto q_0 della base tornando il flag falso in caso di violazione dei limiti fisci del giunto (Pseudocodice in 3).

Algorithm 3 Algoritmo computeBaseAngle

```
1: procedure computeBaseAngle(x, y)
2:     q_0 = atan2(y, x), bFlag = true
3:     If q_0 not in joint range
4:         If (q_0 < 0) then q_0+ = π else q_0− = π
5:         bFlag = false
6:     End If
7:     return [bFlag, q_0 ]
8: end procedure
```

- `solvePlanar`: Calcola i restanti angoli $q_1,\ q_2,\ q_3$ sfruttando la geometria del problema, risolvendo un problema cinematico planare dando in ingresso x come la proiezione del punto sul piano d'appoggio del robot e y come la coordinata z, a meno della lunghezza del link della base (Pseudocodice in 4).

Algorithm 4 Algoritmo solvePlanar

```
1: procedure solvePlanar(x, y, φ)
2:     Adjust coordinate system for base as ground plane
3:     Computing of Wrist coordinates w_x, w_y
4:     Carnot Phases:
5:         bFlag, γ = functions.carnotRule(√(w_y^2 + w_x^2), armLength, forearmLength)
6:         bFlag, β = functions.carnotRule(forearmLength, √(w_y^2 + w_x^2), armLength)
7:     If !FlagsChecking return [false, homeConfig]
8:     q_1 =atan2(w_y, w_x) − β, q_2 = π − γ, q_3 = φ − q_1 − q_2
9:     return [ true, q_1, q_2, q_3 ]
10: end procedure
```

Come si nota dalle implementazioni mostrate, l'algoritmo risulta **specific purpose**; infatti, è estremamente dipendente dalle caratteristiche strutturali del robot. Inoltre, è importante notare come i **giunti di controllo** siano 4 anzichè 5 o 6. Infatti, nonostante i motori presenti siano 6, solo 4 sono quelli interessati alla cinematica diretta e inversa. Il giunto che si occupa della rotazione del polso e dell'apertura della pinza vengono gestiti in maniera indipendente.

4.2.2 Soluzione - Simulink Blocks

Questa soluzione implementa gli algoritmi 1, 2, 3, 4 con la **programmazione visuale** usando i blocchi della libreria standard di Simulink. Uno degli svantaggi di questa implementazione è l'elevato numero di sottosistemi e collegamenti.

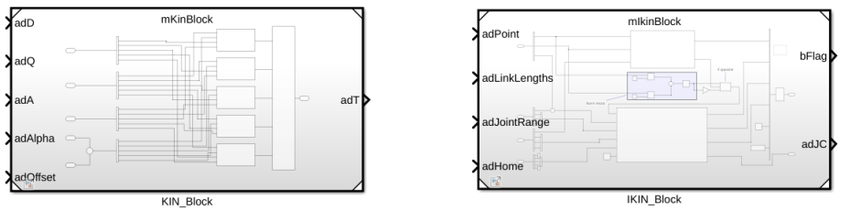

Figura 4.2: Vista ad alto livello dei blocchi KIN e IKIN

Blocco KIN

Il blocco **KIN** prende in ingresso i parametri di Denavit Hartenberg adD, adA, adQ, adAlpha, adOffset $\in R^{DOF}$, e con l'ausilio della funzione `computeT` usata da 5 blocchi matlab function

```
function adT = kin(d, q, a, alpha)
    adT = functions.computeT(d, q, a, alpha);
end
```

si ottengono le 5 matrici di trasformazioni omogenee; come si vede in figura 4.3, queste matrici vengono moltiplicate tra loro in terna corrente ottenendo adT $\in R^{4\times4}$ in uscita.

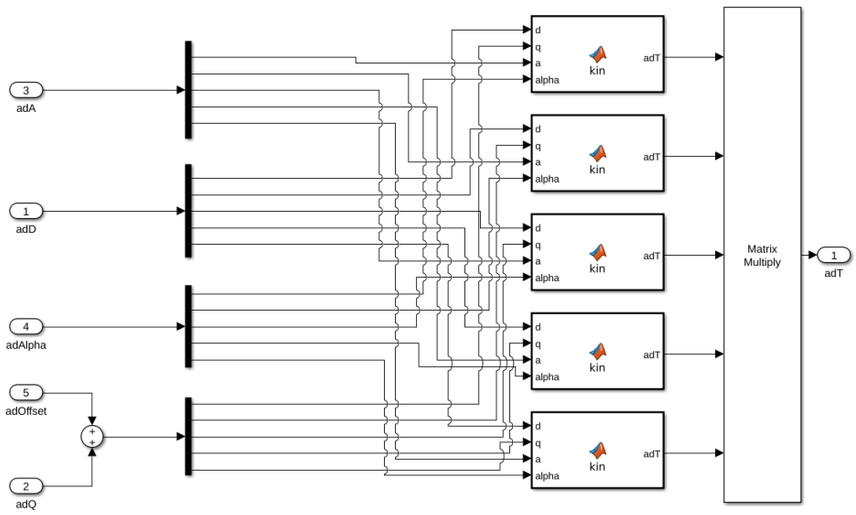

Figura 4.3: Blocco KIN

Blocco IKIN

Il blocco per la cinematica inversa risulta molto più complesso; per semplicità non vengono riportati tutti i sottosistemi, ma solo i più importanti. In ingresso al blocco si hanno:

- Il punto desiderato `adPoint` $\in R^3$.
- La lunghezza dei giunti `adLinkLengths` $\in R^4$.
- I vari range per i giunti `adJointRanges` $\in R^8$. Questo array è da intendersi come una linearizzazione del vettore `adMotorLimits`.

$$\begin{bmatrix} \texttt{minJoint\_1} & \cdots & \texttt{maxJoint\_1} \\ \vdots & \vdots & \vdots \\ \texttt{minJoint\_4} & \cdots & \texttt{maxJoint\_4} \end{bmatrix} \longrightarrow \begin{bmatrix} \texttt{minJoint\_1} \\ \texttt{maxJoint\_1} \\ \vdots \\ \texttt{minJoint\_4} \\ \texttt{maxJoint\_4} \end{bmatrix}$$

- La *homeConfig* `adHome` $\in R^5$.

e in uscita si ha:

- Un flag di controllo `bFlag` che verifica che tutte le procedure siano state svolte correttamente. In caso contrario, segna un *false* in uscita.
- Il vettore di configurazione di giunti `adJC` $\in R^6$. Avere un'uscita a 6 dimensioni permette di collegare questo blocco direttamente ai motori.

Si può osservare il modello completo in Figura 4.4.

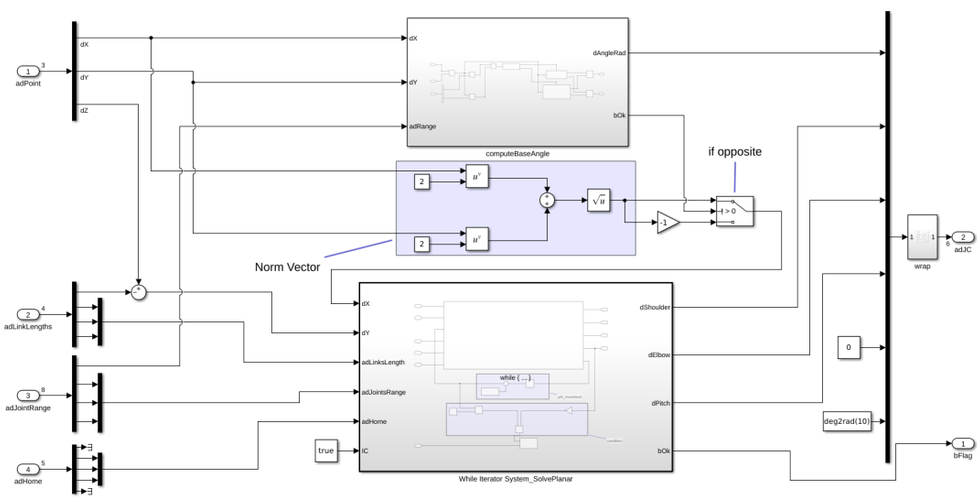

Figura 4.4: mIkinBlock.slx

Parallelismo con il codice

Facciamo un parallelismo tra lo pseudocodice 2 e il blocco mostrato in figura 4.4:

- L'algoritmo `computeBaseAngle` è rappresentato dal sottosistema omonimo e mostrato con maggior dettaglio in figura 4.5. Il blocco `if` verifica se l'angolo è all'interno del range del giunto e invia l'informazione ai vari `if action subsystem`.

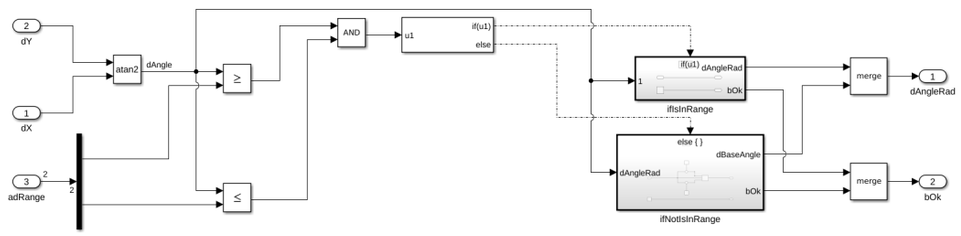

Figura 4.5: ComputeBaseAngle subsystem

- Per far ciclare l'angolo φ è stato usato il blocco `while iterator subsystem`, con la condizione mostrata in figura 4.6 equivalente alle righe 9-12 dell'algoritmo 2.

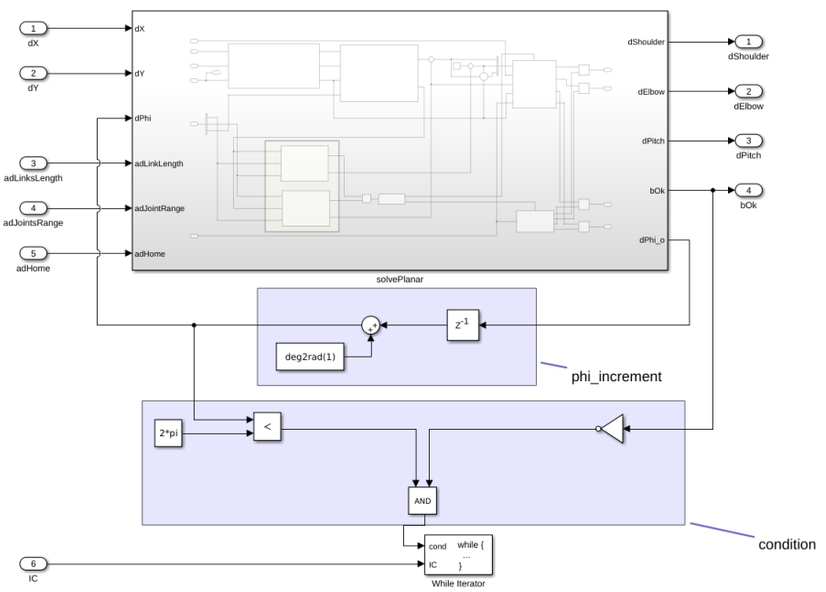

Figura 4.6: Condizione per ciclare l'angolo

- L'algoritmo `solvePlanar`, composto dalle varie fasi, è rappresentato in figura 4.7. Anche per questo algoritmo è stato usato il blocco `if` per verificare i vari flag tornati dalle procedure di Carnot, che sono rappresentate dalle `matlab function`

```
1 function [bOk, dAngleRad] = carnotRule(dL1, dL2, dL3)
2     [bOk, dAngleRad] = functions.carnotRule(dL1, dL2, dL3);
3 end
```

  e i blocchi che si occupano dell'aggiustamento delle coordinate e il calcolo del polso sono solo blocchi matematici, non mostrati in questo contesto per brevità [2].

[2]Si trova nella directory `models| nella repository github [5]`

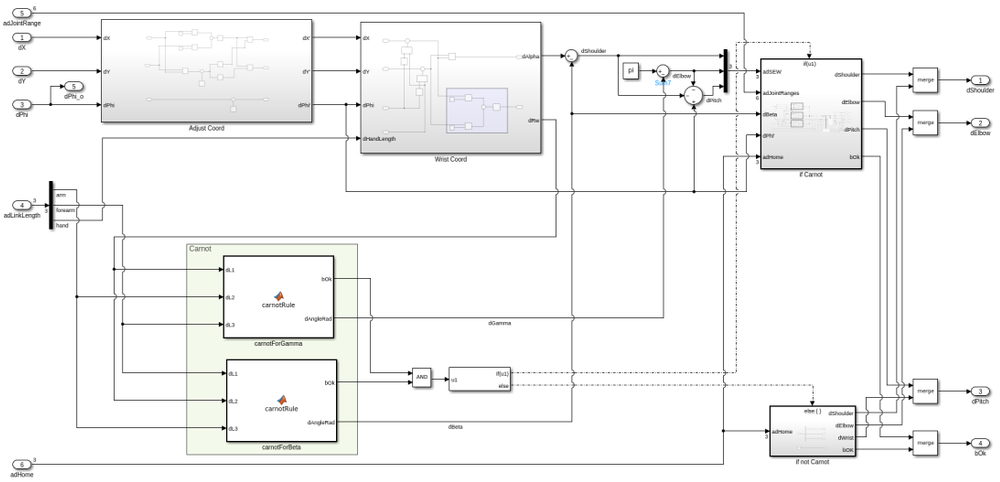

Figura 4.7: Algoritmo solvePlanar

- Se l'algoritmo di Carnot va a buon fine si passa al check per verificare il range degli angoli ottenuti con i range possibili dei vari giunti.

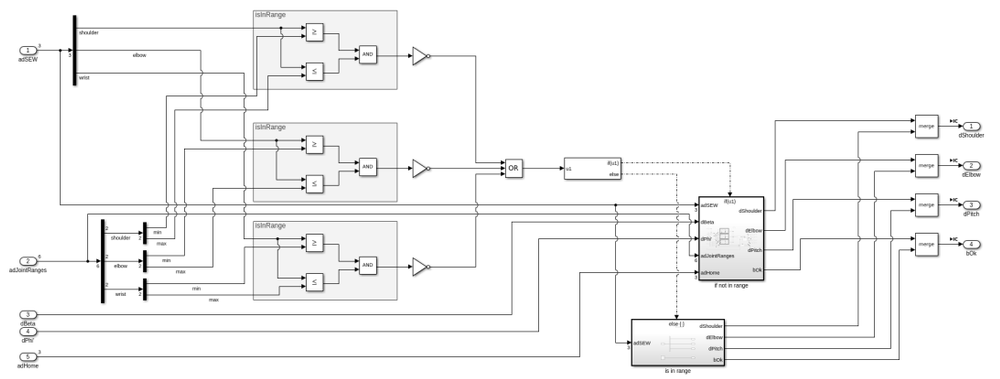

Figura 4.8: Check degli angoli

4.2.3 Soluzione - System Object

Questa soluzione (Figura 4.9) implementa gli algoritmi 1, 2, 3, 4 con la **programmazione ad oggetti**, sfruttando l'interfaccia `matlab.System` fornita da Mathworks. L'idea consiste nell'implementare un oggetto che venga istanziato all'avvio dell'hardware e che venga usato a **run-time** per calcolare di volta in volta il vettore di configurazione di giunti.

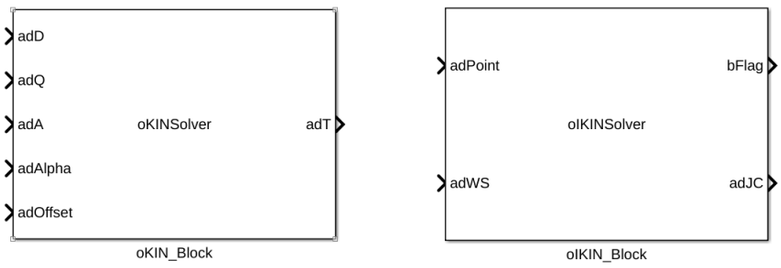

Figura 4.9: Blocchi KIN e IKIN system object

L'architettura software implementata in questo approccio è mostrata in figura 4.10. La classe `Robot5Dof` è *aggregata* con la classe `Joint` e `Link` che definiscono rispettivamente i range dei giunti e la lunghezza dei vari link. L'oggetto viene istanziato passando l'array dei giunti, l'array dei link e la configurazione di Home del robot.

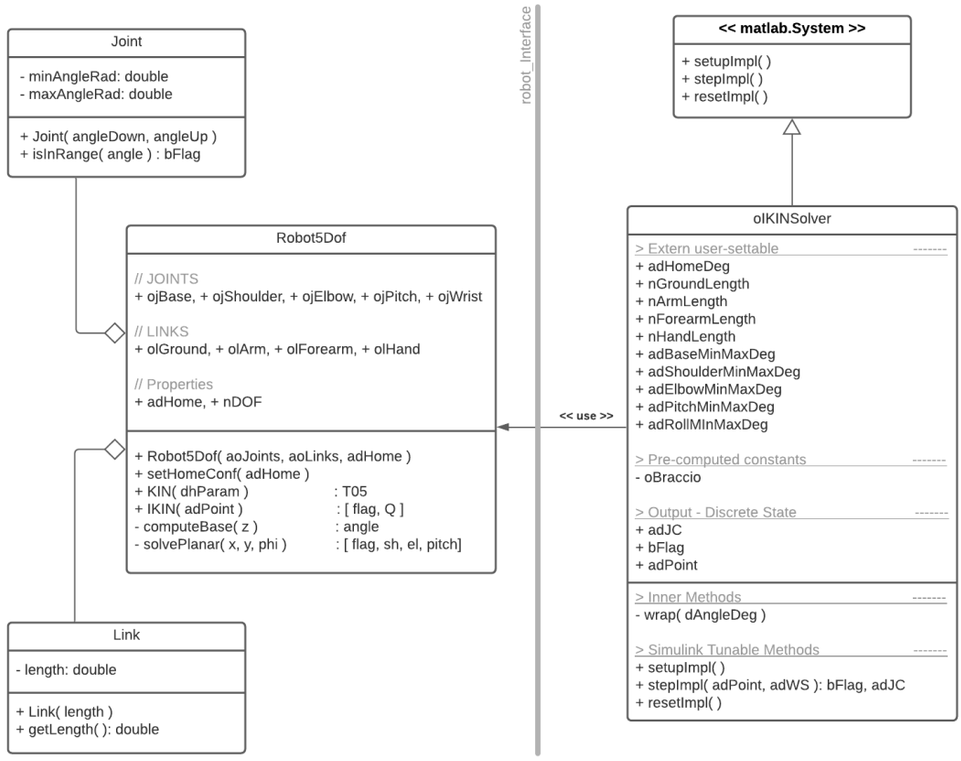

Figura 4.10: Class Diagram

Gli algoritmi di KIN e IKIN sono pubblici e usabili dall'oggetto istanziato `Robot5Dof`. Invece, gli algoritmi che risolvono l'angolo della base e la cinematica planare risultano privati.

Solver

L'architettura viene utilizzata dall'oggetto risolutore `oIKINSolver`, ovvero un oggetto `System Objects` [11] che consiste nella modellazione di sistemi dinamici e la modellazione di un flusso di dati a **run-time**. Ereditando l'interfaccia `matlab.system` occorre implementare i 3 metodi:

- `setupImpl()`: per istanziare una sola volta l'oggetto `Robot5Dof` in fase di inizializzazione, così da non dover istanziarlo ad ogni impulso di clock del processore, risparmiando memoria.
- `stepImpl()`: per aggiornare in modo dinamico a **run-time** lo stato degli oggetti discreti.
- `resetImpl()`: per pulire lo stato degli oggetti.

Gli oggetti del solver sono di 3 tipi:

- Esterni, impostabili dall'utente con una maschera (figura 4.11) fornita dal blocco `matlab system`. Nel nostro caso, sono le caratteristiche del robot.
- Costanti precalcolate, come il robot vero e proprio rappresentato dall'oggetto `oBraccio`.
- Le variabili discrete che tengono lo stato dell'oggetto precalcolato, come le configurazioni di giunto `adJC`, e il flag `bFlag` tiene conto della buona riuscita delle operazioni. La variabile discreta `adPoint` svolge un ruolo molto interessante: permette infatti di **ottimizzare le prestazioni del solver** ricordando l'ingresso precedente e confrontandolo con quello attuale; se dovessero essere uguali, le funzioni di calcolo non vengono rieseguite.

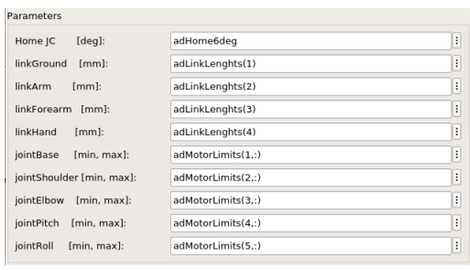

Figura 4.11: Maschera del solver

4.2.4 Modello del Primo Approccio

L'approccio *Naive* (figura 4.12) con i blocchi implementati, per verificare il corretto funzionamento, consiste in un semplice modello che può comandare il robot in modo manuale, impostando il punto obiettivo con delle sidebar. L'attuazione del vettore di configurazione dei giunti è portata a termine con una chart, impostata con un tempo di campionamento pari a 20 millisecondi per ridurre i disturbi dei servo (figura 4.13), che realizza la funzione `nextStep`.

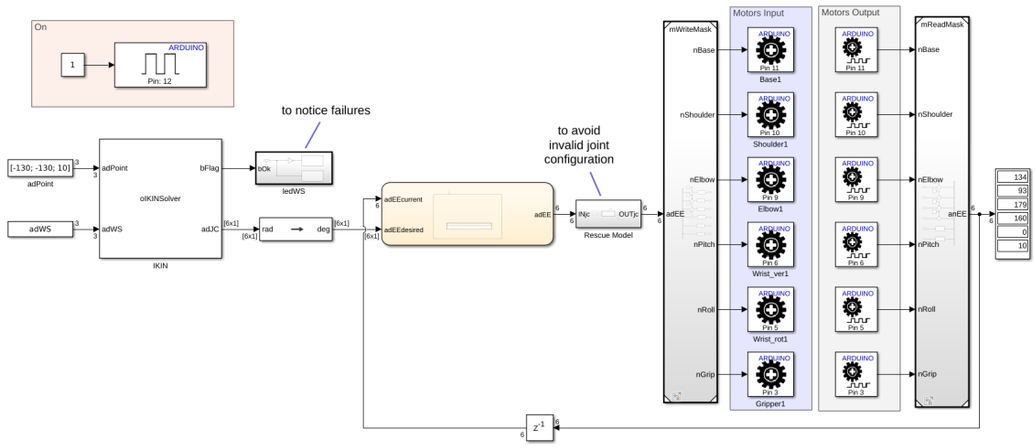

Figura 4.12: mBraccioHW.slx con il system objects

Il sottosistema `rescue Model` prevede delle configurazioni errate ed evita la rottura delle parti fisiche del robot. Verrà approfondito più avanti.

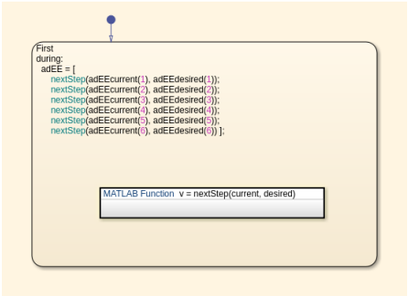

Figura 4.13: chart di mBraccioHW.slx

Ovviamente, le due soluzioni di cinematica inversa sono del tutto intercambiabili (figura 4.14).

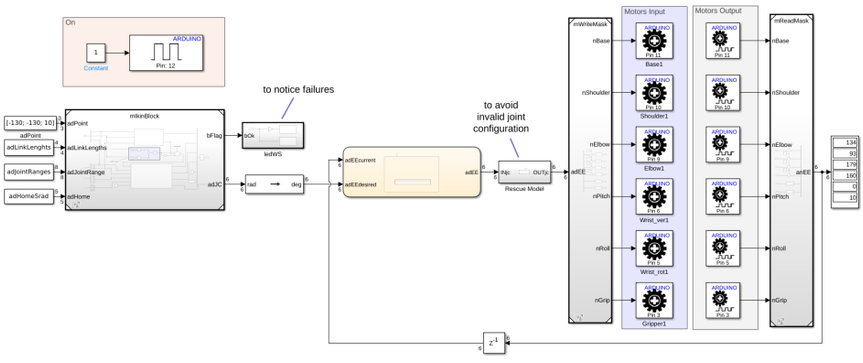

Figura 4.14: Equivalenza delle soluzioni

Il flag di uscita della cinematica inversa viene dato in ingresso al sottosistema `ledWS`, che notifica il successo o meno dell'operazione accendendo un led verde o rosso rispettivamente.

4.3 Testing

Durante la fase di implementazione è necessario provare il robot più volte per verificare la correttezza delle soluzioni. Avendo però a che fare con una piattaforma di sviluppo rigida, occorre prevenire possibili configurazioni di giunto non valide, che possono provocare rotture dei giunti e altri problemi.

I meccanismi di prevenzione sono 2:

- Eseguire gli unit testing per verificare che il sistema software funzioni correttamente.
- Anteporre un sistema ai pin dell'arduino che valuti quali siano le coordinate dell'end-effector in base alla configurazione dei giunti in ingresso, così da individuare eventuali configurazioni che portano alla collissione con la superficie di appoggio del robot.

4.3.1 Unit Test dei moduli

I test che verificano la correttezza del software implementato sono presenti nella directory `+unittest` e vengono eseguiti tutti preventivamente nello script `initHW.m`.

```
+unittests/
    |-- utCarnot.m          # testa la funzione carnotRule
    |-- utComputeT.m        # testa la funzione computeT
    |-- utEstimateMemory.m  # stima la memoria del software
    |-- utFkIk.m            # verifica gli algoritmi KIN e IKIN
```

In particolare, il test `utFkIk` verifica l'algoritmo KIN e IKIN del software in figura 4.10 per ogni punto del workspace del robot (figura 2.5).

Algorithm 5 test utFkIk

```
1: procedure UTFKIK
2:     Define Links, Joints and HomeConfig
3:     oBraccio = Robot5Dof(Links, Joints, HomeConfig)
4:     oWS = makeWS(400, 120, 220)
5:     for p ∈ oWS do
6:         adJC = oBraccio.IKIN(p)
7:         if (oBraccio.KIN(adJC) is p) then PASS else FAIL
8:     end for
9: end procedure
```

Prima di avviare qualsiasi modello nella directory `models` occorre eseguire lo script `initHW` e avere un output simile a quello in figura 4.15

```
Ut: PASS ---- carnotRule
Ut: PASS ---- computeT
UT: PASS ---- Robot built in: 1.08 ms
UT: PASS ---- IK computed in: 1.99 ms
UT: PASS ---- FK computed in: 0.01 ms
Ut: PASS ---- FK_IK class works fine
Ut: PASS ---- Probably bad for arduino Uno
Ut: PASS ---- Probably good for arduino Mega
```

Figura 4.15: Output script preventivo

4.3.2 Rescue Model

Il modello di **Rescue** non fa altro che applicare la funzione di cinematica diretta al vettore di configurazione di giunti per verificare quale sia il punto dell'end-effector. Come si nota in figura 4.16, il modello verifica che la coordinata z risultante non sia troppo piccola; in caso contrario, restituisce la configurazione di Home.

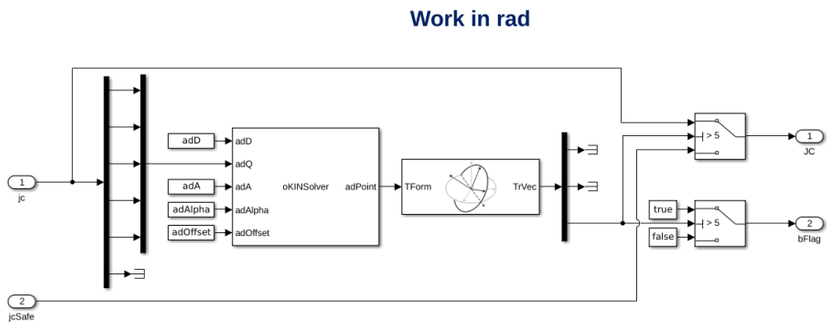

Figura 4.16: mRescue.slx

4.4 Implementazione Pick&Place

Abbiamo tutti gli ingredienti per comporre il modello che esegue il task del **Pick&Place**. In maniera del tutto equivalente, proseguiamo usando come risolutore di cinematica diretta e inversa la soluzione con il `System Objects`.

4.4.1 Modello Simulink

Il modello è mostrato in figura 4.17, dove si possono notare:

- Uno switch manuale per testare più punti di pick.
- La funzione Matlab *compose* che ha il compito di comporre una matrice di configurazioni di giunto, chiamata jointCommands.

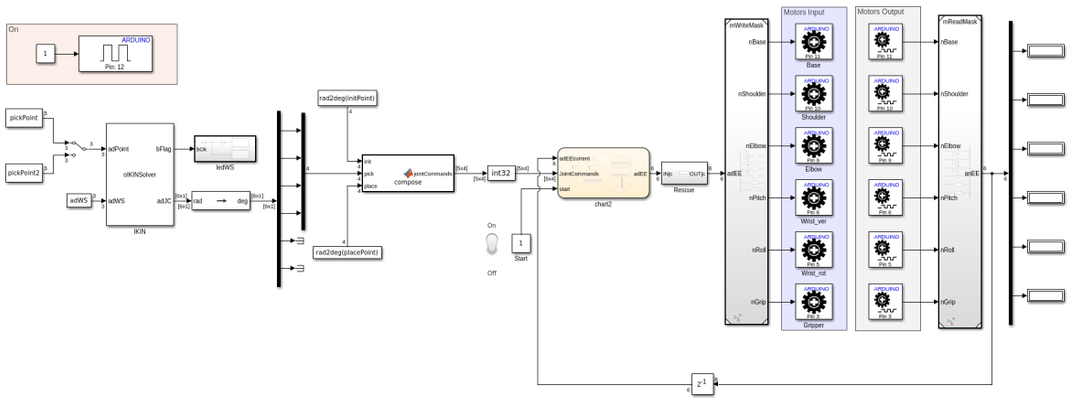

Figura 4.17: mBraccioPickAndPlace.slx

Come abbiamo sempre accennato, il risolutore di cinematica inversa lavora con 4 giunti e lascia gli ultimi 2 costanti. Questo perchè il movimento dell'end-effector è gestito dalla chart in figura 4.18

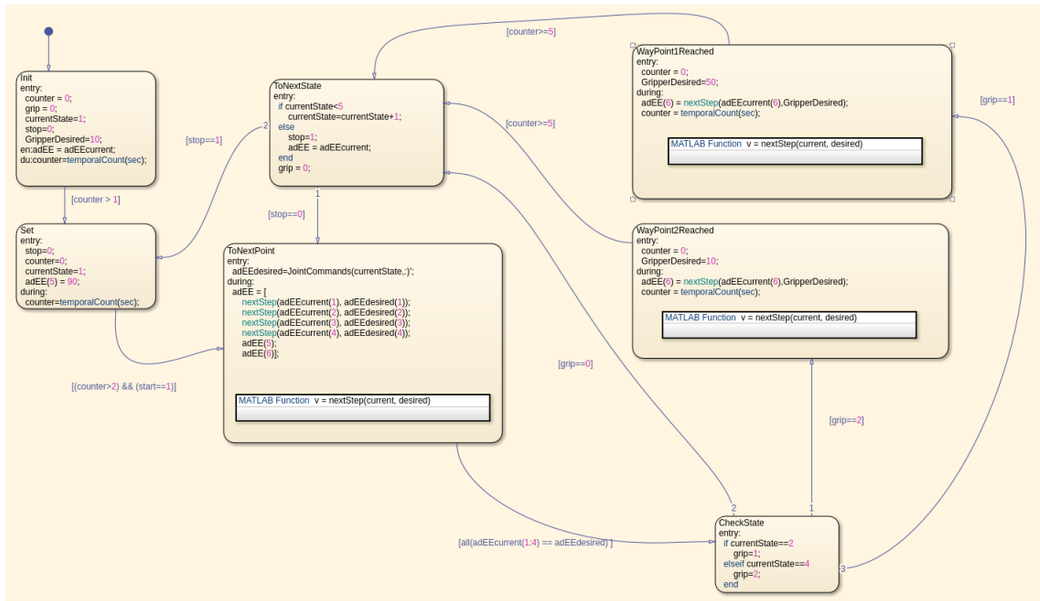

Figura 4.18: Chart che scandisce la matrice e gestisce l'EE

Vediamo come funziona la chart:

- Stato 1 - **Inizializzazione**: le variabili che verranno usate vengono inizializzate e viene inizializzata la configurazione di uscita (in particolare viene posta uguale alla configurazione corrente). In questo stato si seleziona la prima riga della matrice di ingresso attraverso una variabile *CurrentState*, che assume valori interi tra 1 e il numero di righe della matrice.

- Stato 2 - **Wait**: è uno stato di attesa in cui si aspetta di avere un punto di pick disponibile. In prima battuta l'attesa è interrotta con l'azionamento di un pulsante esterno alla chart.

- Sato 3 - **ToNextPoint**: si passa dalla configurazione corrente alla configurazione desiderata, che è quella dettata dalla riga selezionata della matrice di configurazioni dei giunti. A tale scopo si utilizza la funzione `NextStep`, per ottenere dei movimenti molto smooth. Si esegue la transizione allo stato successivo solo quando la configurazione desiderata è stata raggiunta con successo.

- Stato 4 - **CheckWaypoint**: si controlla se ci troviamo in uno dei punti di pick o place: se è così il gripper deve essere opportunamente azionato, altrimenti possiamo passare direttamente alla configurazione successiva.

- Stato 5a - **Waypoint1Reached**: se è stato raggiunto il `pickPoint` dobbiamo azionare il giunto di gripper per far chiudere la mano e afferrare l'oggetto da terra. Anche per questo movimento si utilizza la funzione `NextStep`.

- Stato 5b - **Waypoint2Reached**: se è stato raggiunto il `placePoint` dobbiamo azionare il giunto di gripper per far aprire la mano e rilasciare l'oggetto. Anche per questo movimento si utilizza la funzione `NextStep`.

- Stato 6 - **ToNextState**: la variabile `CurrentState` viene incrementata di un'unita, in modo da passare dalla riga corrente della matrice di configurazione alla successiva. Se c'è una nuova configurazione si ricomincia il ciclo dallo stato 3, mentre se non è disponibile una nuova riga della matrice ci riportiamo nello stato 2 in attesa di una nuova traiettoria.

4.4.2 Esempio di Applicazione

Consideriamo il task composto dalla seguente sequenza:

1. Il robot parte in `homeConfig`;
2. Raggiunge il punto `pickPoint` $= \begin{bmatrix} -130; & -130; & 10 \end{bmatrix}$ e prende l'oggetto target;
3. Posa l'oggetto target nel `placePoint` $= \begin{bmatrix} -130; & 130; & 10 \end{bmatrix}$;
4. Raggiunge il secondo punto `pickPoint2` $= \begin{bmatrix} 160; & -200; & 10 \end{bmatrix}$ e prende l'oggetto target;
5. Posa l'oggetto target nel `placePoint` $= \begin{bmatrix} -130; & 130; & 10 \end{bmatrix}$.

Il modello 4.17 produce i seguenti risultati,

$$\texttt{pickPoint} = \begin{bmatrix} -130; -130; 10 \end{bmatrix} \implies \texttt{adJC} = \begin{bmatrix} 135 & 93 & 179 & 160 & 0 & 10 \end{bmatrix}^T$$

$$\texttt{placePoint} = \begin{bmatrix} -130; 130; 10 \end{bmatrix} \implies \texttt{adJC} = \begin{bmatrix} 44 & 93 & 179 & 160 & 0 & 10 \end{bmatrix}^T$$

$$\texttt{pickPoint2} = \begin{bmatrix} 160; -200; 10 \end{bmatrix} \implies \texttt{adJC} = \begin{bmatrix} 38 & 46 & 56 & 1 & 0 & 10 \end{bmatrix}^T$$

4.5 Validazione dei Risultati

Il cuore dell'implementazione del task Pick&Place è chiaramente l'oggetto `Robot5Dof`, gestito dagli oggetti `oIKINSolver` e `oKINSolver`. Il motivo principale di questa scelta, come già detto, è quello di sostituire il corrispettivo oggetto del Robotics Toolbox e far fronte a tutte le problematiche del `code generator`.

Deve quindi esistere una sorta di *equivalenza* fra gli oggetti mostrati in figura 4.1. Per verificare tale equivalenza è stato implementato un ulteriore modello di confronto `mKinIkinValidator.slx`, avviato con lo script preventivo `scriptValidation.m`. Tale modello non confronta direttamente i vettori di configurazione di giunto, dato che la soluzione al problema di cinematica inversa non è in forma chiusa per il blocco del toolbox, ma piuttosto confronta i vettori risultanti dalla cinematica diretta con in ingresso i vettori di configurazione di giunti.

4.5.1 Confronto con il toolbox

Per avere dei risultati attendibili occorre calcolare un *fattore di scala* nFS per poter confrontare i blocchi in figura 4.9 con quelli in figura 3.7 e 3.8. Infatti, i blocchi del Toolbox usano il modello URDF, che non riporta le dimensioni reali del robot.

Il fattore di scala per passare dalle coordinate reali a quelle simulate dal toolbox risulta,

$$nFS = \frac{1}{963.455} \, .$$

La validazione è stata eseguita su tutti i punti `adPoints` generati nel workspace con la funzione `makeWS`, anteponendo una chart in ingresso che scandisce un punto alla volta. In fondo al modello, inoltre, si controlla la norma del vettore differenza tra i due risultati (opportunamente scalati dove serve) e si tiene conto di tutti i flag restituti dai controlli. Il modello è mostrato in figura 4.19.

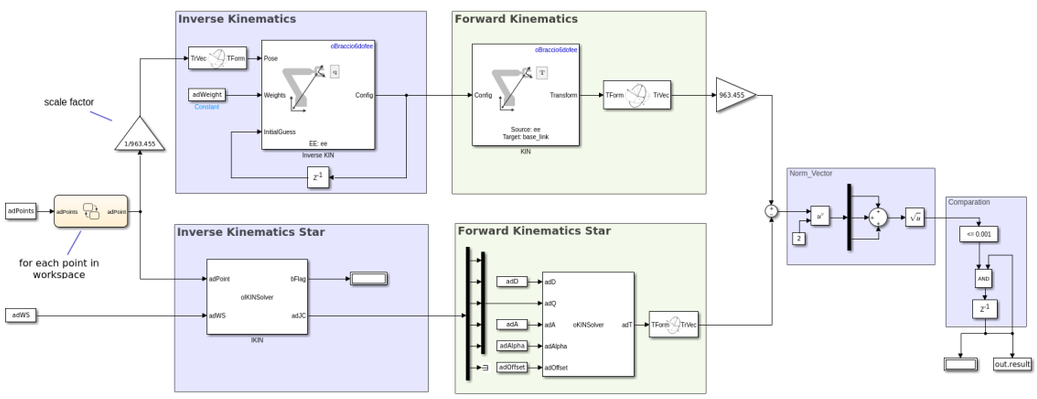

Figura 4.19: mKinIkinValidator.slx

Un esempio, mostrante il solo punto $[-130 \ -130 \ -130]^T$ è riportato in figura 4.20, dove si può notare lo stesso output dai due blocchi.

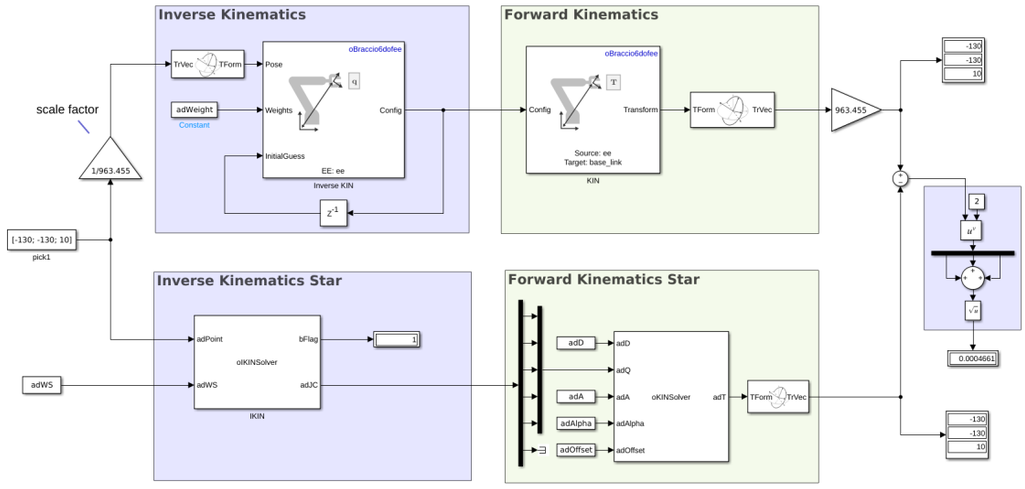

Figura 4.20: Esempio di validazione con un solo punto

Bibliografia

[1] Arduino. Arduino tinkerkit braccio. https://create.arduino.cc/projecthub/rpatterson/arduino-tinkerkit-braccio-robot-arm-kinematics-1a8303.

[2] Arduino. Braccio library. https://www.arduino.cc/reference/en/libraries/braccio/.

[3] Sebastian Castro Jose Avedano. Designing robot manipulator algorithms. https://it.mathworks.com/help/robotics/manipulators.html.

[4] Arduino Lab. Braccio library. https://github.com/bcmi-labs/arduino-library-braccio.

[5] Angelo D'Amante Lorenzo Falai. arm-manipulator-5dof. https://github.com/AngeloDamante/arm-manipulator-5dof, 2021.

[6] Mathworks. Block libraries. https://it.mathworks.com/help/simulink/block-libraries.html.

[7] Mathworks. Get transform block. https://it.mathworks.com/help/robotics/ref/gettransformblock.html.

[8] Mathworks. Inverse kinematic block. https://it.mathworks.com/help/robotics/ref/inversekinematics.html.

[9] Mathworks. Robotics toolbox. https://it.mathworks.com/help/robotics/.

[10] Mathworks. Simulink 3d animation. https://it.mathworks.com/help/sl3d/index.html?s_tid=CRUX_lftnav.

[11] Mathworks. System objects. https://it.mathworks.com/help/matlab/use-system-objects.html.

[12] Mathworks. Vr rigid body tree visualization. https://it.mathworks.com/help/sl3d/examples/rigid-body-tree-visualization.html.

[13] Mathworks. Write data from simulink model to virtual world. https://it.mathworks.com/help/sl3d/vrsink.html.

[14] Siciliano-Sciavicco-Villani-Oriolo. *Robotica modellistica, pianificazione e controllo*. Collana di automatica. McGrawHill.

[15] Tinkerkit. Braccio, quick start guide. https://www.robotstore.it/rsdocs/documents/Tinkerkit_braccio_robotico_guida.pdf.

[16] Damiani Volpi. *RoboGame*. Robotica con Arduino. Ala Libri.

[17] Wikipedia. Teorema del coseno — wikipedia, l'enciclopedia libera, 2021. [Online; in data 20-gennaio-2022].