Università degli Studi di Trieste

Dipartimento di Ingegneria e Architettura

Corso di Studi in Ingegneria Elettronica

Tesi di Laurea Triennale

Simulazione di un sistema fisico per un'applicazione di reinforcement learning

Laureando:
Ionuţ Alexandru Pascariu

Relatore:
prof. Gianfranco Fenu

ANNO ACCADEMICO 2019-2020

Indice

1 Introduzione

Nell'ambito dell'*Artificial Intelligence* gli algoritmi di apprendimento automatico vengono utilizzati per addestrare agenti autonomi ad apprendere una strategia che permetta di raggiungere determinati obiettivi, interagendo con l'ambiente in cui sono immersi.
In particolare con *Reinforcement Learning* (anche chiamato RL) si intende qualsiasi tecnica di apprendimento automatico che si basa su algoritmi di apprendimento per rinforzo. Alla base delle tecniche di RL c'è il concetto di ricompensa (*reward*). Ogni azione che l'agente sceglie di applicare, sulla base della strategia che sta man mano imparando, avvicinerà oppure allontanerà il raggiungimento dell'obiettivo nel futuro (potrebbe anche portare a non raggiungerlo mai, quindi al fallimento). Ogni scelta fatta riceve una valutazione, una ricompensa, in funzione dell'esito che ne è conseguito: l'obiettivo è più vicino? E' stato raggiunto? Ci si sta allontanando dall'obiettivo? Non è più possibile ottenerlo?
Tutti gli algoritmi di RL cercano di individuare la strategia ottima, cioé quella che permette di massimizzare la ricompensa complessiva, ottenuta sommando tra loro tutte le ricompense ottenute passo dopo passo, fino al raggiungimento dell'obiettivo.
Gli algoritmi di RL procedono per tentativi e soprattutto nelle fasi iniziali dell'addestramento questi tentativi portano ad errori ed al fallimento, non raggiungimento dell'obiettivo. Per evitare di danneggiare inutilmente i dispositivi fisici, che dovrebbero essere governati in base alla strategia appresa, di solito si preferisce eseguire l'addestramento in simulazione, creando un modello dell'ambiente.
Negli ultimi anni c'è stato un rinnovato interesse verso le tecniche RL, che ha portato allo sviluppo sia nuove tecniche di apprendimento, basate su reti neurali, che di simulazioni, per eseguire l'addestramento degli agenti RL.
Lo scopo di questo lavoro di tesi consiste nell'implementare la soluzione a un classico problema di apprendimento RL facendo uso di un software per la simulazione in ambito robotico per creare l'ambiente con cui interagirà l'agente RL, mentre per l'addestramento si vuole usare un software diverso che mette già a disposizione algoritmi di tipo *deep RL* (cioé che sfruttano reti neurali), da configurare ed addestrare. In particolare si è scelto di implementare un algoritmo di apprendimento, nel software Matlab, e farlo interagire con una riproduzione tridimensionale virtuale del *Ball on Plate*, creata nel simulatore CoppeliaSim. Per costruire al meglio il modello tridimensionale si è fatto uso del manuale, tratto da [1], posto a disposizione da CoppeliaSim, leggendo e studiando le parti necessarie ai fini della tesi. Seguendo tale guida, in questa tesi verrà presentato le operazioni necessarie ai fini della creazione del modello del *Ball on Plate*.
Il collegamento tra il client Matlab e il server CoppeliaSim è stato realizzato utilizzando le remote API, funzioni che permettono di instaurare il collegamento tra i due software e di controllare il simulatore da un'applicazione esterna, in questo caso Matlab. Verrà inoltre mostrato un esempio nel quale, grazie alle remote API, dal lato client verranno scelte le azioni da eseguire, e nel lato server si osserveranno queste azioni eseguite.
L'ultima parte invece riguarderà l'implementazione dell'algoritmo di apprendimento. Si è utilizzato il *toolbox* di *Reinforcement Learning* per creare una applicazione di controllo

della sfera sulla piattaforma. La parte finale quindi è stata divisa in una parte teorica nella quale verranno alcune nozioni base sul funzionamento del *Reinforcement Learning*, e in una parte pratica che riguarderà la scrittura del codice. L'obiettivo dell'algoritmo sarà quello di poter portare la sfera, collocata sulla piastra quadrata, indipendentemente dalla sua posizione iniziale, al centro della piastra.

Il codice scritto, necessario per l'apprendimento, è diviso in due parti. La prima è l'ambiente (anche chiamato *environment*, codice che servirà per inizializzare ogni episodio, eseguire ogni passo di simulazione e calcolare le coordinate e velocità della sfera sfruttando tecniche di Image Processing. La seconda parte invece è l'algoritmo utilizzato per l'apprendimento (chiamato anche *train*, fornito da MathWorks. I due codici interagiranno fra di loro in quanto nel *train* verrà creato un agente che deciderà le azioni da eseguire, queste verranno impostate nell'ambiente il quale eseguirà il passo di simulazione e fornirà all'agente un risultato.

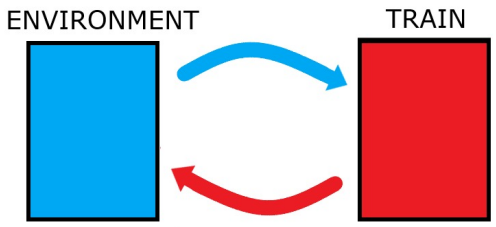
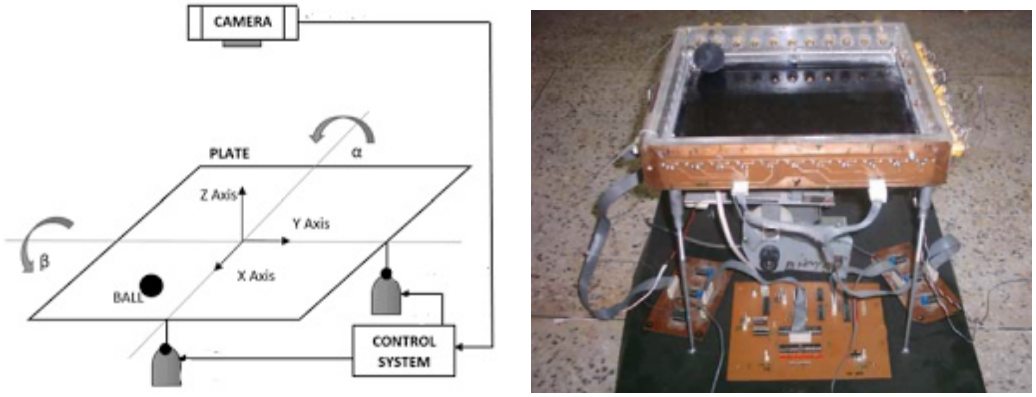


Figura 1: Interazione tra Environment e Train

L'ultima parte della tesi si concentra sull'analisi di alcuni esperimenti effettuati. Verrà mostrato che tipo di comportamento ha avuto l'agente commentando i risultati ottenuti.

2 Ball on Plate - Descrizione del sistema

Il sistema include la piastra quadratica in metallo, che è fissata in modo mobile al centro. Questo permette al sistema di avere quattro gradi di libertà, due della piattaforma e due della sfera che sarà appoggiata su di essa, e quindi di poter ruotare il piatto rispetto al proprio asse x e y.
Il sistema *Ball on Plate* è costituito da una piattaforma rigida, solitamente di forma quadrata o rettangolare, sulla quale è posta una palla (sfera), che può muoversi rotolando senza strisciare. La piattaforma è vincolata: non può traslare, ma può variare la sua inclinazione. Indicando con (x,y,z) il sistema di riferimento (in coordinate cartesiane) posizionato nel centro di massa della piastra, in modo che gli assi x ed y siano nel piano della piastra e paralleli ai lati della piattaforma, mentre l'asse z risulti ortogonale alla piastra (come in figura 2 (a)). La piattaforma può ruotare sia attorno all'asse x che attorno all'asse y. Due attuatori, motori elettrici, permettono di far variare gli angoli di rotazione (angoli α, β in figura 2 (a)), mentre una videocamera, ad esempio una webcam, permette di ottenere la posizione della sfera sulla piattaforma, rispetto alle coordinate (x,y). In figura 2 (b) viene mostrato un modello reale del *Ball on Plate*.
Lo scopo del controllore in retroazione, che tipicamente viene progettato, consiste nel portare e mantenere la sfera in una posizione desiderata sulla piattaforma, oppure nel farla muovere seguendo una traiettoria desiderata. (capitolo tratto da [2])



(a) *Sistema di riferimento del Ball on Plate (tratto da [3])*

(b) *Modello reale del Ball on Plate (tratto da [4])*

Figura 2

3 Creazione del modello

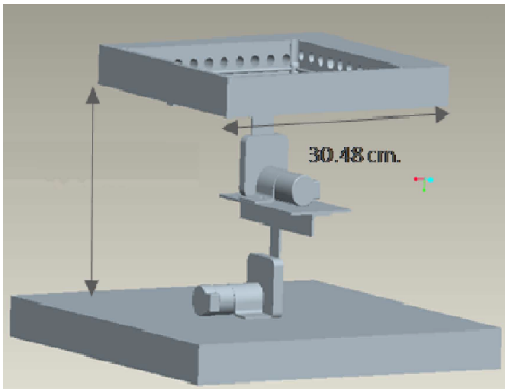
In questo capitolo verrà spiegato il procedimento per poter realizzare un modello tridimensionale per il *Ball on Plate*. Per la creazione del modello si è seguita una guida ben specifica messa a disposizione direttamente da Coppelia Robotics. (guida per la creazione del modello tratto da [5])

3.1 CoppeliaSim

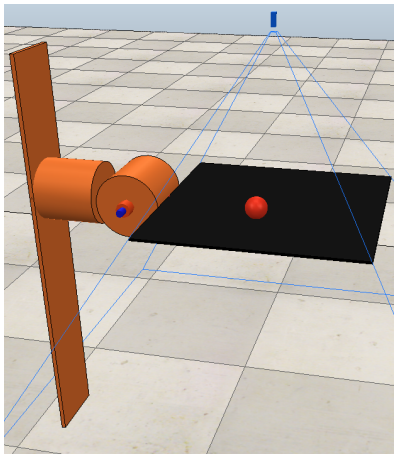
Per quanto riguarda la realizzazione del modello del Ball on Plate si è deciso di realizzarlo nel software CoppeliaSim, creato dall'azienda Coppelia Robotics. Oltre ad essere una piattaforma di sperimentazione di robot virtuale è anche un software di simulazione di realtà virtuale che consente la simulazione di molti sistemi robotici. Tutti gli oggetti e i modelli possono essere controllati individualmente utilizzando, ad esempio, routine incorporate, plug-in o server remoti collegati tramite le Application Program Interface o comunemente chiamate API. Queste routine possono essere implementate usando C / C ++, Python, Java o Matlab. Inoltre questo software permette di importare dati CAD da un'applicazione esterna (ad esempio dal sofware SketchUp).

3.2 Strumenti utilizzati

Per iniziare a costruire il Ball on Plate bisogna partire scegliendo un modello da seguire che sia il più semplice possibile ma che comunque rispecchi la realtà. Per questo motivo si è deciso di realizzare un manipolatore a due gradi di libertà, al quale verrà collegata una piattaforma, realizzando così una struttura simile a quella in figura 3 (a), mentre in figura 3 (b) viene mostrato il modello realizzato nel software CoppeliaSim.



(a) *Modello di riferimento del Ball on Plate (tratto da [4])*



(b) *Modello creato nel software CoppeliaSim*

Figura 3

Non essendo una struttura complessa, per la realizzazione del macchinario si sono utilizzati pochi elementi tra i quali:

- *Shapes* (Forme):
  Sono delle forme rigide composte da facce triangolari (ogni forma in CoppeliaSim è composta da triangoli, più è complessa la forma e da più triangoli sarà composta).

- I *joint* (giunti):
  Sono oggetti che hanno almeno un grado intrinseco di libertà, servono per poter fornire mobilità al robot. Ci sono tre tipi di joint dentro CoppeliaSim:

  - *revolute joints*: Hanno un grado di libertà e sono usati per fornire agli oggetti movimenti di rotazione intorno all'asse z del *joint*
  - *prismatic joints*: Hanno un grado di libertà e sono usati per fornire un mevimento traslazionale agli oggetti lungo l'asse z del *joint*
  - *spherical joints*: Hanno tre gradi di libertà e permettono un movimento rotatorio all'oggetto intorno ai propri assi di riferimento

- *Vision sensor* (Sensore di visione):
  E' un sensore virtuale di tipo fotocamera, che permette di acquisire immagini. Il vision sensor servirà per poter determinare la posizione e la velocità del elemento sferico, posizionato sulla piattaforma, tramite un codice esterno su Matlab.

- *Force sensor* (Sensore di forza):
  E' un collegamento rigido tra due forme in grado di misurare forze e coppie trasmesse. Ha anche la capacità di rompersi se viene superata una soglia di forza o di coppia. Inoltre il *force sensor* funge da collegamento tra due oggetti.

(tratto da [6])

3.3 Proprietà degli oggetti

Alcuni dei oggetti possono avere delle proprietà che consentono ad altri oggetti o al simulatore stesso in fase di calcolo di interagire con essi. Gli oggetti possono essere:

- *Collidable*: oggetti *collidable* possono essere testati per la collisione con altri oggetti *collidable*.

- *Measurable* (Misurabile): gli oggetti misurabili sono oggetti che possono essere utilizzati per il calcolo della distanza minima rispetto ad altri oggetti misurabili.

- *Detectable* (Rilevabile): gli oggetti rilevabili possono essere rilevati dai sensori di prossimità.

- *Renderable* (Renderizzabile): oggetti renderizzabili possono essere visti o rilevati dai *vision sensor*.

- *Respondable* (Rispondente): una forma rispondente provocherà una reazione di collisione con altre forme dello stesso tipo e saranno influenzati nel loro movimento se sono dinamici. Se la forma non è *renderable* allora non verrà calcolata nessuna risposta di collisione se si scontrano con altre forme.

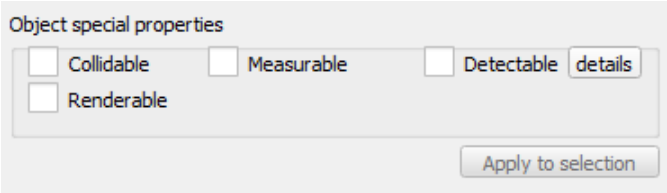


Figura 4: Proprietà speciale degli oggetti

CoppeliaSim ha inoltre implementato una tabella composta da 16 celle nella quale si può impostare il livello di visibilità di un oggetto. Di conseguenza si potrà, per ogni categoria di oggetto, scegliere il livello di visibilità e impostarlo attivo o meno. Questo è comodo poichè ci sono oggetti, come ad esempio i *joint* oppure il *vision sensor*, che dovranno essere nascosti e non saranno visibili. (paragrafo tratto da [7])

3.4 Costruzione delle forme visibili

Essendo ogni forma composta da triangoli l'obiettivo nella prima fase della creazione sarà quello di ridurne al minimo il numero. Questo porterà la figura ad essere più leggera e di conseguenza per il simulatore ci saranno meno calcoli da effettuare in fase di simulazione. Le forme vengono classificate da CoppeliaSim in base alla forma geometrica, possono essere:

- Pure (forma primitiva): forma con geometria regolare (sfere, cubi, dischi,..), basso numero di triangoli ed è dotata di un colore.
- Convesse: forma geometrica irregolare, numero discreto di triangoli ed è dotato di un colore.
- Composte di forme pure: un insieme di forme primitive
- Composte di forme convesse: un insieme di forme convesse

Per rendere il più possibile leggera la figura si è deciso di far uso di sole forme primitive poiché, rispetto alle altre forme, sono composte dal minor numero di triangoli. Questa scelta è stata fatta anche perché si è dato più peso all'aspetto pratico, quindi per facilitare il simulatore, che all'aspetto estetico. Per creare il manipolatore sono stati utilizzati due cilindri e un cuboide. Invece per la palla e la piattaforma si è fatto uso di una sfera e un cuboide. (tratto da [8])

3.5 Forme statiche e dinamiche

Nella creazione del modello di un robot in CoppeliaSim bisogna dividere lo scheletro della macchina in due parti: una parte statica finalizzata all'aspetto estetico e una dinamica finalizzata alla parte computazionale, quindi all'aspetto pratico. La parte statica di conseguenza sarà visibile mentre la dinamica sarà invisibile, dunque le verrà assegnato un *layer* di visibilità diverso da quello della parte statica, che in seguito verrà disabilitato. Una forma dinamica sarà influenzata da forze/coppie esterne. Una forma statica, d'altra parte, rimarrà al suo posto, o seguirà il movimento del suo genitore nella gerarchia della scena. Per creare la parte dinamica bisogna innanzitutto, per comodità di lavoro, crearsi una nuova scena dentro CoppeliaSim nella quale porteremo una forma alla volta. Si avvia il *Toggle shape edit mode* e si selezionano tutti i triangoli che compongono la forma, poi bisogna scegliere quale tra le quattro operazioni di estrazione selezionare (vedasi 5). In base alla forma geometrica della figura scegliamo che tipo di forma pura estrarre.

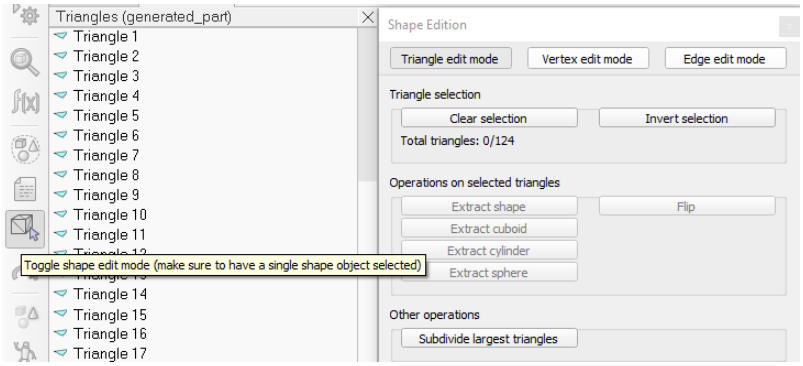


Figura 5: *Toggle shape edit mode*

Ogni forma estratta avrà lo stesso nome della controparte statica ma si aggiunge, sempre per comodità pratiche, il suffisso "dyn". Poiché le forme di partenza sono pure, la parte statica a livello geometrico sarà uguale alla parte dinamica. Il procedimento si ripete per tutte le forme restanti. Fatto ciò, si allega ogni forma statica alla controparte dinamica. Lo si può fare selezionando in contemporanea prima la forma visibile, poi quella invisibile, e poi Barra dei menu -> Modifica -> Rendi l'ultimo oggetto selezionato genitore. Lo stesso risultato può essere ottenuto trascinando la forma visibile sulla corrispettiva parte dinamica nella gerarchia della scena. Una volta create le forme, bisogna configurarle come dinamiche e rispondenti. Lo si fa nelle proprietà della dinamica delle forme, contrassegnando con un flag *"Body is respondable"* e *"Body is dynamic"* (vedasi figura 6). Poi si fa calcolare automaticamente al software le proprietà della massa e dell'inerzia facendo clic su "Calcola proprietà massa e inerzia per le forme selezionate", inoltre si devono abilitare i primi quattro flag *"Local respondable mask"* e disabilitare gli ultimi 4. Affinché i collegamenti, abilitati *respondable*, consecutivi non generino alcuna risposta di collisione quando interagiscono tra loro, al successivo collegamento dinamico vengono disabilitati i primi quattro flag *"Local respondable mask"* e abilitati gli ultimi quattro. La procedura è stata ripetuta anche per le restanti componenti dinamiche alternando

però i *"Local respondable mask"* abilitati. Una volta completata questa parte, bisogna capire quale sarà la base del robot per poterla abilitare diversamente. Nel caso del *Ball on Plate* la base è l'asta, creata con una forma pura di tipo cuboide. A questa forma verranno deselezionati i due flag *"Body is respondable"* e *"Body is dynamic"* (vedasi figura 6). Questa scelta si fa poiché una volta che i pezzi verranno collegati tutti assieme si vuole che l'intera struttura non crolli a causa delle forze esterne del simulatore, quindi se la base non è soggetta a tali forze questa rimarrà immobile tenendo l'intera struttura in piedi.

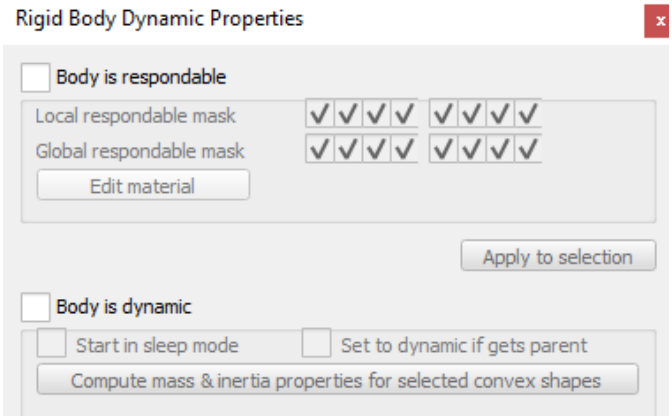


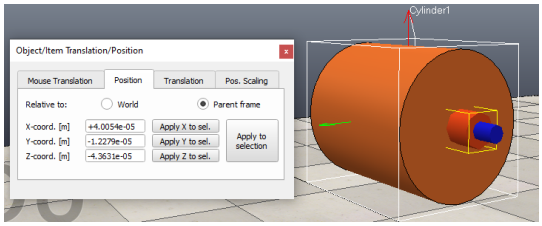
Figura 6: Proprietà dinamiche di una forma

Eseguendo la simulazione, tutte le forme dinamiche, ad eccezione della base del robot, dovrebbero cadere. Le forme visive attaccate (le forme statiche) seguiranno le controparti dinamiche. (tratto da [9])

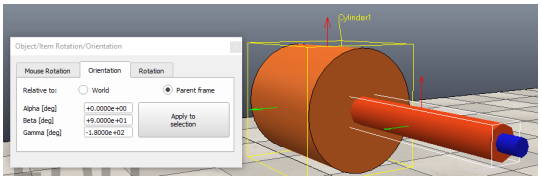
3.6 I joint

Per poter ottenere il movimento rotatorio e fornire due gradi di libertà alla piattaforma, si è fatto uso di due *revolute joint*. Questi sono stati aggiunti nel seguente modo: Barra dei menu -> Aggiungi -> Giunto -> Revolute. Prendendo uno dei due *revolute joint* e tenendolo selezionato si premono i tasti *control-left click* sulla forma del cilindro, statica o dinamica è indifferente. A questo punto nella finestra di dialogo Posizione, nella scheda Posizione, bisogna selezionare "Applica alla selezione": questo sostanzialmente copia la posizione x/y/z del cilindro sul giunto (vedasi fiugra 7 (a)).
Analogamente per l'orientamento, nella finestra di dialogo di orientamento, nella scheda orientamento, bisogna selezionare "Applica alla selezione". Entrambe le posizioni e gli orientamenti sono ora identici (vedasi figura 7 (b)).
Lo stesso procedimento si ripete anche per il secondo *joint* con la seconda forma cilindrica. Poiché nell'ultima fase del progetto si andranno a collegare tutti i pezzi, conviene far sì che ogni forma dinamica abbia stessa posizione e orientazione della forma statica. Quindi lo stesso procedimento appena applicato per la forma con il *joint*, è stato fatto anche per tutte le forme statiche con quelle dinamiche. Ai *joint* verrà assegnato un livello di

visibilità diverso sia dalle forme statiche che da quelle dinamiche. In fase finale della progettazione del macchinario, il livello di visibilità di tutto quello che non è una forma statica verrà disabilitato per rendere il macchinario più semplice a livello estetico. Inoltre, facendo cosi, ogni volta che si deve lavorare con un determinato insieme di oggetti, come ad esempio i *joint*, si disabilitano gli altri livelli di visibilità, per consentire di lavorare solamente con gli elementi desiderati.



(a) *Stessa posizione per il joint e il cilindro*



(b) *Stesso orientamento tra il joint e il cilindro*

Figura 7

3.7 Definzione del modello

Per la parte finale del *Ball on Plate* oltre alle forme e ai giunti verranno utilizzati un sensore di forza e un sensore di visione. Nella gerarchia della scena si parte con l'ultimo elemento da collegare per poi salire verso l'alto arrivo al primo che è la base del modello. Quindi il sensore di visione e la sfera saranno collegati al piatto, il *parent*, come *child*. Ogni elemento, tranne la base, saranno *parent* per gli elementi sottostanti e *child* per gli elementi soprastanti. Il piatto è stato collegato, tramite un sensore di forza, al primo cilindro. Quindi il piatto è *child* per il sensore di forza il quale a sua volta è *child* per il secondo cilindro. Quest'ultimo elemento a sua volta è stato collegato, come *child*, al secondo *revolute joint* che a sua volta è stato collegato, come *child*, al primo cilindro. Infine il primo cilindro è stato collegato al primo *joint* e quest'ultimo alla base (vedasi figura 8).

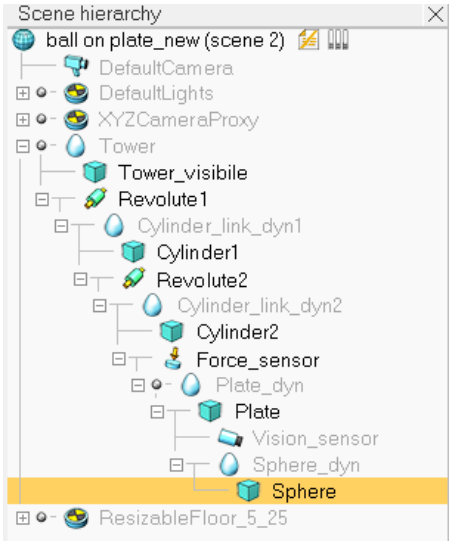


Figura 8: Gerarchia della scena - collegamento parent/child tra gli oggetti

Una volta fatti i collegamenti nella gerarchia della scena bisogna inoltre collegare virtualmente i vari elementi anche nella scena di CoppeliaSim. La base, che nella gerarchia della scena è stata chiamata *Tower*, è stata posizionata perpendicolarmente al pavimento. A

una determinata altezza è stato posizionato il primo cilindro con la base perpendicolare alla *Tower* e parallelo al pavimento. Il secondo cilindro è stato posizionato perpendicolarmente al primo (vedasi figura 9). Essendo i due cilindri ortogonali tra loro, si è riusciti cosi a fornire i due gradi di libertà al macchinario. Poiché i due *revolute joints*, connessi ai cilindri, ruotando intorno al proprio asse z, genereranno un movimento di rotazione intorno all'asse x e intorno all'asse y del sistema di riferimento della piattaforma.

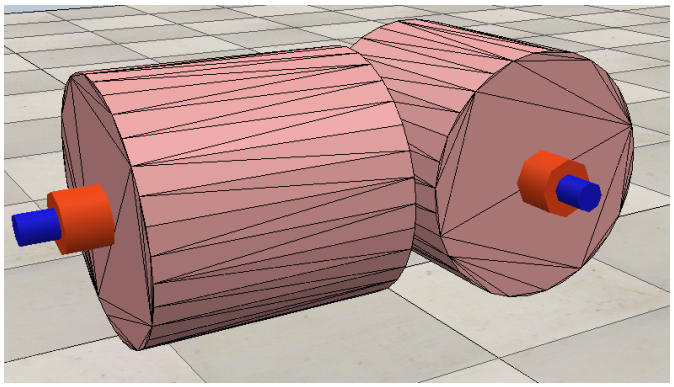


Figura 9: I due *revolute joint* posizionati perpendicolarmente tra di loro

Al centro della superficie laterale del secondo cilindro, sulla parte destra è stato posizionato il sensore di forza, al quale verrà attaccato il piatto. Sul piatto invece è stata posizionata la sfera. A una determinata altezza è stato posizionato il sensore di visione rivolto verso il basso. Poiché il sensore di visione è *child* della piattaforma, questo farà sì che quando la piattaforma inizierà a ruotare, il sensore di visione seguirà in modo solidale la stessa traiettoria. Questo permetterà di rilevare l'immagine della piattaforma come se fosse ferma e la sfera che rotola. Dato che poi sia il piatto che la sfera dovranno essere rilevate dal sensore nelle proprietà comuni dell'oggetto sono state abilitate le proprietà speciali dell'oggetto, cioè *collidable*, *measurable*, *detectable* e *renderable*. Nell'ultima fase della definizione del modello alla base dell'albero della gerarchia, cioè *Tower*, si è abilitato *Object is model base* nelle proprietà comuni dell'oggetto. Questo permette di creare una scatola invisibile, *bounding box*, che racchiude l'intero macchinario all'interno. Per far sì che la scatola abbia un'altezza pari a quella della base, al *vision sensor* si è abilitato il comando *Igno-*

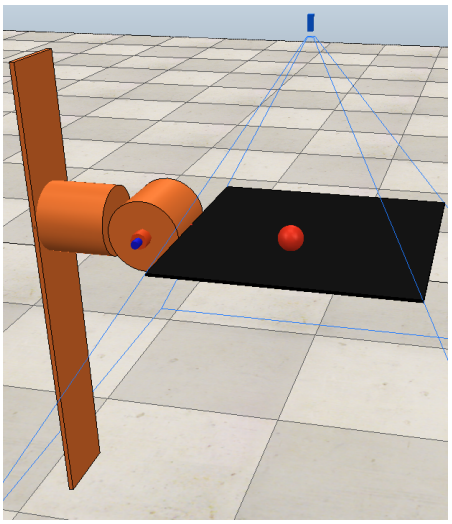


Figura 10: Modello finale del *Ball on Plate*

red by model bounding box, mentre a tutte le forme visibili si è abilitato *Select base of model instead*. Quest'ultimo comando permette di mostrare l'ingombro dell'intero macchinario, quando si seleziona un elemento con tale proprietà abilitata, invece che quello dell'elemento stesso. (tratto da [10])

3.8 Prima simulazione

Quando si esegue la prima simulazione, il macchinario crollerà poiché i *revolute joint* non sono controllati per impostazione predefinita. Quando i due *joint* sono stati creati di default erano configurati in modalità coppia/forza ma il motore era disabilitato. Per poter eseguire una simulazione nella quale il macchinario non cadesse si è abilitato, nelle proprietà dinamiche del *joint*, le proprietà *motor enabled*, *control loop enabled* e poi *position control (PID)* (tratto da [11]). Questa simulazione è stata fatta solo allo scopo di verifica, per vedere se con i vari motori di simulazione del software il controllore PID riusciva a mantenere il *Ball on Plate* nella sua posizione, senza che quest'ultimo crollasse come è successo nella prima simulazione.

3.9 Dimensioni del modello

Il modello creato non ha seguito un modello ben preciso dal quale prendere le misure per i vari elementi utilizzati. Però si è cercato comunque di creare un modello che rispecchi le dimensioni reali. Gli elementi più importanti sono la piattaforma che ha una base di 0.25 x 0.25 metri con uno spessore di 2.5 mm, mentre la sfera ha un diametro di 1.5 cm. Per comprendere meglio tali misure basti pensare che un foglio A4 ha una base del 0.2 x 0.3 m mentre una pallina da ping pong ha un raggio di 2 cm. I due cilindri hanno il raggio uguale all'altezza pari a 7.5 cm. L'ultimo elemento l'asta invece ha un'altezza pari a 0.5 m con uno spessore pari a 5 mm e una larghezza di 7.5 cm.

4 Collegamento con CoppeliaSim

4.1 Remote Api

Come detto prima CoppeliaSim permette di essere controllato da remoto grazie a funzioni chiamate *remoteAPI (Application Program Interface)* oppure API remota. L'API remota consente di controllare una simulazione (o il simulatore stesso) da un'applicazione esterna. Questa infatti può consentire a una o più applicazioni esterne di interagire con CoppeliaSim in modo sincrono o asincrono (asincrono per impostazione predefinita) ed è supportato anche il controllo remoto del simulatore (ad esempio caricamento remoto di una scena, avvio, pausa o interruzione di una simulazione). La modalità sincrona viene utilizzata per garantire che ogni passaggio di simulazione viene eseguito in sincronia con l'applicazione API remota, ovvero il simulatore attenderà un segnale di *trigger* dal client per far avanzare la simulazione per un intervallo di tempo prestabilito (uno *step* di simulazione) per poi attendere il successivo segnale di *trigger*. La funzionalità API remota è disponibile in 2 entità separate, che interagiscono tramite la comunicazione socket: il lato client (ovvero l'applicazione) (tratto da [12]) e il lato server (ovvero CoppeliaSim) (tratto da [13]). Per abilitare il lato client per prima cosa bisogna procurarsi alcuni file, i quali si trovano nella directory di installazione di CoppeliaSim. Bisogna trovare la cartella *remoteApiBindings* e poi scegliere il linguaggio di programmazione con il quale si desidera creare il programma client. In questo caso si è scelto Matlab, e dunque i file saranno i seguenti:

- remApi.m
- remoteApiProto.m
- remoteApi.dll (Windows), remoteApi.dylib (Mac) o remoteApi.so (Linux)

I tre file dovranno essere inseriti in una cartella dentro la quale poi verrà salvato il codice Matlab. Bisogna inoltre assicurarsi che il client usi la stessa architettura di bit della libreria remoteApi: Matlab a 64 bit con libreria remoteApi a 32 bit non funzionerà, e viceversa. L'API remoto sul lato client consiste in un insieme di funzioni che permettono l'interazione del programma client con il modello in CoppeliaSim, e i tre file serviranno rispettivamente per richiamarsi gli oggetti utilizzati dentro il progetto creato con CoppeliaSim, definire le operazioni che posso essere effettuate dentro il simulatore e la compatibilità con il sistema operativo utilizzato. L'abilitazione invece sul lato server dell'API remota può essere effettuata in due modi diversi.Il primo è quello del servizio server dell'API remoto continuo. All'avvio di CoppeliaSim il plug-in API remoto proverà a leggere un file di configurazione denominato remoteApiConnections.txt e, in base al suo contenuto, avvia i servizi server appropriati. Con questo metodo le funzioni API remota verranno sempre eseguite sul lato server, anche se la simulazione non è in esecuzione. Il secondo metodo invece è quello del servizio server API remoto temporaneo. L'utente ha il controllo all'avvio o all'arresto del servizio, però quando un servizio server API remoto temporaneo viene avviato da uno script di simulazione, il servizio verrà automaticamente arrestato al termine della simulazione. Ciò significa che se si vuole creare un collegamento

tra CoppeliaSim e un'applicazione esterna bisogna inserire alcune righe di codice dentro la scena in CoppeliaSim per permettere tale collegamento. (paragrafo tratto da [14])

4.2 Proprietà delle funzioni

Una funzione API remota presenta due principali caratteristiche:

- restituisce un valore a potenza di 2, chiamato return code o codice di ritorno, e bisogna interpretare correttamente il valore restituito per capire se la funzione viene eseguita correttamente oppure ci sono degli errori. I valori dei return code possono essere seguenti:

```
simx_return_ok ( 0 ) tutti i bit a 0
       La funzione è stata eseguita correttamente
simx_return_novalue_flag ( 1 ) bit 0 o LSB (bit meno significativo)
       Non c'è  risposta al comando nel buffer di input. Questo non deve
       essere sempre considerato un errore, a seconda della modalità
       operativa selezionata
simx_return_timeout_flag ( 2 ) bit 1
       Timeout della funzione (probabilmente la rete è inattiva o troppo
       lenta)
simx_return_illegal_opmode_flag ( 4 ) bit 2
       La modalità operativa specificata non è supportata per la
       funzione specificata
simx_return_remote_error_flag ( 8 ) bit 3
       La funzione ha causato un errore sul lato server (ad esempio, è
       stato specificato un handle non valido)
simx_return_split_progress_flag ( 16 ) bit 4
       Il thread di comunicazione sta ancora elaborando il precedente comando
simx_return_local_error_flag ( 32 ) bit 5
       La funzione ha causato un errore sul lato client
simx_return_initialize_error_flag ( 64 ) bit 6
       simxStart non era ancora stato chiamato
```

  Nel caso di più bit "allo stato alto" contemporaneamente, sono attive tutte le situazione di errore ad essi corrispondenti. Ad esempio se avessi 00000011 otterrei sia il messaggio d'errore del bit 0 sia del bit 1.

- la maggior parte delle funzioni API remote richiedono due argomenti aggiuntivi: la modalità operativa (operation mode) (tratto da [15]) e l' ID client (identificatore restituito dalla funzione simxStart) (tratto da [16])

Un esempio di una funzione remoteApi è il seguente:

```
[number returnCode] = simxStartSimulation(number clientID,number operationMode)
```

La necessità di una modalità operativa e di un codice di ritorno specifico deriva dal fatto che la funzione API remota deve interagire con il server (CoppeliaSim), eseguire una determinata azione e restituire al client una risposta. La modalità operativa verrà scelta in base all'azione che una *remoteAPI* dovrà eseguire. Un approccio regolare sarebbe di

fare in modo che il client invii una richiesta e attenda che il server elabori la richiesta e risponda: nella maggior parte dei casi ciò richiederebbe troppo tempo e il ritardo penalizzerebbe l'applicazione client. Invece, l'API remota consente all'utente di scegliere il tipo di modalità operativa e il modo in cui la simulazione avanza fornendo quattro meccanismi principali per eseguire chiamate di funzione o per controllare l'avanzamento della simulazione:
Chiamata di una funzione bloccante (*Blocking function calls*): pensata per situazioni in cui non possiamo permetterci di non attendere una risposta dal server. Questa tipologia di modalità operativa di solito viene utilizzata per richiamarsi gli oggetti, poiché per poter comandare ad esempio un joint bisogna prima averlo richiamato dentro Matlab. Un esempio di questa modalità è il seguente

```
[ret1,Joint1]=vrep.simxGetObjectHandle(clientID,'Revolute1',vrep.simx_opmode_blocking);
[ret2,Joint2]=vrep.simxGetObjectHandle(clientID,'Revolute2',vrep.simx_opmode_blocking);
 % ret1 e ret2 sono i valori dei return code della prima e della seconda stringa
 di codice rispettivamente;
 % Joint1 e Joint2 sono i nomi assegnati dal utente dentro Matlab riferiti ai veri nomi
 dei joint utilizzati dentro il progetto, chiamati rispettivamente Revolute1 e Revolute2
```

Le uniche modalità bloccanti sono:

```
simx_opmode_blocking e simx_opmode_oneshot_wait
```

Chiamata di una funzione non bloccante (*Non-blocking function calls*): pensata per situazioni in cui si vuole semplicemente inviare dati a CoppeliaSim senza la necessità di una risposta. In alcune situazioni, è importante inviare vari dati all'interno di uno stesso messaggio, al fine di avere quei dati applicati contemporaneamente sul lato server (ad esempio, si vuole che i 3 giunti di un robot vengano applicati al suo Modello CoppeliaSim nello stesso istante di tempo, ovvero nello stesso passaggio di simulazione). In tal caso, l'utente può interrompere temporaneamente il thread di comunicazione per ottenere ciò tramite il comando simxPauseCommunication(IDcliente, 1) inserisce le operazioni che si vogliono effettuare in contemporanea e chiude con il comando simxPauseCommunication(IDcliente, 0).

```
vrep.simxPauseCommunication(clientID,1);
         [ret_velocity1]=vrep.simxSetJointTargetVelocity(clientID,Joint1,1000,...
         vrep.simx_opmode_oneshot);
         [ret_force1]=vrep.simxSetJointForce(clientID,Joint1,5,...
         vrep.simx_opmode_oneshot);
vrep.simxPauseCommunication(clientID,0);
```

Le modalità non bloccanti più utilizzate sono le seguenti:

```
simx_opmode_oneshot:
Il comando viene inviato e viene restituita una risposta precedente
allo stesso comando (se disponibile). La funzione non attende la risposta effettiva.
simx_opmode_buffer:
Viene restituita una risposta precedente allo stesso comando (se disponibile).
Il comando non viene inviato, e la funzione non attende una risposta effettiva.
```

Streaming di dati (*Data streaming*): il server può prevedere il tipo di dati richiesti dal client. Affinché ciò accada, il client deve segnalare questo al server con un flag di modalità di funzionamento *streaming* (ovvero la funzione è memorizzata sul lato server, viene eseguita e inviata su base temporale regolare, senza la necessità di una richiesta dal client). L'unica modalità *streaming* è la seguente:

```
simx_opmode_streaming
```

Funzionamento sincrono (*Synchronous operation*): la simulazione, di default, avanza senza tenere conto dell'avanzamento del client API remoto e le chiamate delle funzioni remoteAPI verranno eseguite in modo asincrono per impostazione predefinita. Se pero si vuole sincronizzare il client API remoto e l'avanzamento della simulazione, controllandolo dal lato client, bisogna utilizzare la modalità sincrona. In tal caso, il servizio server deve essere pre abilitato per il funzionamento sincrono. Ciò può essere ottenuto tramite il file di configurazione del servizio server API remoto continuo remoteApiConnections.txt, e quindi fatto in automatico dal server se ci si collega con un numero di porta specifico. Il comando da inserire nel codice Matlab per creare il collegamento è il seguente.

```
simxSynchronous(clientID,boolean enable)
```

In questo caso il valore booleano sarà true in quanto si vuole abilitare la modalità sincrona. Quando però viene adottata tale modalità di collegamento bisogna utilizzare i seguenti comandi:

```
simxSynchronousTrigger (clientID);
simxGetPingTime (clientID);
```

Il comando di Trigger serve per permettere l'avanzamento della simulazione. Quando viene chiamato il comando simxSynchronousTrigger, il passaggio di simulazione successivo inizierà l'elaborazione. Ciò non significa che quando ritorna la chiamata di funzione, il prossimo passaggio di simulazione avrà terminato il calcolo. Per questo motivo bisogna assicurarsi di leggere i dati corretti. Per questo motivo viene introdotto il comando simxGetPingTime, cosi si evita di leggere i dati del passo di simulazione sbagliato.

```
   simxSynchronous (IDcliente, true); // Abilita la modalità sincrona
   simxStartSimulation (IDcliente, simx_opmode_oneshot);

//Il primo passo di simulazione attende un trigger prima di essere eseguito

simxSynchronousTrigger (IDcliente); // Attiva il passaggio di simulazione successivo

//Il primo passo di simulazione è ora in esecuzione

simxGetPingTime (IDcliente); // Dopo questa chiamata, il primo passaggio della
                             // simulazione è terminato

// Ora si possono leggere in sicurezza tutti i dati nell'ordine corretto
```

L'ultima caratteristica delle remote Api è il clientID, un identificatore restituito dalla funzione simxStart. Questo valore dovrà essere inserito in ogni Api remoto poiche grazie ad esso viene creato il collegamento vero e proprio tra client e server.

```
[clientID]= simxStart(string connectionAddress,number connectionPort,
            boolean waitUntilConnected,boolean doNotReconnectOnceDisconnected,
            number timeOutInMs,number commThreadCycleInMs)
```

Connection address: è l'indirizzo del computer su cui è in esecuzione il server. Se il server è in esecuzione sullo stesso computer del client, è possibile utilizzare l'indirizzo "127.0.0.1".
Connection port: se ci si connette sulla porta 19997 allora sul ci sarà sul lato server un servizio API remota continua già in esecuzione e pre-abilitato per la modalità sincrona, se invece ci si collega sulla porta 19999 bisogna inserire dentro il lato server il seguente comando: *simExtRemoteApiStart(19999)*. Questo comando dovrà essere inserito, sul lato server, in un *associated child script non-threaded*.
Wait unitl connected : valore booleano che se vero allora la funzione si blocca fino a quando non viene connessa.
Do not reconnect once disconnected : anche questa è una funzione booleana che se vera la comunicazione non verrà tentata una seconda volta in caso di perdita di una connessione.
Time out in Ms : è il time out per il primo tentativo di connessione e ha un valore consigliato di 5000 millisecondi.
Comm thread cycle in Ms : indica la frequenza con cui i pacchetti di dati vengono inviati avanti e indietro. Ridurre questo numero migliora la reattività e si consiglia un valore predefinito di 5.
Bisogna inoltre ricordarsi che a ogni chiamata *simxStart* riuscita bisogna aggiungere alla fine del codice un comando *simxFinish* per chiudere la connessione. Il valore che Matlab restituisce quando viene eseguito il comando può essere o il codice identificativo o il valore -1. Quest'ultimo valore viene restituito nel caso in cui la connessione con il server non è stata possibile, è stato raggiunto il Timeout ad esempio. (tratto da [17])

4.3 Esempio di codice

Per avere dunque un'idea più chiara di come funziona tale collegamento, il seguente codice esegue una sequenza di istruzioni semplici con lo scopo di azionare uno dei due *joint*. Per prima cosa viene caricata la libreria *remoteApiProto.m*. Da adesso ogni volta che si farà uso di una funzione Api remota allora bisogna ricordarsi di aggiungere il prefisso *vrep.* prima di ogni chiamata *simx* cosi da poter utilizzare tali funzioni. Poi nel caso in cui ci fossero connessioni già aperte con CoppeliaSim allora verranno chiuse tramite il comando *simxFinish(-1)* e infine viene creato il collegamento sulla porta 19997. Ora se l'istruzione *if(clientID > -1)* è verificata significa che il collegamento è andato a buon fine e si possono dunque richiamare dentro Matlab gli oggetti con i quali si vuole interagire. In questo caso si vuole controllare solamente il *Joint2*, cioè quello più vicino alla piattaforma. Il comando *simxGetObjectHandle* dunque ci permette di richiamare l'oggetto *Revolute2*, nome originale del *joint* assegnato dentro la scena con il progetto, e ci restituisce un *return code* con valore ret1, e un nome dell'oggetto che in questo caso è *Joint2*, questi due valori vengono definiti direttamente dall'utente. Tale nome inoltre sarà quello che verrà utilizzato nei comandi successivi come ad esempio nel comando

simxGetJointPosition il quale richiede il nome dell'oggetto al quale si vuole restituire la posizione e restituisce anch'esso un *return code* chiamato ret2 e la posizione del *joint* chiamata con la variabile *position*. Bisogna inoltre prestare attenzione che nella parte client, le chiamate della API restituiscono angoli in radianti, mentre interagendo direttamente solo con CoppeliaSim dovrà fornire e riceverà misure d'angolo in gradi. Il nome delle variabili di *return code* devono essere diversi cosi da capire, in fase di esecuzione del codice, per ogni istruzione se l'esecuzione è andata a buon fine oppure si sono riscontrato dei problemi. Gli *operation mode* vengono direttamente consigliati dal sito dal quale si prendono le funzioni[16]. Per quanto riguarda la parte di simulazione, si crea il collegamento sincrono grazie al comando *simxSynchronous* e poi tramite il comando *simxStartSimulation* si fa partire la simulazione dentro il software di CoppeliaSim. In un ciclo for si inseriscono i comandi *simxSynchronousTrigger* e *simcGetPingTime* poichè la modalità è sincrona. Quindi all'avanzare di ogni passo dentro il ciclo for corrispoderà un passo di simulazione dentro CoppeliaSim. Prima di assegnare al *joint* una velocità e una forza tramite i rispettivi comandi *simxSetJointTargetVelocity* e *simxSetJointForce*, si richiama nuovamente la posizione e si fa un controllo con una posizione minima imposta dall'utente. Se la posizione del *joint* è inferiore alla posizione minima, in questo caso $-\frac{\pi}{4}$, allora si uscirà dal ciclo for tramite un comando *break* e la simulazione terminerà grazie ai comandi *simxStopSimulation* e *simxFinish(clientID)*, altrimenti verrà assegnato al *joint* una velocità e una forza in modo da farlo muovere, e con esso la piattaforma, fino a quando tale posizione minima non verrà superata.

```
function Jointdueversoilbasso()
   disp('Program started');
   vrep=remApi('remoteApi'); % using the prototype file (remoteApiProto.m)
   vrep.simxFinish(-1); % just in case, close all opened connections
   clientID=vrep.simxStart('127.0.0.1',19997,true,true,5000,5);
   if (clientID>-1)
       disp('Connected to remote API server');

       [ret1,Joint2]=vrep.simxGetObjectHandle(clientID,'Revolute2',...
           vrep.simx_opmode_blocking);
       [ret2,position]=vrep.simxGetJointPosition(clientID,Joint2,...
           vrep.simx_opmode_streaming);
       position_min = -pi/4;

       vrep.simxSynchronous(clientID,true);
       vrep.simxStartSimulation(clientID,vrep.simx_opmode_blocking);
       t = 1:50;
          for i=1:length(t)
           vrep.simxSynchronousTrigger(clientID);
           [retPing,pingTime]=vrep.simxGetPingTime(clientID);
           [retPos,position]=vrep.simxGetJointPosition(clientID,Joint2,...
               vrep.simx_opmode_buffer);

           if (position < position_min)
               disp('stop')
               break
           end
```

```
            vrep.simxPauseCommunication(clientID,1);
            [ret_velocity1]=vrep.simxSetJointTargetVelocity(clientID,...
                Joint2,-1000,vrep.simx_opmode_oneshot);
            [ret_force1]=vrep.simxSetJointForce(clientID,Joint2,5,...
                vrep.simx_opmode_oneshot);
            vrep.simxPauseCommunication(clientID,0);

           end

        vrep.simxStopSimulation(clientID,vrep.simx_opmode_blocking);
        % Now close the connection to V-REP:
        vrep.simxFinish(clientID);
       else
        disp('Failed connecting to remote API server');
    end
    disp('Program ended');
end
```

5 Reinforcement Learning

5.1 Introduzione

Il Reinforcement Learning (RL) è una tecnica di apprendimento automatico che punta a realizzare agenti autonomi in grado di scegliere azioni da compiere per il conseguimento di determinati obiettivi tramite interazione con l'ambiente in cui sono immersi. L'agente deve scegliere le azioni migliori da eseguire nello stato in cui si trova l'ambiente, con l'obiettivo di massimizzare il punteggio che gli verrà fornito per ogni azione (viene introdotto cosi il concetto di ricompensa o *reward*). Quando si utilizza RL, al computer viene fornito l'obiettivo da portare a termine, esso dovrà imparare a raggiungere tale obiettivo tramite interazioni di tentativi ed errori con l'ambiente. Per capire meglio il funzionamento di tale metodo di apprendimento si consideri come obiettivo quello di imparare ad andare in bicicletta senza cadere. Supponendo che al primo tentativo, il sistema dopo aver eseguito una serie di azioni si trovi nella condizione di essere sulla bici inclinata di 45 gradi verso destra. Le azioni possibili sono due girare il manubrio verso destra o verso sinistra. Se l'agente decide che l'azione da eseguire sia quella di girare verso destra, allora il primo tentativo è destinato a fallire poiché la bici cadrà. Si supponga però che nell'esperimento successivo il sistema si trovi nella stessa situazione, ora però l'agente decide di girare il manubrio verso sinistra. Anche in questo caso però l'esperimento è destinato a fallire poiché ormai l'inclinazione è tale da non poter più ritornare nella condizione di equilibrio. Questo porterà inevitabilmente a un altro fallimento. A questo punto l'agente capisce che la condizione in sé, cioè di trovarsi a 45 gradi inclinato verso destra, è fallimentare a prescindere e quindi nelle prossime simulazioni proverà a evitare tale situazione. Continuando con i tentativi l'agente apprenderà quali devono essere le azioni migliori per cui l'obiettivo proposto venga raggiunto (tratto da [18] pagine 1 e 2).

5.2 Caratteristiche del Reinforcement Learning

L'agente quando deve interagire con l'ambiente, in base alla situazione in cui si trova, sceglierà un'azione da eseguire. Tale azione cambierà la status dell'ambiente e ciò verrà comunicato all'agente sotto forma di segnale scalare di rinforzo. Le tre caratteristiche principali del Reinforcement Learning sono dunque l'ambiente (*environment*), la funzione di rinforzo e la funzione valore (*value function*) (tratto da [18] pagina 2).

Ambiente

Come si è spiegato prima, ogni sistema apprende una mappatura dalle situazioni alle azioni tramite le interazioni di tentativi ed errori con un ambiente dinamico. L'*environment* deve essere osservabile dal sistema di apprendimento e le osservazioni possono essere sotto forma di descrizione simbolica o lettura dei sensori. L'osservazione dunque fornirà le informazioni relative allo stato in cui si trova l'ambiente in quel preciso istante. Se il sistema riesce a osservare perfettamente l'ambiente allora questo influenzerà la scelta dell'azione da eseguire. Le azioni possono essere di basso livello, come ad esempio imprimere una forza, scegliere una velocità o una tensione da applicare oppure di alto livello, come ad esempio istruire un bot a pensare come un essere umano. L'ambiente dunque

deve essere una descrizione quanto più accurata possibile del sistema reale che si vuole analizzare. Tornando all'esempio della bicicletta, l'ambiente dovrebbe descrivere le forze che sono applicate, come ad esempio la gravità e l'attrito, o quelle da applicare (tratto da [18] pagina 2).

Funzione di rinforzo

L'obiettivo da raggiungere, chiamato anche goal, è definito utilizzando il concetto di funzione di rinforzo, la quale è esattamente la funzione che l'agente cercherà di massimizzare. Viene creata una mappatura dalle coppie stato/azione e il rinforzo ottenuto. Dopo aver eseguito una determinata una determinata azione all'agente viene fornito un rinforzo (reward o ricompensa) sotto forma di valore scalare. L'agente apprenderà a massimizzare la somma totale dei reward ottenuti dall'inizio alla fine della simulazione. È compito del programmatore descrivere quanto più accuratamente la funzione di rinforzo che descrive il goal al quale l'agente deve arrivare. Possono essere definite funzioni molto complesse ma le più usate sono il *Pure Delayed Reward and Avoidance Problems* e il *Minimum Time to Goal*. La prima tipologia è definita con tutti i rinforzi pari a zero tranne nello stato finale che avrà un valore positivo o negativo. Il segno dello scalare indica se nello stato finale si è raggiunti il goal (segno positivo), e quindi si è ricompensati, oppure se è uno stato da evitare poiché ha portato a un fallimento dell'esperimento (segno negativo), e quindi verrà punito.

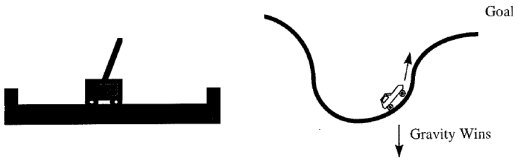


Figura 11: *Cart-pole and Car on the hill (tratto da [18] pagina 3)*

Un esempio nel quale viene applicato tale metodo è quello dello *standard cart-pole problem* o il problema del pendolo inverso. Viene posto in un carrello un'asta incernierata che in base al movimento del carrello, su una pista finita, può inclinarsi verso destra o sinistra. L'obiettivo dell'agente è quello di imparare a bilanciare l'asta in modo tale da rimanere perpendicolare rispetto al piano sul quale è posto il carrello senza inclinarsi. Le uniche azioni valide sono quelle verso destra e verso sinistra. La funzione di rinforzo è zero dappertutto tranne quando alla fine della pista l'asta o è inclinata e quindi riceverà -1 oppure rimane perpendicolare al piano e quindi riceverà +1. Dato che l'agente deve massimizzare il *reward* allora imparerà la sequenza di azioni da applicare per arrivare alla fine della pista senza che l'asta si inclini.

La seconda tipologia invece porta l'agente a trovare il percorso più breve per arrivare a soddisfare la condizione di goal. Un esempio nel quale può essere applicato tale metodo è quello della macchina sulla collina. Una macchina ferma posizionata in mezzo a due colline deve riuscire a salire la collina alla destra. Il problema però è descritto in modo tale da impedire alla macchina, dalla posizione nella quale si trova, di riuscire a salire

direttamente sulla collina. La macchina dunque dovrà percorrere un pezzo in retromarcia sulla prima collina in modo da riuscire, grazie alla spinta, di salire la seconda collina. Dato che l'agente riceverà -1 per ogni volta che percorrerà un tratto che non lo porterà a raggiungere l'obiettivo, e invece gli verrà fornito un *reward* pari a zero una volta raggiunto il goal, esso dovrà apprendere qual è la distanza minima che gli permette di avere una spinta tale da salire, cosi da massimizzare il *reward* finale (tratto da [18] pagina 3).

Funzione Valore

Bisogna capire ancora come fa l'agente ad apprendere a scegliere l'azione giusta, o ancora come misurare l'utilità di una determinata azione. Vengono definiti due nuovi elementi. La *policy*, che determina quale azione dovrebbe essere eseguita in ogni stato, e il valore di uno stato che viene definito come la somma di tutti i *reward* ottenuti quando si parte da tale stato e si segue una *policy* fissa fino allo stato finale. La *policy* ottima è dunque quella che mappa il percorso che massimizza la somma dei rinforzi ottenuti quando si parte da uno stato casuale e si arriva a quello finale, arrivando a soddisfare cosi la condizione di goal. La funzione valore è una mappatura da stati a valore di stato. Per capire meglio questo concetto conviene fare il seguente esempio. Si consideri una griglia 4x4. Ogni casella rappresenta uno stato. La funzione di rinforzo è definita come -1 dappertutto, cioè l'agente riceverà -1 per ogni azione che eseguirà. Le azioni possibili sono 4, cioè ci si può spostare secondo i 4 punti cardinali. L'obiettivo è quello di arrivare nell'angolo in alto a sinistra o in quello in basso a destra, con meno spostamenti possibili.

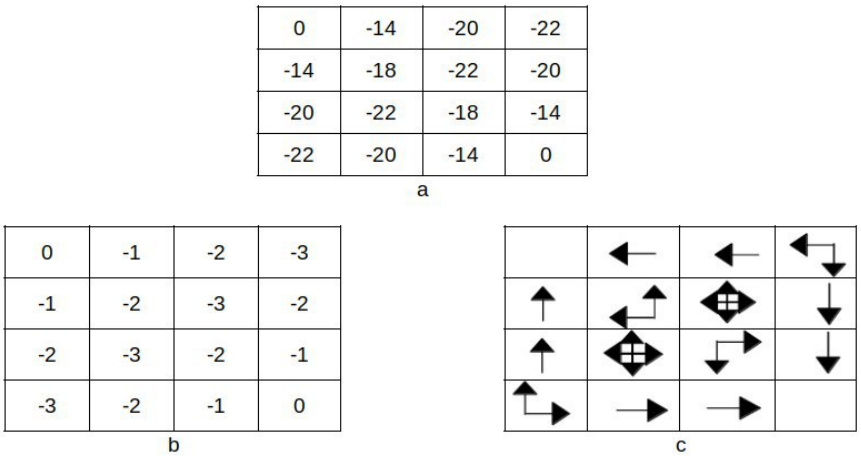


Figura 12: Mappatura stato/azione (tratto da [18] pagina 4)

La funzione valore per una *policy* casuale è mostrata nella figura 12 (a), dove per ogni stato la *policy* ha scelto delle azioni casuali prima di poter arrivare al goal. Il valore numerico indica quante azioni sono servite per soddisfare l'obiettivo, quindi se ad esempio si dovesse partire dall'angolo in alto a destra, servirebbero 22 passi per arrivare in uno dei due punti. Dopo una serie di esperimenti però si viene a creare la funzione valore ottima mostrata in figura 12 (c). Partendo di nuovo dall'angolo in alto a destra però questa volta ci si mettono solo 3 passi. Data la funzione valore ottima, estrarre la *policy*

ottima (vedasi figura 12 (b)), la quale nel minor numero di azioni arriva al goal, diventa banale (tratto da [18] pagina 4).
Capiti i concetti base rimane solo da capire come creare un algoritmo che sia efficiente nel trovare la funzione valore ottima.

5.3 Programmazione Dinamica

Quando un esperimento fornisce risulta fallimentare bisogna capire quale azione ha portato a tale risultato. L'addestramento del pilota automatico di un aereo descrive perfettamente questa situazione. L'agente deve prendere centinaia di decisioni nell'arco di pochi secondi. Se tali decisioni portano però al fallimento, che in questo caso corrisponde al crollo dell'aereo, bisogna capire quale azione tra tutte quelle eseguite è la causa di tale fallimento. Diventa molto difficile quando i problemi assegnati sono complessi e quindi si è dovuto trovare un metodo per facilitare l'apprendimento. La programmazione dinamica è una branca della matematica che facilità tale ricerca ed è basata su due principi. Il primo afferma che se un azione porta immediatamente al fallimento, allora quell'azione è da evitare. Il secondo invece afferma che se tutte le azioni applicate in un determinato stato portano al fallimento allora quello stato è da evitare. Tale principio è fondamentale poiché l'agente ora può apprendere anche senza arrivare a far crollare l'aereo. Quindi se a un certo punto della simulazione, dopo aver eseguito un certo numero di azioni, l'agente sceglie un azione che lo porterà nello stato da evitare, capirà che anche quella azione è da evitare prima ancora di finire l'esperimento. Inizialmente la funzione valore non si avvicina minimamente all'ottimo. Questo significa che la mappatura stato/azione non è valida. Trovare la mappatura corretta porterà poi facilmente ad estrarre la *policy* ottima. Il legame tra il valore ottimo e il valore ottenuto è dato dall'equazione (1) dove la funzione valore approssimata, in funzione dello stato x al tempo t, è data dalla somma dell'errore, dell'approssimazione dello stato, e della funzione valore ottima nello stato x al tempo t.

$$V(x_t) = e(x_t) + V^*(x_t) \tag{1}$$

Dopo avere eseguito un azione si passa dall'istante t a t+1 e quindi l'equazione diventerà.

$$V(x_{t+1}) = e(x_{t+1}) + V^*(x_{t+1}) \tag{2}$$

Il valore dello stato x_t della *policy* ottima è la somma di tutti i rinforzi quando si parte dallo stato x_t e si eseguono azioni ottimali fino al raggiungimento di uno stato terminale. Da questa definizione si capisce che esiste una relazione tra i valori degli stati successivi x_t e x_{t+1}. Questa relazione è descritta dall'equazione di Bellman (3).

$$V^*(x_t) = r(x_t) + \gamma V^*(x_{t+1}) \tag{3}$$

La funzione valore approssimata ha lo stesso tipo di relazione come mostra l'equazione (4).

$$V(x_t) = r(x_t) + \gamma V(x_{t+1}) \tag{4}$$

Viene introdotto il valore numerico γ, cioè il *discount factor*, compreso nell'intervallo [0,1]. Il valore scelto fa sì che i rinforzi immediati abbiano più importanza dei rinforzi futuri, ponderandoli più pesantemente. Il *discount factor* viene utilizzato per decrementare esponenzialmente il peso dei rinforzi futuri. Sostituendo la parte destra delle equazioni (1) e (2) nell'equazione (4) si ottiene l'equazione (5). Sviluppando tale equazione si ottiene la (6).

$$e(x_t) + V^*(x_t) = r(x_t) + \gamma(e(x_{t+1}) + V^*(x_{t+1})) \tag{5}$$

$$e(x_t) + V^*(x_t) = r(x_t) + \gamma e(x_{t+1}) + \gamma V^*(x_{t+1}) \tag{6}$$

Utilizzando ora la (3), il termine $V^*(x_t)$ si semplifica portando cosi alla seguente relazione (7).

$$e(x_t) = \gamma e(x_{t+1}) \tag{7}$$

Il significato dell'equazione (7) può essere spiegato con l'utilizzo di una catena di Markov a 4 stati. L'ultimo stato, identificato con la lettera T, è quello nel quale si conosce il valore dello stato a priori. L'errore in questo stato dovrà essere pari a zero. Questo perché nello stato finale si conoscono i valori che devono essere restituiti, più precisamente saranno due valori scalari, ad esempio +1 se si raggiunge il goal, -1 se l'esperimento è risultato fallimentare. Il processo di apprendimento si può definire come la ricerca della funzione valore approssimata $V(x_t)$ che soddisfa le equazioni (3) e (7) per tutti gli stati x_t. Se l'approssimazione dell'errore nel penultimo stato, cioè il terzo, è un fattore di γ più piccolo dell'errore nello stato T, che per definizione ha errore nullo, allora l'errore nello stato 3 dovrà essere nullo anch'esso. Se l'equazione (7) è vera per tutti gli stati $V(x_t)$, allora l'approssimazione d'errore in ogni stato sarà necessariamente zero, questo implica che $V(x_t) = V^*(x_t)$ per tutti gli $V(x_t)$. Il processo di apprendimento è dunque un processo nel quale si cerca una soluzione per l'equazione (4) per tutti gli stati x_t, la quale sarà una soluzione anche per l'equazione (7).

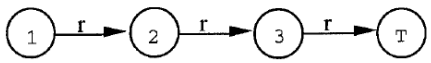


Figura 13: Catena di Markov a 4 stati

Value Iteration

La funzione valore approssimata può essere immaginata come una tabella di ricerca in continuo aggiornamento, dove ogni a ogni stato corrisponde un valore approssimato. Per trovare il valore ottimo viene eseguita una ricerca per la miglior coppia stato/azione che andrà ad aggiornare il valore, dello stato nella tabella, continuamente secondo l'equazione (8). Quando tale valore converge, e non si trova una coppia stato/azione migliore, allora si è arrivati all'ottimo.

$$\Delta w_t = \max_u(r(x_t, u) + \gamma V(x_{t+1})) - V(x_t) \tag{8}$$

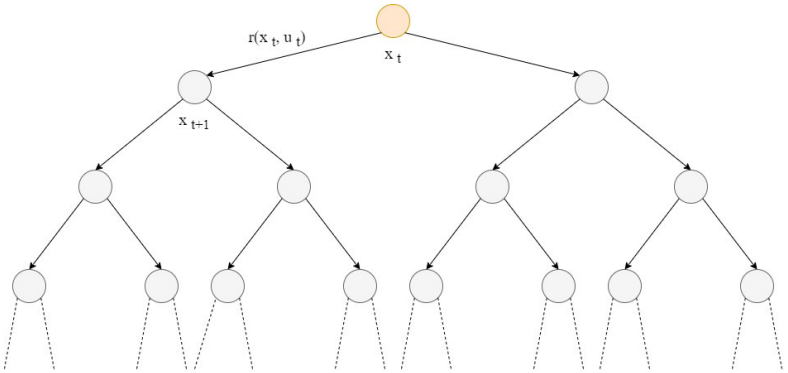


Figura 14: Stato x_t con due diversi stati futuri x_{t+1} (tratto da [18] pagina 10)

Nell'equazione (8) u rappresenta l'azione eseguita nello stato x_t che porta una transizione nello stato x_{t+1} e $r(x_t, u)$ rappresenta il reward ottenuto per aver eseguito tale azione.
La figura 14 mostra un esempio nel quale basta un solo aggiornamento per trovare l'ottimo. Dallo stato x_t ci sono solo due possibili scelte che porteranno entrambe in uno stato x_{t+1} diversi tra loro. Solo uno fornirà il valore ottimo. L'unico modo però per stabilire quale azione porterà nello stato migliore è quello di calcolare la somma di tutti i reward accumulati, utilizzando un *discount factor* adatto, durante una simulazione e il valore approssimato del successivo stato $V(x_{t+1})$. Questo procedimento deve essere fatto per ogni possibile azione u nello stato x_t. Non è possibile dallo stato x_t eseguire un azione, vedere il valore ricevuto una volta arrivato nello stato x_{t+1}, tornare indietro ed eseguire l'altra azione. Bisogna portare a termine la simulazione prima, in base ai risultati ottenuti scegliere quale azione ha portato il massimo valore. L'equazione (8) non è altro che l'equazione di Bellman generalizzata per i Markov decision processes (processi decisionali di Markov - MDP), dove Δw_t non è altro che l'errore $e(x_t)$, ogni nuovo aggiornamento nella tabella di ricerca porterà a una riduzione del valore dell'errore. Se il numero di aggiornamenti tende all'infinito questo porterà a $e(x_t) = 0$, tale risultato soddisferà l'equazione (4) e quindi $V(x_t) = V^*(x_t)$. Ciò significa che l'apprendimento è finito.
A differenza della catena di Markov dove è possibile effettuare una singola azione in un determinato stato, nel processo decisionale di Markov ci sono molteplici azioni in un dato stato (figura 12). Tale processo è caratterizzato da un insieme di stati X, un insieme di stati iniziali S, il quale è un sottoinsieme di X, un insieme di azioni A, una funzione di rinforzo R dove R(x,a) è il rinforzo che si si aspetta eseguendo l'azione a nello stato x e in fine un modello P dove P(x'|x,a) fornisce la probabilità che eseguendo l'azione a nello stato x questa ci porterà nello stato x'. Un requisito è che la scelta dell'azione dipenda solamente dall'osservazione dello stato attuale x. Se la conoscenza di azioni o stati precedenti influisce sulla scelta dell'azione corrente, allora il processo di decisione non è di Markov (tratto da [18] pagine 5-8).

5.4 L'esplorazione

Come già spiegato nei paragrafi precedenti, un algoritmo di apprendimento buono è uno che riesce a trovare la *value function* in un intervallo di tempo ragionevole. Facendo ciò verrà soddisfatta l'equazione di Bellman. La *value function* ottima dovrà dunque soddisfare tale equazione per tutti gli stati x_t nello spazio degli stati. Questo porta all'introduzione di un nuovo concetto, quello dell'esplorazione. L'esplorazione è definita come la scelta intenzionale di eseguire una determinata azione, che non è considerata la migliore, solo per lo scopo di acquisire conoscenza di stati ancora non visti. Gli algoritmi di apprendimento iniziano l'addestramento con una prima fase di esplorazione, nella quale si impostano già alcune basi su quali siano le azioni buone da eseguire e quali no. Per poter trovare una soluzione approssimativamente ottima, lo spazio degli stati dovrà essere esplorato a sufficienza (tratto da [18] pagina 15).

6 L'utilizzo di RL con Matlab

Dalla versione 19a Matlab permette di utilizzare i tool di Reinforcement Learning, mettendo a disposizione una serie di algoritmi da utilizzare. In questo capitolo verrà trattato l'utilizzo che si è fatto di tale tool e verrà descritto il funzionamento dell'algoritmo utilizzato.

6.1 L'ambiente utilizzato

Come spiegato nel capitolo precedente affinché un algoritmo di apprendimento funzioni al meglio è necessario descrivere il sistema da analizzare il meglio possibile. L'ambiente creato per il Ball on Plate è un codice che servirà a impostare le varie fasi della simulazione. In questo paragrafo verrà spiegato il codice in ordine di funzionamento e non in ordine di scrittura, ciò servirà per facilitarne la comprensione. Per creare la struttura dell'ambiente si è fatto uso di un modello utilizzato per la realizzazione dell'ambiente del cart and pole, messo a disposizione direttamente da MathWorks, azienda che realizza e vende Matlab. Inserendo nel command window di Matlab *rlCreateEnvTemplate(nomeDellaClasse)* verrà creato un *template* per una classe, a partire dalla classe astratta *rl.env.MATLABEnvironment*. Il *template* è stato preso come punto di riferimento per la realizzazione dell'*environment* usato per il *Ball on Plate*, ed è divisto in tre parti principali:

- Proprietà dell'ambiente
- Metodi richiesti per l'ambiente
- Metodi opzionali

Nella sezione delle proprietà dell'ambiente vengono specificati i parametri necessari per la creazione e simulazione dell'ambiente. Tra questi parametri si possono trovare:

```
properties

        %variabili a b che servono al calcolo della posizione casuale iniziale
        %della pallina
        %sono in pixel e dovrebbero andare da 0 a 128 perch ho scartato i
        %primi 13 e gli ultimi 13 cosi la pallina si pu posizionare poi in un quadrato
        %di 0.4x0.4 e non 0.5x0.5
        a = 13;
        b = 115;

        %valori posizione e velocità pallina goal
        xgoal = 0;
        ygoal = 0;
        xvgoal = 0;
        yvgoal = 0;

        %limite della pallina oltre al quale la simulazione andrebbe fermata
        limit_ball = 0.12;
```

```
%limite dei due motori che impedisce di portare il piatto in
%una posizione estrema
theta_limit = pi/6;


%varibili vuote che poi mi aiuteranno a richiamare gli oggetti
%in altre funzioni
var_vrep = [];
var_clientID = [];
var_joint1 = [];
var_joint2 = [];
var_pallina = [];
var_piatto = [];
var_camera = [];
var_pb0x = [];
var_pb0y = [];
var_figure = [];
var_x0 = [];
var_y0 = [];
var_xnew = [];
var_ynew = [];
var_xpnew = [];
var_ypnew = [];
var_thj1new = [];
var_thj2new = [];
var_vj1new =[];
var_vj2new =[];

var_xv =[];
var_yv =[];
var_xvelold=[];
var_yvelold=[];


Rew1 =[];
var_Vel =[];

sogliagoal = 0.02;

%definisco la struttura
s = struct('xpallina',{},'ypallina',{},'velxpallina',{},'velypallina',{},..
       'Rew',{},'Forzajoint1',{},'Forzajoint2',{});

% Sample time
Ts = 0.05; % tempo per passo di simulazione
counter_Ts = 0; % contatore di passi
counter_Ep = 0; % contatore di episodi

%costanti per la camera
my_timestep = 0;
dimxPiatto = 0.25;
dimyPiatto = 0.25;
```

```
        %limiti delle due coppie dei joint
        var_LowerLimit = [-1.0; -1.0];
        var_UpperLimit = [10.0;10.0];


        % valore che serve per il calcolo del reward positivo
        espT = 0.01

        % Penalità per il fallimento dell'esperimento
        PenaltyForFalling = -1;

end

properties
        %valori delle variabili iniziali
                vj1_0 = 0; %velocità iniziale del joint1
                vj2_0 = 0; %velocità iniziale del joint2
                fj1_0 = 5; %coppia iniziale del joint1
                fj2_0 = 7.23; %coppia iniziale del joint2
                thj1_0 = 0; %angolo inziale in radianti del joint1
                thj2_0 = 0; %angolo inziale in radianti del joint2
                vb0x = 0; %velocità iniziale della pallina lungo x
                vb0y = 0; %velocità iniziale della pallina lungo y

        % Initialize system state
        State = zeros(8,1);
end

properties(Access = protected)
        % Initialize internal flag to indicate episode termination
        IsFailure = false
        IsGoal = false
end
```

I parametri dichiarati sono ad esempio l'intervallo di coppia applicabile ai due giunti, un vincolo angolare oltre il quale l'esperimento è considerato un fallimento, o semplicemente la posizione dei giunti e della sfera, che per ora saranno vuoti dato che dovranno essere presi direttamente da CoppeliaSim in seguito.
Nel primo metodo possiamo trovare i seguenti metodi:

- Un costruttore che avrà lo stesso nome della classe: *myBallOnPlateENV*
- *Step*
- *Reset*

Il costruttore avrà il compito di inizializzare due parametri fondamentali per l'ambiente. Il primo è il vettore *ObservationInfo* composto da otto elementi, posizione in x e y e le componenti x e y della velocità della sfera, e le posizioni angolari e le velocità dei due giunti.

```
ObservationInfo = rlNumericSpec([8 1]);
ObservationInfo.Name = 'Ball on Plate States';
ObservationInfo.Description = 'pos_ballx, pos_bally, vel_ballx,
vel_bally, theta_j1, theta_j2, vel_j1, vel_j2';
```

Il secondo parametro è il vettore *ActionInfo* il quale avrà due elementi.

```
ActionInfo = rlNumericSpec([2 1]);
ActionInfo.Name = 'Ball on Plate Action';
```

Il vettore *ActionInfo* servirà per far applicare una coppia piuttosto che un'altra ai due giunti. I valori del vettore sono compresi in un intervallo di numeri continui prestabilito. I valori riportati sotto, sono stati dichiarati nelle proprietà dell'ambiente.

```
%limiti delle due forze dei joint
var_LowerLimit = [-1.0; -1.0];
var_UpperLimit = [10.0;10.0];
```

Il metodo Reset avrà il compito di impostare il simulatore nelle condizioni iniziali. Esso inoltre ritornerà il vettore *InitialObservation* che conterrà i valori iniziali dello stato. Vengono calcolati due valori casuali per le posizioni x e y della sfera, nei limiti permessi dalla dimensione della piattaforma. Il valore numerico prima è in pixel poi verrà convertito in metri. Il vettore *pb_ 0* avrà le componenti x e y con i valori calcolati in maniera casuali, mentre per la componente z si è scelto un valore pari a 0.009 per renderlo quasi attaccato alla piattaforma. Se per la componente z si fosse scelto il valore 0 allora la sfera sarebbe comparsa dentro la piattaforma, e quindi incastrata, e non sulla piattaforma e quindi libera di rotolare.

```
function InitialObservation = reset(this)
          %qua tramite il reset metto il macchinario nelle condizioni
          %iniziali settandogli forza velocità e posizione casuale
          %della pallina sul piatto
          p0x = randi([this.a,this.b],1);
          p0y = randi([this.a,this.b],1);
          p0x = 128-p0x;
          p0y = 128-p0y;
          pb0x = (p0x -64) * (this.dimxPiatto/128);
          pb0y = (p0y -64) * (this.dimyPiatto/128);
          this.var_pb0x = pb0x;
          this.var_pb0y = pb0y;
          pb_0 = [pb0x, pb0y, 0.009];
```

In seguito una variabile logica, chiamata *IsFailure*, verrà settata con valore *false*. Questa variabile servirà in seguito. ma poichè ora non è stata nemmeno fatta partire la simulazione, tale parametro sarà impostato come *false*. Grazie a un istruzione if, se c'è una simulazione in corso attiva, questa verrà terminata. Inoltre in seguito verrà chiusa ogni connessione aperta tra il client Matlab e il server CoppeliaSim, e se ne aprirà subito dopo una nuova.

```
          this.IsFailure = false;
```

```
if (~isempty(this.var_clientID))
    this.var_vrep.simxStopSimulation(this.var_clientID,
    this.var_vrep.simx_opmode_blocking);
    this.var_vrep.simxFinish(this.var_clientID);
    this.var_vrep.delete;
end

vrep=remApi('remoteApi');
vrep.simxFinish(-1); % stop the running communication with V-Rep,

clientID=vrep.simxStart('127.0.0.1',19997,true,true,5000,5);
vrep.simxSynchronous(clientID,true);
```

Gli oggetti che verrano chiamati da CoppeliaSim sono i due *joint*, il *vision sensor* e la sfera.

```
[ret1,Joint1]=vrep.simxGetObjectHandle(clientID,'Revolute1',
vrep.simx_opmode_blocking);
[ret2,Joint2]=vrep.simxGetObjectHandle(clientID,'Revolute2',
vrep.simx_opmode_blocking);
[ret3,camera]=vrep.simxGetObjectHandle(clientID,'Vision_sensor',
vrep.simx_opmode_blocking);
[ret4,pallina]=vrep.simxGetObjectHandle(clientID,'Sphere_dyn',
vrep.simx_opmode_blocking);
```

Prima di poter assegnare una coppia e una posizione angolare ai due *joint* e la posizione casuale alla sfera, questi devono essere chiamati da CoppeliaSim grazie ai seguenti comandi.

```
%qua richiamo le coppie e le posizioni dei due giunti per la
%prima volta

[ret6,position1]=vrep.simxGetJointPosition(clientID,Joint1,
vrep.simx_opmode_streaming);
[ret7,position2]=vrep.simxGetJointPosition(clientID,Joint2,
vrep.simx_opmode_streaming);
[ret8,force1]=vrep.simxGetJointForce(clientID,Joint1,
vrep.simx_opmode_blocking);
[ret9,force2]=vrep.simxGetJointForce(clientID,Joint2,
vrep.simx_opmode_blocking);
[ret_pos_pallina,pos_pallina]=vrep.simxGetObjectPosition(clientID,pallina,
piatto,vrep.simx_opmode_oneshot_wait);
```

Una volta ottenuti i dati dal server CoppeliaSim, vengono assegnati i nuovi valori, per la coppia e la posizione angolare dei due *joint* e la posizione della sfera. I valori per i due *joint* vengono presi direttamente dalle proprietà dell'ambiente. Quindi il

```
[ret_velocity1]=this.var_vrep.simxSetJointTargetVelocity(this.var_clientID ,
this.var_joint1,this.vj1_0,this.var_vrep.simx_opmode_oneshot_wait);

[ret_velocity2]=this.var_vrep.simxSetJointTargetVelocity(this.var_clientID,
this.var_joint2,this.vj2_0,this.var_vrep.simx_opmode_oneshot_wait);
```

```
            [ret_force1]=this.var_vrep.simxSetJointForce(this.var_clientID,
            this.var_joint1,this.fj1_0,this.var_vrep.simx_opmode_oneshot_wait);

            [ret_force2]=this.var_vrep.simxSetJointForce(this.var_clientID,
            this.var_joint2,this.fj2_0,this.var_vrep.simx_opmode_oneshot_wait);

            [ret10]=this.var_vrep.simxSetJointTargetPosition(this.var_clientID,
            this.var_joint1,this.thj1_0,this.var_vrep.simx_opmode_oneshot_wait);

            [ret11]=this.var_vrep.simxSetJointTargetPosition(this.var_clientID,
            this.var_joint2,this.thj2_0,this.var_vrep.simx_opmode_oneshot_wait);

            [ret_position_rand]=this.var_vrep.simxSetObjectPosition(this.var_clientID,
            this.var_pallina,this.var_piatto,pb_0,this.var_vrep.simx_opmode_oneshot_wait);
```

In seguito viene inizializzata la camera e si ottiene la prima immagine da visualizzare in Matlab. Questo permetterà visualizzare la sfera rossa sulla piattaforma nera (vedasi figura 17 del paragrafo successivo). L'istruzione if consente di riutilizzare la stessa finestra, per la visualizzazione della piattaforma e la sfera nelle simulazioni successive, invece di crearne un'altra.

```
            [ret7,resolution,image]=this.var_vrep.simxGetVisionSensorImage2
            (this.var_clientID,this.var_camera,0,this.var_vrep.simx_opmode_streaming);
            if isempty(this.var_figure)
               disp('creo finestra')
               this.var_figure=figure();
               [ret_vision2,resolution,image]=this.var_vrep.simxGetVisionSensorImage2
               (this.var_clientID,this.var_camera,0,this.var_vrep.simx_opmode_buffer);
               figure(this.var_figure)
            else
                figure(this.var_figure)
            end
```

Alla fine del metodo si ottiene la prima osservazione dello stato del sistema, che forma il vettore *InitialObservation*.

```
            InitialObservation = [this.var_pb0x;this.var_pb0y;this.vb0x;this.vb0y;
            this.thj1_0;this.thj2_0;this.vj1_0;this.vj2_0];
            this.State = InitialObservation;

        end
```

Il metodo Step avrà il compito di applicare l'azione prescelta, simulare l'ambiente per un passo di simulazione δt, pari a 50 ms in questo caso, e fornire in uscita l'osservazione, cioè lo stato dopo aver eseguito il passo di simulazione. Inoltre il metodo restituirà il valore del *reward* ottenuto. Infine deve settare una variabile logica che indica se la simulazione è terminata o per raggiungimento dell'obiettivo o per fallimento.
Per prima cosa viene chiamata una funzione esterna di nome *getForce*. Il parametro *Force* sarà un vettore a due componenti, il quale conterrà i valori delle coppie da assegnare ai due *joint*.

```
Force = getForce(this,Action);
```

La funzione *getForce* si trova nei metodi opzionali, alla fine del codice. Il codice è il seguente.

```
function force = getForce(this,action)
          if ~and(all(action >= this.ActionInfo.LowerLimit), ...
                 all(action <= this.ActionInfo.UpperLimit))
              error('Action must be greater or equal to (%g %g) and lower or
              equal to (%g %g).',...
                  this.ActionInfo.LowerLimit(1),this.ActionInfo.LowerLimit(2) ,...
                  this.ActionInfo.UpperLimit(1), this.ActionInfo.UpperLimit(2) );
          end
          force = action;
      end
```

Come si può notare, viene creata un nuovo vettore, con due elementi, di nome *force*. L'agente che interagirà con l'ambiente, dovrò scegliere di eseguire una determinata azione e quindi fornirà due valori di coppia, uno per *joint*, che dovranno essere compresi nel seguente intervallo:

```
%limiti delle due coppie dei joint
var_LowerLimit = [-1.0; -1.0];
var_UpperLimit = [10.0;10.0];
```

I parametri del primo vettore, cioè *var_ LowerLimit*, sono i valori minimi che possono essere assegnati ai due *joint*. I parametri del secondo vettore, cioè *var_ UpperLimit*, invece sono i valori massimi che possono essere assegnati ai due *joint*. In particolare il valore della coppia, scelta dall'agente, per il primo *joint* dovrà essere compreso tra il primo elemento del primo vettore e il primo elemento del secondo vettore. Per il secondo *joint* invece il valore della coppia, scelta dall'agente, dovrà essere compreso tra il secondo elemento del primo vettore e il secondo elemento del secondo vettore. L'istruzione if serve per segnalare un errore nel caso in cui l'agente sceglie un valore di coppia, per uno dei due *joint*, non compreso in questo intervallo. Tale azione sarà considerata invalida e non verrà eseguita, l'agente dovrà scegliere nuovi valori. Se invece i valori sono compresi in tale intervallo, allora significa che l'azione scelta dall'agente è eseguibile e quindi il vettore *force* sarà uguale ai due valori scelti dall'agente, i quali si trovano nel vettore *action*.
In questa parte di codice vengono assegnati i valori della velocità e della coppia dei due *joint*. Al primo *joint* verrà assegnato una valore di coppia pari al valore della prima componente del vettore *Force*, cioè *Force(1)*, e una velocità pari a *1000*sign(Force(1))*. Analogamente al secondo *joint* verrà assegnato un valore di coppia pari al valore della seconda componente del vettore *Force*, cioè *Force(2)*. Il verso di rotazione del singolo *joint* viene dato dal segno della velocità. Se la velocità è positiva allora il *joint2* ruoterà in maniera tale da alzare la piattaforma, mentre se la velocità è negativa allora abbasserà la piattaforma. Analogamente per il *joint1*, ponendosi di fronte al *Ball on Plate* e avendo la *Tower* come ultimo oggetto visualizzato (vedasi figura 14), se la velocità è positiva il *joint1* farà ruotare la piattaforma verso sinistra, mentre se è negativa allora la piattaforma ruoterà verso destra. La velocità, di 1000 rad/s, non è una velocità istantanea. Il simulatore cercherà di arrivare a quella velocità nel minor tempo possibile. Questo è

dovuto anche al fatto che dentro il simulatore ci sono forze, come ad esempio la gravità, che dovranno essere combattute e inoltre bisgona tenere conto che ogni oggetto, dentro CoppeliaSim, è dotato di massa. La scelta di un valore così elevato è stata fatta perché si voleva avere una risposta immediata da parte dei due *joint*.

```
[ret_velocity3]=this.var_vrep.simxSetJointTargetVelocity(this.var_clientID,
this.var_joint2,10000*sign(Force(2)),this.var_vrep.simx_opmode_oneshot);

[ret_force3]=this.var_vrep.simxSetJointForce(this.var_clientID,
this.var_joint2,abs(Force(2)),this.var_vrep.simx_opmode_oneshot);

[ret_velocity4]=this.var_vrep.simxSetJointTargetVelocity(this.var_clientID,
this.var_joint1,10000*sign(Force(1)),this.var_vrep.simx_opmode_oneshot);

[ret_force4]=this.var_vrep.simxSetJointForce(this.var_clientID,
this.var_joint1,abs(Force(1)),this.var_vrep.simx_opmode_oneshot);
```

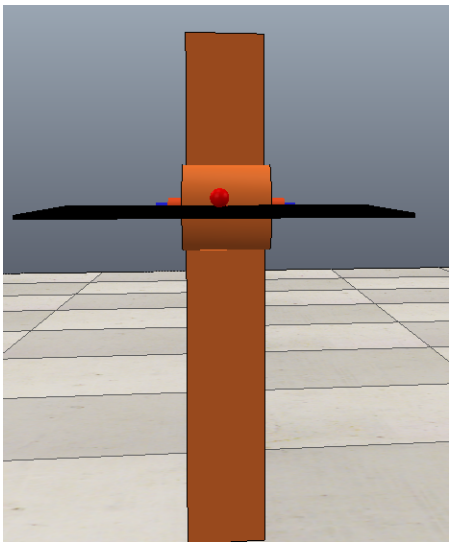


Figura 15: Ball on Plate visto da davanti

A questo punto viene fatta partire la simulazione in modalità sincrona.

```
this.var_vrep.simxStartSimulation(this.var_clientID,
                this.var_vrep.simx_opmode_blocking);

this.var_vrep.simxSynchronousTrigger(this.var_clientID);
```

In seguito viene chiamata una funzione esterna di nome *RGBballdetection*, questa funzione verrà descritta in maniera più dettagliata nel prossimo paragrafo. Per ora basti sapere che questa funzione servirà al calcolo delle velocità e della posizione della sfera per ogni passo di simulazione.

Viene stimata la velocità angolare, come velocità attuale meno quella allo stato precedente fratto il tempo di un passo di simulazione. Il vettore Osservazione viene aggiornato con i valori ottenuti in questo passo di simulazione. Il vettore Osservazione mostra dunque le condizioni nelle quali lo stato si trova in ogni passo di simulazione. E infine viene definita la condizione di fallimento. Per questo lavoro si sono decise le seguenti condizioni: se le posizioni x e y, in valore assoluto, della sfera superano un certo valore limite o le posizioni angolari dei due giunti superano un certo angolo limite allora l'esperimento può considerarsi un fallimento e bisogna fermarlo subito e farne partire uno nuovo. Il valore limite per la posizione della sfera è di 0.12m, quindi sul piatto 0.25 m x 0.25 m la sfera si può muovere dentro una piattaforma grande 0.24 m x 0.24 m. Il valore 0.12 è dovuto al fatto che il centro della piattaforma coincide con il centro degli assi cartesiani. Il valore dell'angolo limite invece è $\pi/6$. Un'altra condizione, per la quale la simulazione verrà considerata un fallimento, è il superamento del numero di passi di simulazione permessi. Una simulazione non può andare all'infinito, quindi si è deciso di porre anche questo vincolo. Con questa scelta si forza l'agente ad apprendere una legge di controllo entro il numero di passi di simulazione permessi. Poi viene definita la condizione di goal come segue:

```
IsGoal = (this.Rew1 < this.sogliagoal);
```

Dove il valore della sogliagoal è pari a 0.02 m mentre Rew1 è definito come la norma 2 del vettore X. Il vettore X rappresenta la distanza effettiva dal centro della piattaforma con le componenti della velocità della sfera.

```
X = [(this.var_xnew);(this.var_ynew);(this.var_xpnew);(this.var_ypnew)];
this.Rew1 = norm(X,2);
```

Quindi secondo questa definizione di goal si può capire che una volta che la sfera si avvicina al centro della piattaforma con velocità e posizione praticamente nulla allora la simulazione può considerarsi un successo, e quindi l'obiettivo è soddisfatto. Tenendo conto delle dimensioni della sfera, cioè 1.5 cm di diametro, la soglia del goal è stata scelta in modo tale da contenere la sfera ipotizzando si arrivi ad avere una velocità praticamente nulla in prossimità del centro della piattaforma. A questo punto viene fornito il Reward per questo passo di simulazione. La formulazione della funzione di reward verrà spiegata nel capitolo successivo, nel quale si parlerà degli esperimenti effettuati. Questo perché la formula che fornisce il reward è stata cambiata più di una volta. L'ultima cosa che il codice farà sarà quello di stabilire se andare avanti un altro passo di simulazione o fermarla grazie alla seguente condizione:

```
IsDone = (IsFailure || IsGoal);
```

Se è soddisfatta una tra le due condizioni, o di fallimento o di goal allora la simulazione terminerà altrimenti si andrà avanti un altro passo di simulazione.

6.2 Ball detection

Il codice "RGBballdetection" serve per il calcolo delle coordinate e della velocità della sfera. Per fare ciò si è scelto di cambiare il colore della piattaforma in nero e della sfera in rosso. Questo si è fatto per avere due tonalità di colore completamente diverse e quindi non avere errori nel rilevamento della sfera. La funzione che definisce il codice riceve in input l'immagine rilevata dal sensore di visione, le coordinate x e y della sfera, il valore numerico del time step, tempo per passo di simulazione, e la dimensione della piattaforma. L'immagine si ottiene grazie alla funzione *simxGetVisionSensorImage2* che mi restituisce l'immagine in forma di un array a 8 bit 128x128x3. I valori contenuti dentro l'array sono quelli dei colori che rileva, in particolare al colore nero è assegnato il valore 23 mentre quello del rosso è in un intorno di 200. Il valore dunque è direttamente proporzionale all'intensità del colore. Il codice trasforma l'immagine ricevuta da array a matrice 128x128 con dentro i valori numerici dei colori rilevati. Questa matrice verrà poi trasformata in una matrice logica nella quale il valore 1 verrà assunto dal colore rosso e 0 altrimenti. [t]

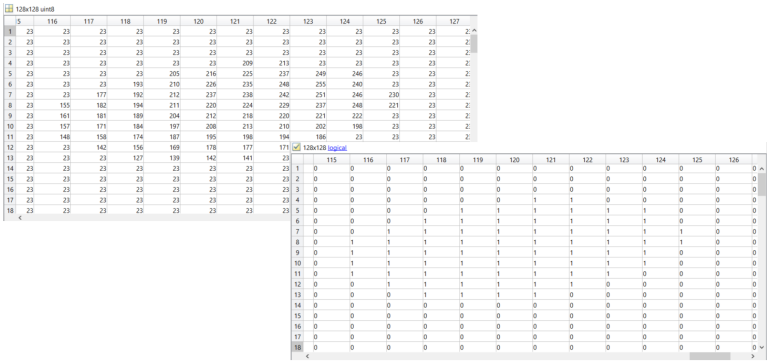


Figura 16: Matrice iniziale e matrice logica

A questo punto grazie al comando *regionprops dell'Image Processing Toolbox di Matlab* viene calcolato il centroide della sfera. Le coordinate però saranno restituite in pixel e quindi si è dovuto fare una conversione da pixel a metri. Per fare tale conversione però si è dovuta prestare attenzione al sistema di riferimento del piatto che ha gli assi cartesiani x e y con origine al centro della piattaforma e invece i pixel vengono contati da 0 a 128 da sinistra verso destra per l'asse x e dall'alto verso il basso per l'asse y con origine nello spigolo in alto a destra(vedasi figura 17).
La velocità invece è stata stimata con la classica formula del moto rettilineo uniforme $v = \frac{\Delta s}{\Delta t}$. La velocità viene calcolata in ogni istante la differenza fra la posizione attuale e quella al passo di simulazione precedente fratto

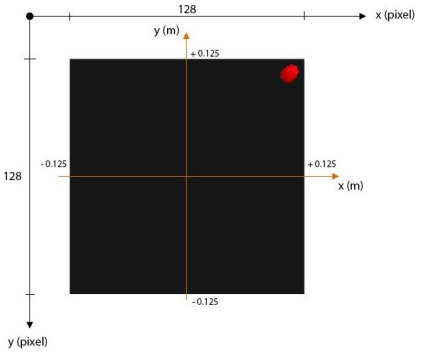


Figura 17: Sistema di riferimento dell'immagine

il valore del tempo per passo di simulazione. Si è scelto di utilizzare dentro il simulatore un valore di t pari a 50ms. (paragrafo tratto da [19])

```
    function [xnew,ynew,x_pnew,y_pnew, ...
    flag_outOfTheBox] = RGBballdetection(imageRGB,x0,y0,TimeStep, dimxPiatto,dimyPiatto)
% The function computes the ball-centroid cordinates and its velocity
% INPUT: - Image of the red ball on the black plate
% - old ball-centroid cordinates
% - timestep of the simualtion
% - plate size x and y
% OUTPUT: - x and y coordinates of the ball centroid
% - ball centroid velocity along x and y
    Rdetection = imageRGB(:,:,1);
    Rdetection1 = (Rdetection > 200);

    % PixelPos = regionprops('table',Rdetection1,'Centroid');
    PixelPos = regionprops(Rdetection1,'Centroid');
    if isempty(PixelPos)
       fprintf(1,'******* Warning! Ball out of the plate at timestep %g! ********\n',
           TimeStep);

       centroids = [128, 128];
       flag_outOfTheBox = true;
    else
        flag_outOfTheBox = false;
        centroids = cat(1, PixelPos.Centroid);
    end
    xPixelnew = 128- centroids(1,1);
    yPixelnew = 128- centroids(1,2);

    xnew = (64-xPixelnew)*(dimxPiatto/128);
    ynew = (yPixelnew-64)*(dimyPiatto/128);
    if ~flag_outOfTheBox
        x_pnew = ((xnew -x0)/TimeStep);
        y_pnew = ((ynew -y0)/TimeStep);
    else
        x_pnew = 0;
        y_pnew = 0;
    end
    if flag_outOfTheBox
        fprintf(1, '************** Warning! Ball cohordinates: %f, %f\n', ...
            xnew, ynew);
    end
```

6.3 L'algoritmo DDPG

La parte finale del progetto è stata quella della scelta di un algoritmo di apprendimento e del suo utilizzo. Come già detto lo scopo del progetto è quello di riuscire a portare la sfera, indipendentemente dalla posizione iniziale, al centro della piattaforma su cui si trova. L'algoritmo scelto doveva tener conto del fatto che il numero di azioni possibili in uno stato non erano un comprese in un intervallo numerico discreto ma continuo,

quindi in un solo stato ci potevano essere infinite possibilità di scegliere una determinata azione. L'algoritmo scelto è dunque il Deep Deterministic Policy Gradient (DDPG). Tale algoritmo è a disposizione nel toolbox *Reinforcement Learning* in Matlab (tratto da [20]). L'algoritmo è stato utilizzato a "scatola chiusa". Sfruttando alcuni esempi presenti nel manuale d'uso del toolbox RL di Matlab, sono stati individuati la configurazione ed i parametri da assegnare affinché fosse possibile l'utilizzo dell'algoritmo DDPG. Si è evitato di andare a capire la logica interna ma si è cercato di comprendere il funzionamento di tale codice. L'algoritmo in questione è il seguente.

```
% myTrainBallOnPlateRL

env = myBallOnPlateENV;
obsInfo = getObservationInfo(env);
numObs = prod(obsInfo.Dimension);
actInfo = getActionInfo(env);
numAct = prod(actInfo.Dimension);

% specify the number of outputs for the hidden layers.
hiddenLayerSizecritic = 50;
hiddenLayerSizeactor = 75;

observationPath = [
    imageInputLayer([numObs 1 1],'Normalization','none','Name','observation')
    fullyConnectedLayer(hiddenLayerSizecritic,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(hiddenLayerSizecritic,'Name','fc2')
    additionLayer(2,'Name','add')
    reluLayer('Name','relu2')
    fullyConnectedLayer(hiddenLayerSizecritic,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(1,'Name','fc4')];
actionPath = [
    imageInputLayer([numAct 1 1],'Normalization','none','Name','action')
    fullyConnectedLayer(hiddenLayerSizecritic,'Name','fc5')];

% create the layerGraph
criticNetwork = layerGraph(observationPath);
criticNetwork = addLayers(criticNetwork,actionPath);

% connect actionPath to obervationPath
criticNetwork = connectLayers(criticNetwork,'fc5','add/in2');

criticOptions = rlRepresentationOptions('LearnRate',1e-02,'GradientThreshold',1);

critic = rlRepresentation(criticNetwork,obsInfo,actInfo,...
    'Observation',{'observation'},'Action',{'action'},criticOptions);

actorNetwork = [
    imageInputLayer([numObs 1 1],'Normalization','none','Name','observation')
    fullyConnectedLayer(hiddenLayerSizeactor,'Name','fc1')
    reluLayer('Name','relu1')
    fullyConnectedLayer(hiddenLayerSizeactor,'Name','fc2')
```

```
    reluLayer('Name','relu2')
    fullyConnectedLayer(hiddenLayerSizeactor,'Name','fc3')
    reluLayer('Name','relu3')
    fullyConnectedLayer(numAct,'Name','fc4')
    tanhLayer('Name','tanh1')];

actorOptions = rlRepresentationOptions('LearnRate',1e-03,'GradientThreshold',1);

actor = rlRepresentation(actorNetwork,obsInfo,actInfo,...
    'Observation',{'observation'},'Action',{'tanh1'},actorOptions);

agentOptions = rlDDPGAgentOptions(...
    'SampleTime',env.Ts ,...
    'TargetSmoothFactor',1e-3,...
    'ExperienceBufferLength',1e6 ,...
    'DiscountFactor',0.99,...
    'MiniBatchSize',256);
agentOptions.NoiseOptions.Variance = 1e-4;
agentOptions.NoiseOptions.VarianceDecayRate = 1e-6;
agentOptions.SaveExperienceBufferWithAgent = true;


agent = rlDDPGAgent(actor,critic,agentOptions);

trainOpts = rlTrainingOptions;
trainOpts.MaxEpisodes = 100000;
trainOpts.MaxStepsPerEpisode = 50;
trainOpts.StopTrainingCriteria = "AverageReward";
trainOpts.StopTrainingValue = 2;
trainOpts.ScoreAveragingWindowLength = 50;

trainOpts.SaveAgentCriteria = "EpisodeReward";
trainOpts.SaveAgentValue = 1;
trainOpts.SaveAgentDirectory = "savedAgents";

trainOpts.Verbose = true;
trainOpts.Plots = "training-progress";
trainingInfo = train(agent,env,trainOpts);
```

Per prima cosa viene richiamato l'ambiente poi le informazioni sui vettori Osservazione e Azione. In seguito vengono creati due elementi fondamentali per l'addestramento, l'Actor e il Critic (tratto da [21]). L'Actor avrà il compito di prendere il vettore Osservazione in input e di fornire in output una corrispettiva azione possibile. Quindi la funzione dell'Actor è quella di stabilire l'azione da eseguire in un determinato stato. Il Critic invece avrà in input lo stesso vettore Osservazione e in più l'azione fornita dall'Actor e in output fornirà una stima della funzione valore. Infine l'Agente, di cui si parlava nel capitolo precedente, è l'insieme dell'Actor con il Critic e l'AgentOptions. Quest'ultimo è un insieme di parametri che andrà opportunamente impostato. In particolare troviamo (tratto da [22]):

- Sample Time: è il tempo per passo di simulazione, in questo caso preso direttamente

dall'ambiente, ha un valore pari a 50ms;

- Target Smooth Factor: è un fattore di attenuazione usato per gli aggiornamenti dell'Actor e del Critic, ha un valore scalare positivo minore o uguale a 1;
- Experience Buffer Length: dimensione del buffer dell'esperienza, specificata come numero intero positivo. Durante l'addestramento, l'agente aggiorna l'Actor e il Critic utilizzando un mini-batch di esperienze campionate casualmente dal buffer.
- Discount Factor: fattore di sconto applicato ai reward futuri, dando più peso a quelli attuali, ha un valore scalare positivo minore o uguale a 1;
- Mini Batch Size: dimensione del mini-batch di esperienza casuale, specificata come numero intero positivo. Durante ogni simulazione, l'agente campiona casualmente le esperienze dall'experience buffer durante il calcolo dei gradienti per l'aggiornamento delle proprietà del Critic. I mini-batch di grandi dimensioni riducono la varianza durante il calcolo dei gradienti ma aumentano lo sforzo di calcolo.
- Noise Options: per un agente con più azioni, se le azioni hanno intervalli e unità differenti, è probabile che ogni azione richieda parametri del modello di rumore differenti. Se le azioni hanno intervalli e unità simili, è possibile impostare i parametri del rumore per tutte le azioni sullo stesso valore. Ad esempio, per un agente con due azioni, si può impostare la varianza di entrambe le azioni su uno stesso valore e utilizzando lo stesso tasso di decadimento, sempre se hanno unità e intervallo uguali.

I parametri sopra mostrati,sono dei parametri utili all'addestramento della rete neurale. In quanto l'agente, unione dell'Actor con il Critic, è una rete neurale. La trattazione e lo studio di tale argomento non è stato approfondito poiché non era oggetto di tesi. Per quanto riguarda però il funzionamento di tale algoritmo è giusto almeno sapere come lavora senza entrare nel dettaglio tecnico della progettazione di una rete neurale. L'agente utilizzato sfrutta la tecnica del Policy Gradient per andare a imparare la policy, cioé la legge di controllo del sistema. Il Policy Gradient va a utilizzare una funzione approssimante della policy, che viene definita attraverso dei pesi e aggiorna i pesi in funzione della discesa al gradiente della funzione descritta dai pesi. Calcola a ogni passo di simulazione il gradiente della funzione rispetto ai pesi. In una rete neurale, in genere vengono presi i dati di uno o più episodi, che sono fondamentalmente un'associazione di stato azione e reward e vengono presi tutti i passi di simulazione di uno o più episodi, tale lunghezza viene scelta in base alla lunghezza dell'experience buffer. Tale memoria di dati, di uno o più episodi, viene mandata in input all'agente, il quale in questo caso è una rete neurale. La rete neurale può lavorare in due modi, con un operazione di forward, in cui va in avanti ad aggiornare i propri pesi oppure backward, in cui va a calcolare la discesa al gradiente per andare a minimizzare un determinato errore. Il calcolo della discesa al gradiente può essere fatto a ogni passo di simulazione, e quindi sfruttando l'intera memoria di dati, e si chiama batch size oppure decide di aggiornare il gradiente rispetto a un sottoinsieme di dati nella memoria del buffer e quindi si chiamerà mini batch size. Quindi sulla base

di un'unica memoria di dati che vengono forniti, verranno fatti N calcoli della discesa al gradiente per approssimare meglio la funzione.
Quindi per chiarire meglio il concetto spiegato sopra, esso può essere interpretato come segue. Viene utilizzata una rete neurale descritta da dei pesi i quali vengono aggiornati secondo una discesa al gradiente. Tale discesa al gradiente può essere calcolata o sull'intera memoria di dati, con una determinata lunghezza, o su sotto sezioni di tale memoria, si parla dunque di minibatch size, approssimando meglio la funzione.
Per la scelta dei valori di questi parametri, c'è stato l'aiuto del supervisore del progetto.
Alla fine del codice invece vengono impostati i parametri per l'addestramento. Questi parametri sono:

- MaxEpisodes: numero massimo di episodi (simulazioni), si è scelto un numero pari a cento mila simulazioni. Tale numero è stato scelto poiché a priori non si ha la conoscenza del numero effettivo di simulazioni necessarie ai fini dell'apprendimento.

- MaxStepPerEpisode: numero massimo di passi per simulazione, il valore assegnato verrà spiegato meglio nel prossimo capitolo, poiché nel croso degli esperimenti è stato modificato.

- StopTrainingValue: è il valore, del reward cumulato alla fine di una simulazione, per il quale l'addestramento verrà fermato, anche il valore di questo parametro verrà trattato nel prossimo capitolo poiché nel corso degli esperimenti è stato modificato.

- ScoreAveragingWindowLenght: questo paramentro indica il numero di simulazioni su cui si calcolerà l'Average reward, cioè una media dei reward. Si è scelto un valore pari a 50 simulazioni.

- StopTrainingCriteria: è il criterio con cui l'addestramento viene fermato in questo caso viene scelto il valore AverageReward. Quindi solamente quando le ultime 50 simulazioni convergeranno a un determinato valore, la media sarà sempre la stessa e quindi si può dire che l'addestramento è completato.

Per quanto riguarda il salvataggio del file dell'agente in una determinata simulazione si è scelto il criterio del reward cumulato a fine simulazione. Tale punteggio ottenuto, se superata una certa soglia, porterà alla condizione di salvataggio dei dati dell'agente. Si è scelto dunque di salvare gli agenti che hanno portato un reward positivo a fine della simulazione. Il valore anche in questo caso è stato scelto in base a come è stato descritta la funzione del reward.
Uno degli ultimi parametri utilizzati dall'algoritmo sono gli hiddenLayerSize. Tale parametro rappresenta il numero di strati di cui è composta una rete neurale, escludendo lo strato in input e quello in output che sono visibili. Le reti neurali però non sono state trattate in questo progetto, anche perché l'algoritmo non utilizza reti neurali normali ma deep che sono ancora più complesse. Per la scelta del valore di tale parametro, anche in questo caso,c'è stato l'aiuto e i consigli del supervisore del lavoro.

7 Esperimenti

7.1 Analisi dei dati

In questo capitolo verrà trattata la parte di analisi dei dati di alcuni esperimenti effettuati. L'algoritmo DDPG fornisce in output, in tempo reale, l'andamento di ogni episodio.

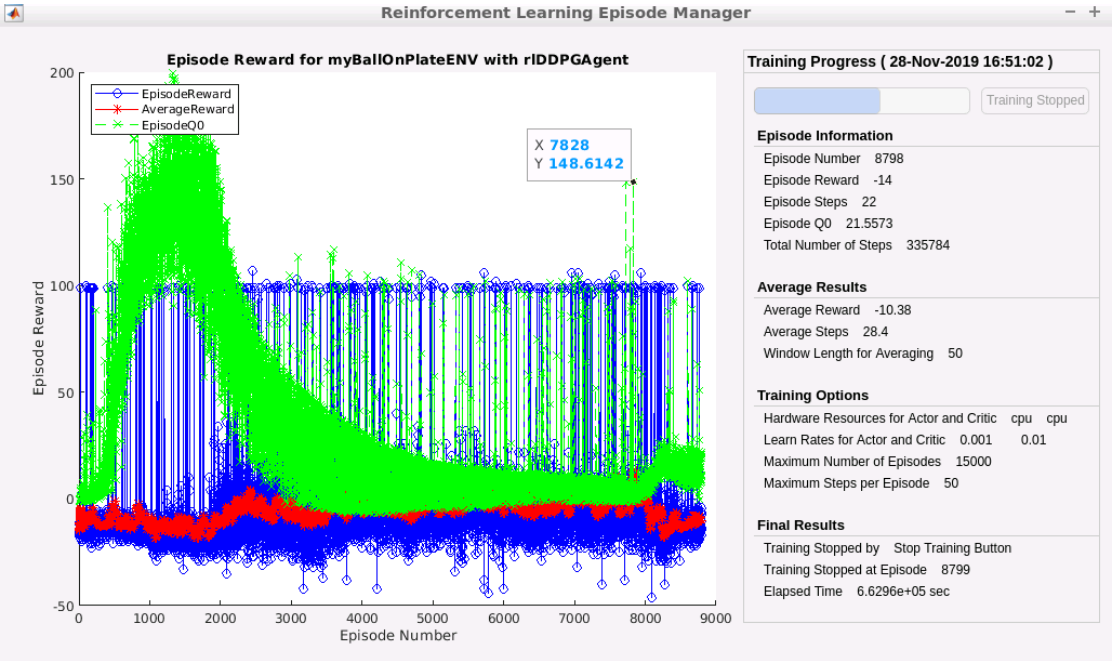


Figura 18: Andamento delle simulazioni

Come si può vedere dalla figura 18, l'algoritmo fornisce un grafico nel quale ci sono 3 parametri. In ordinata in blu mostra il valore del *reward* cumulato a fine di ogni episodio, in rosso invece il *reward* medio calcolato sulle ultime 50 simulazioni. In verde invece mostra la stima, fatta dal Critic della *value function* per ogni episodio. In ascissa invece mostra il numero di episodi effettuati.
Come spiegato nel capitolo scorso un episodio può considerarsi concluso quando si arriva a soddisfare il *goal*, e quindi con un successo, oppure quando vengono violati i vincoli imposti, e quindi un fallimento. L'apprendimento invece può essere considerato concluso se una successione di episodi converge al valore ottimale. Il grafico quindi dovrebbe mostrare il valore blu che converge, ma se un numero discreto di episodi converge all'ottimo allora significa che anche *l'average reward* convergerà all'ottimo, e infine anche le stime del Critic dovranno convergere allo stesso valore. Se ciò accade allora l'addestramento può considerarsi un successo e significa che l'agente ha appreso una legge di controllo per soddisfare il goal indipendentemente dallo stato iniziale in cui si trova. Se invece l'addestramento finisce a causa del raggiungimento del numero limite di episodi, senza che nessun parametro converga all'ottimo, allora l'addestramento è considerato un fallimento

e significa che bisogna cambiare qualche parametro prima di avviare una nuovo addestramento. Verranno proposti due esperimenti nei quali si vedranno due comportamenti, da parte dell'agente, completamente diversi.

7.2 Analisi degli esperimenti

Per questo esperimento si è scelto di definire il *reward* come segue: Per prima cosa vengono definiti i due vettori a quattro componenti X e Y. Il vettore X contiene le componenti x e y della posizione della sfera e le componenti v_x e v_y della velocità v della sfera nello stato x_t. Il vettore Y invece contiene le componenti x e y della posizione della sfera e le componenti v_x e v_y della velocità v della sfera sempre nello stato x_{t-1}.

```
X = [(this.var_xnew);(this.var_ynew);(this.var_xpnew);(this.var_ypnew)];

Y = [(this.var_xv);(this.var_yv);(this.var_xvelold);(this.var_yvelold)];
```

In seguito viene calcolata la norma due di entrambi i vettori.

```
N(x) = norm(X,2);
N(y) = norm(Y,2);
```

Se nello stato x_t il sistema soddisfa la condizione N(x) minore della *sogliagoal*, pari a 0.02, allora il *reward* fornito è pari a 100, altrimenti se nello stato x_t soddisfa la condizione N(x) minore di N(y), allora il *reward* fornito è pari a 1 altrimenti gli viene fornito un *reward* pari a -1. Se nessuna delle due condizione è soddisfatta allora significa che sono stati violati i vincoli imposti, cioè superamento dell'angolo limite o la sfera è caduta fuori dalla piattaforma, allora il *reward* fornito è pari a -5.

```
if ~this.IsFailure
   if N(x) < this.sogliagoal
      Reward = 100;
   elseif N(x) < N(y)
      Reward = 1;
   else
      Reward = -1;
   end
else
  Reward = this.PenaltyForFalling; % il valore di PenalityForFalling è pari a -5
end
```

La condizione N(x) minore di N(y) è stata scelta per far apprendere all'agente di arrivare verso il centro della piattaforma con una velocità praticamente nulla. Ogni volta che la norma 2 dello stato x_t è più piccola di quella nello stato x_{t-1}, allora significa che la distanza del punto X (il vettore X può essere pensato come un punto in un sistema quadridimensionale) dal centro diminuisce. Quindi ogni volta che la sfera si avvicina al centro della piattaforma, però diminuendo anche la velocità, l'agente verrà premiato con un *reward* positivo. Se invece la sfera si avvicina al centro, aumentando la velocità, questo potrebbe portare la norma N(x) ad essere più grande di N(y) e quindi l'agente verrà punito. Non basta dunque essere in prossimità del centro della piastra, ma bisogna

anche avere una velocità quasi nulla.
In questo esperimento il numero massimo di passi simulazioni per episodio è stato impostato a 50 mentre il numero massimo di episodi è stato impostato a cento mila. Non sapendo qual fosse il numero di episodi necessari per l'agente affinché apprendesse una legge di controllo, si è scelto un numero molto elevato.
L'andamento di questo esperimento è il seguente:

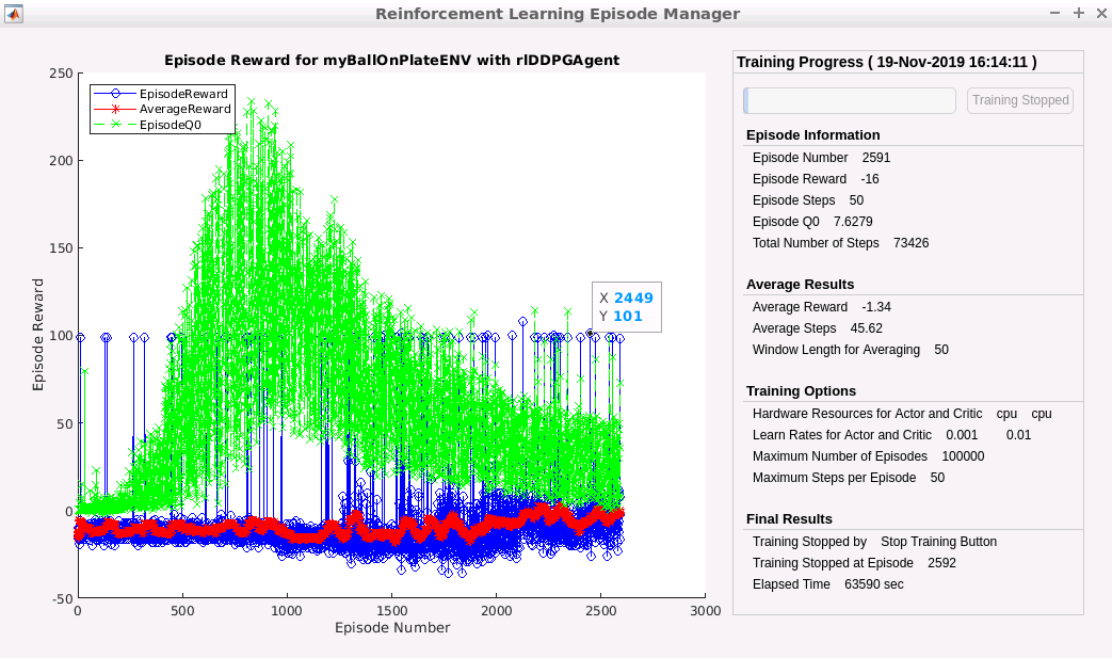
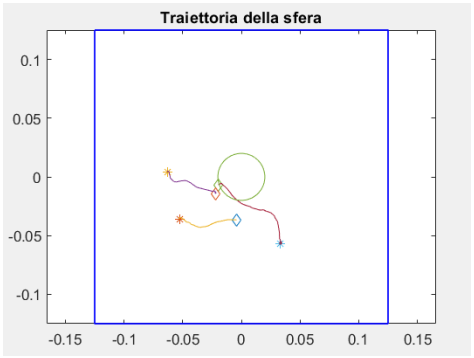


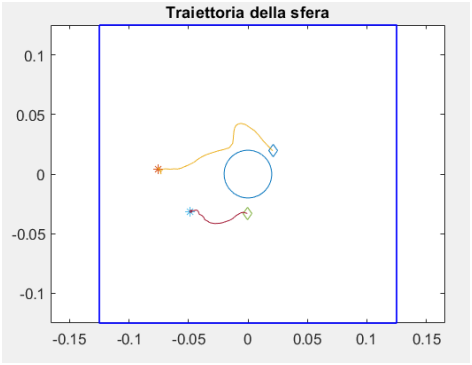
Figura 19: Andamento del primo esperimento presentato

Come si può vedere direttamente dal grafico le prime 1300 (circa) simulazioni hanno portato solamente fallimenti. Come già spiegato nel capitolo 5 paragrafo 4, la prima fase di un esperimento è quella dell'esplorazione, quindi l'agente sceglie casualmente le azioni da eseguire. Questo porterà l'agente ad apprendere che determinate azioni sono da evitare e quindi le simulazioni successive iniziano ad avere un *reward* più alto. All'episodio numero 2591 l'addestramento è stato fermato per poter analizzare meglio cosa avesse appreso l'agente.
In questo esperimento l'agente ha agito in due maniere diverse, in entrambi i casi però ha sfruttato il numero massimo di passi di simulazione disponibili. Analizzando alcuni episodi si è visto che l'agente in determinate occasioni preferiva sfruttare il maggior numero di passi di simulazione a disposizione, per guadagnare più *reward* possibile. Questo però portava l'episodio a concludersi con un fallimento, poiché il numero di passi di simulazioni non erano più sufficienti per arrivare verso il centro della piattaforma. La scelta, da parte dell'agente, di voler massimizzare il valore del *reward* a fine episodio non è sbagliata, infatti come spiegato nel capitolo 5 l'obiettivo dell'agente è proprio quello di ottenere più *reward* positivo possibile. Questa scelta però portava l'addestramento a

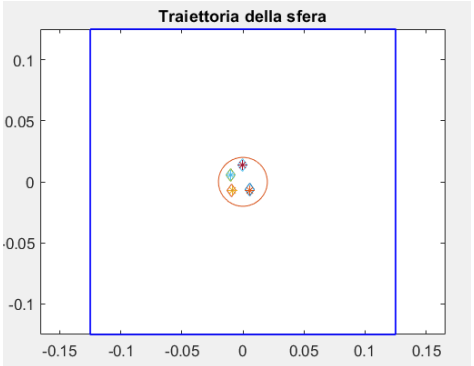
diventare molto più lungo, infatti l'agente dava più priorità al *reward* accumulato nel percorso verso il centro della piastra piuttosto che al singolo *reward* ottenibile per aver soddisfatto il goal, e ciò portava nella maggior parte dei casi che un episodio venisse considerato fallimentare per esaurimento del numero di passi di simulazione disponibili. Nella figura 20, i grafici a e b mostrano la traiettoria che la sfera ha compiuto in 5 simulazioni diverse. Per evitare che le traiettorie si sovrapponessero si sono scelte solo queste 5 simulazioni in modo che il grafico fosse più chiaro, molti episodi però hanno avuto una traiettoria della sfera simile a quelle riportate nei grafici sottostanti. La traiettoria della sfera comincia dal punto iniziale, contrassegnato nel grafico con un asterisco, per poi arrivare alla fine dei 50 passi di simulazione nel punto finale, contrassegnato con un rombo. La circonferenza al centro rappresenta la zona dentro la quale, se soddisfatta la prima condizione dell'istruzione if, viene assegnato il *reward* con valore 100 e l'episodio è considerato un successo. I grafici mostrano l'intento da parte dell'agente di voler portare la sfera al centro della piattaforma, ma anche il voler accumulare più *reward* possibile, però in modo inevitabile comporta il fallimento della simulazione.



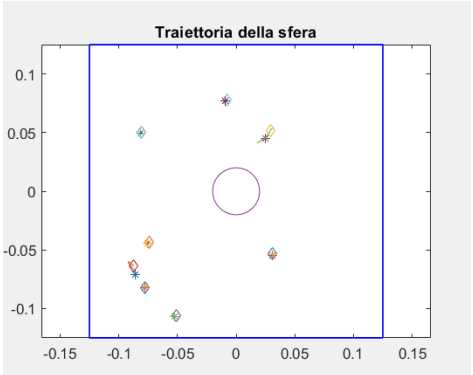
(a) *Traiettoria della sfera di 3 simulazioni diverse*



(b) *Traiettoria della sfera di 2 simulazioni diverse*



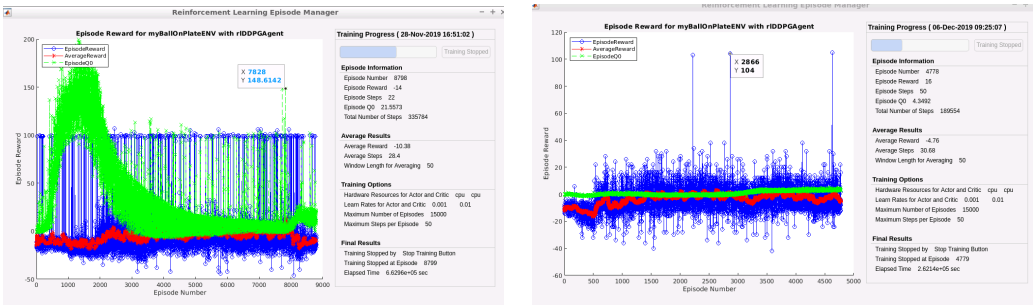
(c) *Posizione iniziale direttamente al centro della piastra*



(d) *Sfera ferma sulla piastra*

Figura 20

Nella scelta della posizione casuale della sfera, a inizio simulazione, non si è esclusa la possibilità di posizionarla direttamente al centro della piattaforma. Quando la sfera a inizio simulazione si trovava direttamente all'interno della circonferenza mostrata in figura 20 c, dopo uno o due passi di simulazione l'episodio terminava con un successo in quanto sia la posizione che la velocità soddisfacevano la prima condizione dell'istruzione if e quindi veniva assegnato il *reward* pari a 100. La gran parte dei picchi blu che si possono vedere nel grafico 19 sono infatti episodi terminati nei primi passi di simulazione in quanto la sfera si trovava già al centro (vedasi figura 20 c). Questo però ha portato l'agente ad assumere il secondo comportamento, cioè quello di rimanere quasi fermo, spostandosi il minimo possibile per guadagnare sempre un *reward* positivo senza arrivare al goal (vedasi figura 20 d).
Il secondo comportamento, da parte dell'agente, è un problema superabile. L'agente anche se a ogni passo di simulazione avesse guadagnato un punto, alla fine dei 50 passi di simulazioni avrebbe guadagnato un massimo di 50. Poiché il *reward* per raggiungimento del goal è 100, l'agente dovrebbe apprendere che la scelta di rimanere fermi non è quella giusta e quindi dopo un determinato numero di episodi questo comportamento dovrebbe sparire. Tale apprendimento deve però essere esteso per ogni stato iniziale x_0, e quindi per ogni posizione sulla piattaforma, escluso lo stato iniziale nel quale soddisfa direttamente il goal.
Il primo comportamento invece avrebbe portato l'addestramento a concludersi dopo un numero sproporzionato di simulazioni. Tale deduzione è stata tratta dai seguenti due esperimenti mostrati in figura 21.



(a) *Esperimento con posizione casuale lasciato per più tempo*

(b) *Esperimento con posizione fissa*

Figura 21

L'esperimento in figura 21 (a) ha la stessa definizione di *reward* ma è stato fermato dopo quasi nove mila simulazioni. L'andamento dell'apprendimento non era soddisfacente e si è deciso dunque di provare a cambiare una condizione iniziale. L'esperimento in figura 21 (b) mostra l'andamento dello stesso esperimento ma cambiando la condizione iniziale della sfera. Invece di inizializzare il sistema con una posizione casuale, si è deciso di tenere le coordinate x e y della sfera ferme. Ogni simulazione infatti cominciava nello stesso stato iniziale x_0 nella posizione (0.10;0.10). Dopo circa cinque mila simulazioni

però, l'agente è riuscito a raggiungere il centro della piattaforma solamente 3 volte. Con condizioni iniziali casuali l'agente ci avrebbe impiegato troppo per apprendere.
Si è deciso dunque di cambiare la formula del *reward*. Questa volta è stato assegnato un *reward* pari a 1 se N(x) fosse stato minore della soglia del goal, -1 se avesse violato i vincoli e 0 altrimenti, cioè quando la sfera rotolava sulla piastra. Questa scelta è stata fatta per evitare che l'agente accumulasse *reward* durante lo spostamento verso il centro della piastra. In questo caso il *reward* massimo accumulabile è pari a 1. Con questa scelta si è dunque cercato di evitare di avere un comportamento, da parte dell'agente, simile a quello mostrato nel primo esperimento.

```
if ~this.IsFailure
   if N(x) < this.sogliagoal
      Reward = 1;
   else
      Reward = 0;
   end
else
  Reward = this.PenaltyForFalling; % il valore di PenalityForFalling è pari a -5
end
```

L'addestramento è stato fermato dopo 37 mila episodi, poiché anche questa volta l'agente non è riuscito ad apprendere una legge di controllo. Guardando inoltre l'andamento dell'addestramento, in figura 22 si può notare che non c'è un miglioramento del *reward* medio e della stima del Critic, quindi anche se l'addestramento avesse proseguito ancora non ci sarebbe stato alcun cambiamento.

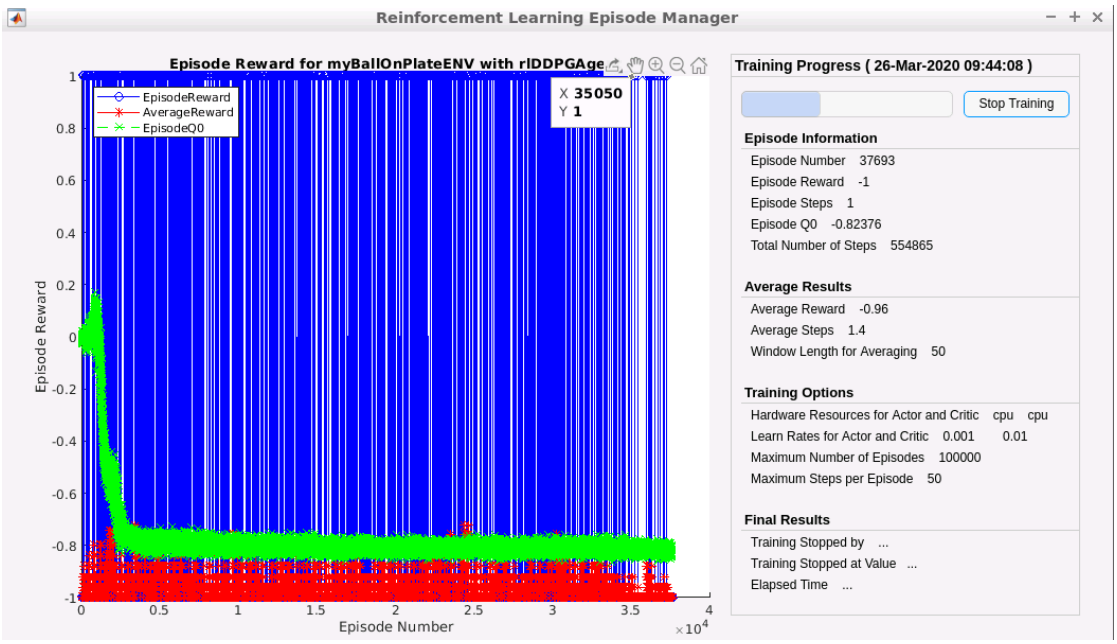


Figura 22: Esperimento con formula del reward cambiato

Come si può vedere in figura 22, il *reward* medio è rimasto sempre vicino al valore -1, Questo permette di capire che il numero degli esperimenti con esito positivo è più piccolo di quelli conclusi con un fallimento. In particolare questa volta si è posta una condizione di salvataggio dei dati dell'agente che permetteva di capire il numero degli esperimenti che hanno avuto successo. Ogni volta che l'agente otteneva un *reward* pari a 1, i dati dell'agente di quella simulazione venivano salvati. Su poco più di 37 mila simulazioni totali, ci sono solamente 931 successi, poiché ci sono solamente 931 file contenenti i dati dell'agente. Di questo numero però bisognerebbe sottrarre il numero delle volte che la sfera si trovava direttamente al centro della piattaforma, a causa dell'assegnazione delle coordinate casuali.

7.3 Conclusioni

In conclusione sia la prima che la seconda impostazione dell'assegnazione del *reward* non hanno portato a conclusione l'addestramento. Non si è riusciti ad addestrare l'agente in modo che compiesse le azioni giuste per soddisfare il *goal*. In questo ultimo capitolo sono stati mostrati solo alcuni esperimenti, più significativi e non tutti, ma nessuno ha fornito risultati positivi.
Non avendo avendo avuto più tempo a disposizione il numero degli esperimenti non è stato sufficiente, forse avendone fatti altri si avrebbe ottenuto un risultato diverso. Anche se gli esperimenti non sono andati a buon fine, in questa tesi si è riusciti comunque a implementare l'algoritmo e analizzare il comportamento dell'agente, vedendo come interagiva con il sistema.

Riferimenti

[1] CoppeliaSim User Manual. *CoppeliaSim from the creators of V-REP*. URL: https://www.coppeliarobotics.com/helpFiles/index.html.

[2] A. Knuplez, A. Chowdhury e R. Svecko. "Modeling and control design for the ball and plate system". In: *IEEE International Conference on Industrial Technology, 2003*. Vol. 2. Dic. 2003, 1064–1067 Vol.2. DOI: 10.1109/ICIT.2003.1290810.

[3] Luis Morales. et al. "A Sliding-Mode Controller from a Reduced System Model: Ball and Plate System Experimental Application". In: *Proceedings of the 14th International Conference on Informatics in Control, Automation and Robotics - Volume 1: ICINCO,* INSTICC. SciTePress, 2017, pp. 590–597. ISBN: 978-989-758-263-9. DOI: 10.5220/0006425905900597.

[4] A. Zeeshan, N. Nauman e M. Jawad Khan. "Design, control and implementation of a ball on plate balancing system". In: *Proceedings of 2012 9th International Bhurban Conference on Applied Sciences Technology (IBCAST)*. 2012, pp. 22–26.

[5] CoppeliaSim User Manual. *Building a clean model tutorial*. URL: https://www.coppeliarobotics.com/helpFiles/en/buildingAModelTutorial.htm.

[6] CoppeliaSim User Manual. *Scene objects*. URL: https://www.coppeliarobotics.com/helpFiles/en/objects.htm.

[7] CoppeliaSim User Manual. *Object common properties*. URL: https://www.coppeliarobotics.com/helpFiles/en/commonPropertiesDialog.htm.

[8] CoppeliaSim User Manual. *Shapes*. URL: https://www.coppeliarobotics.com/helpFiles/en/shapes.htm.

[9] CoppeliaSim User Manual. *Shape dynamics properties*. URL: https://www.coppeliarobotics.com/helpFiles/en/shapeDynamicsProperties.htm.

[10] CoppeliaSim User Manual. *Designing dynamic simulations*. URL: https://www.coppeliarobotics.com/helpFiles/en/designingDynamicSimulations.htm.

[11] CoppeliaSim User Manual. *Joint dynamics properties*. URL: https://www.coppeliarobotics.com/helpFiles/en/jointDynamicsProperties.htm.

[12] CoppeliaSim User Manual. *Enabling the remote API - client side*. URL: https://www.coppeliarobotics.com/helpFiles/en/remoteApiClientSide.htm.

[13] CoppeliaSim User Manual. *Enabling the remote API - server side*. URL: https://www.coppeliarobotics.com/helpFiles/en/remoteApiServerSide.htm#continuousRemoteApiService.

[14] CoppeliaSim User Manual. *Legacy remote API*. URL: https://www.coppeliarobotics.com/helpFiles/en/legacyRemoteApiOverview.htm.

[15] CoppeliaSim User Manual. *Remote API modus operandi*. URL: https://www.coppeliarobotics.com/helpFiles/en/remoteApiModusOperandi.htm#synchronous.

[16] CoppeliaSim User Manual. *Remote API functions (Matlab)*. URL: https://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsMatlab.htm.

[17] CoppeliaSim User Manual. *Remote API Constants*. URL: https://www.coppeliarobotics.com/helpFiles/en/remoteApiConstants.htm.

[18] Mance E Harmon e Stephanie S Harmon. *Reinforcement Learning: A Tutorial*. Rapp. tecn. WRIGHT LAB WRIGHT-PATTERSON AFB OH, 1997.

[19] MathWorks. *regionprops-Measure properties of image regions*. URL: https://it.mathworks.com/help/images/ref/regionprops.html.

[20] MathWorks. *Train DDPG Agent to Swing Up and Balance Cart-Pole System*. URL: https://it.mathworks.com/help/reinforcement-learning/ug/train-ddpg-agent-to-swing-up-and-balance-cart-pole-system.html.

[21] MathWorks. *Deep Deterministic Policy Gradient Agents*. URL: https://it.mathworks.com/help/reinforcement-learning/ug/ddpg-agents.html.

[22] MathWorks. *rlDDPGAgentOptions*. URL: https://it.mathworks.com/help/reinforcement-learning/ref/rlddpgagentoptions.html.

Ringraziamenti

E finalmente ci sono riuscito pure io ad arrivare alla fine di questo percorso. A volte penso quanto sia stato faticoso, quanti esami falliti e quante volte ho pensato di non essere all'altezza, ma ci sono riuscito. Se sono arrivato fino a questo punto è merito anche del mio relatore Gianfranco Fenu che mi ha aiutato fino alla fine, mi ha supportato e sopportato durante questo lungo percorso e vorrei ringraziarlo per tutto l'impegno che ci ha messo. Vorrei però ringraziare anche i ragazzi del laboratorio per tutto l'aiuto che mi hanno fornito, in particolare Erica che anche dopo aver detto "ciclo if" ha continuato ad aiutarmi.
A volte invece penso che se io sono riuscito a fare questa università è merito solamente dei miei genitori, che decisero di non iscrivermi all'alberghiero di mandarmi a fare il liceo, e quindi di aprirmi la strada verso l'università. Grazie a loro che sono qui e penso che siano state le persone che più mi hanno aiutato durante tutti questi anni, che credevano in me quando nemmeno io ci credevo. Vi ringrazio con tutto il cuore e se leggete questa pagina sappiate che vi voglio bene. Ringrazio pure i miei due fratelli che mi hanno aiutato anche con un semplice consiglio, ma sempre fondamentale.
A Federico e Mathias un sincero e caloroso grazie, perché anche se siamo stati distanti in questi ultimi tre anni, siamo riusciti comunque a vederci e a fare una vacanza assieme (e che vacanza!).
Ai miei ex compagni del liceo vorrei dire ragazzi grazie di tutto, per le sessioni di studio in biblioteca, per le serate passate e per la bellissima vacanza passata assieme (questa è un'altra)
Con i ragazzi del corso penso di aver passato gran parte di questi tre anni, sia sui banchi dell'università sia dopo per studiare assieme.In particolare il team Stepinscolas, tre ragazzi (di cui uno il sottoscritto) con la voglia di fare tutto, ma anche nulla. Sempre a tirarci su il morale a vicenda, e tra un pisolino e l'altro ci si trovava in qualche aula a studiare per un qualche esame. Alla fine siamo riusciti finalmente a portare a casa i risultati.
Questi anni di università mi hanno fatto stringere legami con persone che ora considero come una seconda famiglia. Anche se quasi ogni fine settimana tornavo a casa, la settimana passata in una stanza tutto da solo non era il massimo. Con i ragazzi della casa dello studente ho passato delle bellissime giornate e serate, vi voglio bene ragazzi dal primo all'ultimo. E in particolare vorrei ringraziare Pietro il quale mi è stato vicino e mi ha sempre dato una mano.
Ai miei 2 nuovi coinquilini vorrei dire semplicemente che con voi penso di aver passato le migliori e le più faticose giornate in aula studio, ogni giorno a fare chiusura, ma ne è valsa la pena. Vi voglio bene ragazzi.