PROGETTO ROBOTICA – RELAZIONE (PDDLSTREAM)

Enrico Zardini – matr. 731543

Introduzione

Task and Motion Planning (TAMP) rappresenta una delle sfide fondamentali nella robotica autonoma: la necessità di integrare la pianificazione simbolica ad alto livello ("cosa fare") con la pianificazione geometrica a basso livello ("come farlo"). Mentre i pianificatori classici PDDL eccellono nel ragionamento simbolico su azioni discrete, falliscono quando le variabili del problema appartengono a spazi continui — come le pose degli oggetti, le configurazioni del robot o le traiettorie di movimento.
PDDLStream (Garrett et al., 2020) estende il linguaggio PDDL introducendo il concetto di stream: generatori procedurali che producono valori continui on-demand, permettendo al pianificatore simbolico di ragionare su spazi infiniti senza doverli discretizzare a priori. Questa architettura separa il cosa (dominio PDDL) dal come (funzioni Python), delegando la generazione di pose, configurazioni e traiettorie a procedure esterne.

1.0 Obiettivi del Progetto

Questo lavoro presenta due implementazioni del problema Continuous_Tamp — un dominio di manipolazione 2D con blocchi, regioni e una piattaforma sopraelevata — con l'obiettivo di:

1. Illustrare i meccanismi di PDDLStream attraverso un esempio minimale ma completo (continuous_tamp)
2. Dimostrare l'integrazione con algoritmi di motion planning realistici (continuous_tamp_RRT)
3. Confrontare i due approcci in termini di completezza, efficienza e applicabilità

Le Due Implementazioni

continuous_tamp — Approccio Didattico
La prima implementazione utilizza una strategia di motion planning parametrica: il robot segue sempre una traiettoria predefinita "lift-move-drop" a quota fissa. Questa scelta, pur essendo geometricamente naive, permette di focalizzare l'attenzione sui concetti chiave di PDDLStream:

- Stream generators (s-region, s-gasp, s-ik) per la generazione di valori continui
- Test streams (t-cfree, t-on-ground) per la verifica di vincoli geometrici
- Fluents per passare lo stato corrente agli stream (AtPose, AtGrasp)
- Il ciclo plan-fail-sample-replan dell'algoritmo Focused

Il valore didattico risiede nel mostrare come PDDLStream orchestra la generazione di pose, il calcolo dell'inverse kinematics e la verifica delle collisioni — non nella sofisticatezza del motion planner.

continuous_tamp_RRT — Approccio più Realistico
La seconda implementazione sostituisce la traiettoria parametrica con un motion planner basato su RRT (Rapidly-exploring Random Tree). Questa scelta è motivata da:

- Completezza: RRT è probabilisticamente completo e può trovare percorsi in spazi con ostacoli complessi
- Adattività: il planner considera dinamicamente la configurazione corrente (blocchi posizionati, oggetto trasportato)
- Realismo: la piattaforma gialla diventa un ostacolo fisico che il robot deve aggirare [Figura 1]

L'implementazione segue l'algoritmo originale di LaValle (1998) con goal bias al 15% per accelerare la convergenza.

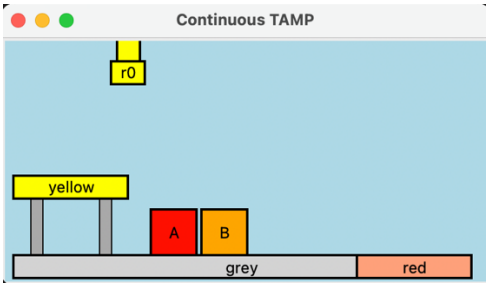

Figura 1 - problema: new_problem: da pose iniziali definite randomicamente, il robot r0 ha il goal di spostare il blocco B nella regione red e il blocco A nella regione yellow

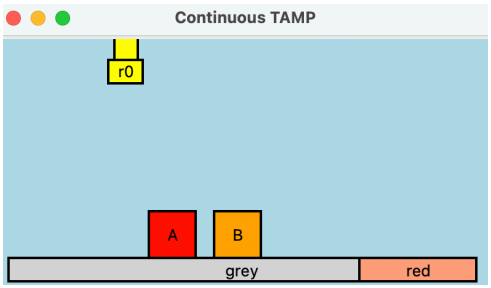

Figura 2 - problema tight: da pose iniziali dei blocchi definite randomicamente, il robot r0 ha il goal di posizionare sovrapposti il blocco A e B nella regione red

2. Background

2.1 Il Linguaggio PDDL

Il Planning Domain Definition Language (PDDL) è un linguaggio formale introdotto da McDermott et al. come standard per la specifica di problemi di pianificazione automatica. Un problema PDDL è costituito da due componenti distinte: il domain, che definisce il vocabolario del problema attraverso predicati e schemi d'azione, e il problem, che specifica lo stato iniziale, gli oggetti e la formula di goal.

Un'azione è formalmente descritta da una tripla ⟨params, pre, eff⟩, dove params rappresenta i parametri, pre le precondizioni che devono essere valide per ottenere gli eff, ovvero gli effetti. Si consideri, a titolo esemplificativo, l'azione di presa di un oggetto:

```
(:action pick
 :parameters (?r - robot ?b - block ?p - pose)
 :precondition (and (at ?r ?p) (on ?b ?p) (hand-empty ?r))
 :effect (and (holding ?r ?b) (not (on ?b ?p)) (not (hand-empty ?r))))
```

Il pianificatore opera mediante grounding: ogni variabile nelle azioni viene sostituita con tutti gli oggetti compatibili, generando l'insieme delle azioni istanziate sulle quali effettuare la ricerca. Tale architettura impone un vincolo fondamentale: tutti gli oggetti devono essere enumerati esplicitamente nella definizione del problema.

Nei domini robotici questa limitazione risulta insostenibile. Nel dominio continuous_tamp:

- Le pose dei blocchi hanno una componente x continua, con infinite possibilità all'interno dell'intervallo di campionamento.
- Le traiettorie sono sequenze di configurazioni in $\mathbb{R}^2$, appartenenti a uno spazio infinito-dimensionale.

Le configurazioni del robot sono rappresentate in $\mathbb{R}^2$ (posizione x, y dell'effettore). Nell'approccio parametrico, le traiettorie sono vincolate a pattern geometrici predefiniti (lift–move–drop), mentre nell'implementazione RRT lo spazio delle configurazioni è esplorato liberamente. L'enumerazione a priori di tali oggetti risulta quindi impossibile, precludendo l'applicazione diretta di pianificatori PDDL classici.

2.2 Estensioni temporali: PDDL 2.1

L'estensione PDDL 2.1, proposta da Fox e Long, introduce i fluents numerici e le azioni durative, consentendo la modellazione di variabili quantitative e vincoli temporali. Le clausole at start, at end e over all permettono di specificare condizioni ed effetti in diversi istanti dell'azione.

PDDL 2.1 è particolarmente efficace per pianificare sequenze di azioni che rispettano vincoli su risorse numeriche, quali carburante, energia o tempo. Tuttavia, le azioni devono operare tra stati discreti predefiniti: ad esempio, drive(cityA, cityB) presuppone che le città siano enumerate esplicitamente.

Nel contesto del motion planning robotico, questa limitazione diventa critica. Una traiettoria geometrica — intesa come sequenza di configurazioni nello spazio continuo — non può essere enumerata a priori: i waypoint intermedi che evitano gli ostacoli devono essere calcolati dinamicamente mediante algoritmi dedicati (ad es. RRT). PDDL 2.1 non fornisce meccanismi per invocare tali algoritmi durante la pianificazione.

2.3 PDDL+

PDDL+ estende ulteriormente il linguaggio introducendo process ed events per la modellazione di sistemi ibridi discreto–continui. Un process descrive un cambiamento continuo attivo fintantoché una determinata condizione rimane soddisfatta:

```
(:process moving
 :parameters (?r - robot)
 :precondition (and (engine-on ?r)
                    (< (position ?r) (target ?r)))
 :effect (increase (position ?r)
                   (* #t (velocity ?r))))
```

Un event rappresenta invece una transizione istantanea innescata dal verificarsi di una condizione, ad esempio l'arresto del robot al raggiungimento della destinazione. Malgrado questa evoluzione PDDL+ risulta espressivo per domini con dinamiche fisiche semplici. I pianificatori numerici che lo supportano gestiscono efficacemente problemi con dinamiche lineari o polinomiali. Tuttavia, come osservato da Garrett et al., sebbene sia teoricamente possibile modellare vincoli geometrici complessi (ad esempio, collision checking tra oggetti) mediante equazioni numeriche in PDDL+, l'encoding risultante sarebbe di dimensioni enormi, ben oltre le capacità dei pianificatori numerici attuali.

Nel dominio continuous_tamp, la verifica di collisioni tra blocchi, robot e piattaforma richiederebbe la codifica esplicita di disequazioni per ogni coppia di oggetti e per ogni configurazione lungo la traiettoria predefinita — un approccio impraticabile. PDDLStream risolve questo problema delegando la verifica geometrica a funzioni procedurali (test stream) invocate on-demand durante la pianificazione.

2.4 PDDLStream: integrazione con algoritmi geometrici

PDDLStream, proposto da Garrett et al., supera quindi le limitazioni descritte introducendo il concetto di stream: costrutti procedurali che generano oggetti on-demand mediante funzioni esterne. Uno stream è definito dalla tupla $\langle X, Y, D_s, F_s, C_s \rangle$ dove:

- X denota le variabili di input;
- Y le variabili di output;
- $D_s(X)$ i predicati richiesti sugli input;
- $F_s : X \rightarrow 2^Y$ la funzione generatrice;
- $C_s(X, Y)$ i predicati certificati per ogni output.

Nel dominio continuous_tamp, gli stream consentono l'integrazione di algoritmi geometrici:

Stream	Funzione	Oggetti generati
s-region	Campionamento pose	Pose continue in $\mathbb{R}^2$
s-grasp	Calcolo grasp	Offset di presa
s-ik	Cinematica inversa	Configurazioni del robot
s-motion	Motion planning (es. RRT)	Traiettorie collision-free
t-cfree	Collision checking	Certificati di non-collisione

2.4.1 Tipologie di Stream

In PDDLStream i stream si differenziano in base alla natura dell'output che producono e al modo in cui contribuiscono alla pianificazione: ci sono generatori che, data una condizione di input, producono molteplici valori possibili come pose, configurazioni o grasp che servono a esplorare lo spazio delle soluzioni; ci sono funzioni che, sempre a partire da input specifici, calcolano e restituiscono un unico valore deterministico come una configurazione di cinemática inversa; esistono test che non generano nuovi oggetti ma verificano una condizione e certificano un fatto booleano utile come la validità di una traiettoria rispetto alla collisione; e infine funzioni di costo che ritornano un valore numerico associato a una metrica come distanza o durata da usare nelle precondizioni o nei fluenti numerici del dominio.

2.4.2 Fluents negli Stream

Gli stream possono accedere allo stato corrente del mondo attraverso il meccanismo dei fluents:

```
pddl
(:stream s-motion
  :inputs (?q1 ?q2)
  :domain (and (Conf ?q1) (Conf ?q2))
  :fluents (AtPose AtGrasp)
  :outputs (?t)
  :certified (and (Motion ?q1 ?t ?q2) (Traj ?t)))
```

All'invocazione dello stream s-motion, PDDLStream fornisce automaticamente l'insieme di tutti i fatti AtPose e AtGrasp correntemente veri. Tale meccanismo consente al motion planner di conoscere la configurazione di tutti gli oggetti nell'ambiente e pianificare conseguentemente traiettorie prive di collisioni.

2.4.3 Algoritmi di Risoluzione

PDDLStream implementa diversi algoritmi per l'orchestrazione della generazione di valori e la ricerca del piano. Il Focused Algorithm, adottato nel presente lavoro, implementa una strategia efficiente:

- Individuazione di uno skeleton del piano contenente variabili libere generato da Fast Downward
- Identificazione degli stream potenzialmente in grado di produrre valori per tali variabili
- Invocazione selettiva dei soli stream necessari
- Validazione del piano con i valori concreti generati

Il Focused Algorithm evita la generazione indiscriminata di valori, focalizzando le risorse computazionali su quelli effettivamente richiesti dalla struttura del piano.

2.5 Algoritmo di Motion Planning

Rapidly-exploring Random Trees

L'algoritmo RRT, introdotto da LaValle, costruisce incrementalmente un albero che esplora rapidamente lo spazio delle configurazioni. Lo pseudocodice è il seguente:

Algoritmo 1: RRT(q_{init}, q_{goal}, K)

```
T.init(q_init)
for k ← 1 to K do
  q_rand ← RANDOM_CONFIG()
  q_near ← NEAREST(T, q_rand)
  q_new ← EXTEND(q_near, q_rand, Δ)
  if COLLISION_FREE(q_near, q_new) then
    T.add_vertex(q_new)
    T.add_edge(q_near, q_new)
    if DISTANCE(q_new, q_goal) < ε then
      return PATH(q_init, q_goal)
return FAILURE
```

L'algoritmo RRT implementato nel motion planner del problema TAMP sfrutta due meccanismi di esplorazione:
Completezza probabilistica: RRT garantisce che, qualora esista un percorso libero da collisioni tra configurazione iniziale e goal, la probabilità di trovarlo aumenta monotonicamente con il numero di iterazioni. Questa proprietà è critica nel contesto di motion planning con ostacoli dinamici (blocchi sulla tavola), poiché assicura che il pianificatore possa determinare in modo affidabile l'esistenza o l'inesistenza di una traiettoria sicura entro un tempo limitato.
Goal bias per convergenza accelerata: piuttosto che affidarsi unicamente al bias naturale di esplorazione, l'implementazione impiega una strategia di goal bias con probabilità $p_g = 0.15$. Con questa frequenza (15% dei campionamenti), il campione casuale viene sostituito direttamente dalla configurazione goal q_2, bypassando il campionamento uniforme dello spazio. Questo approccio accelera significativamente la convergenza verso l'obiettivo mantenendo un'esplorazione efficiente: nei restanti 85% dei campioni, la ricerca mantiene

l'esplorazione dello spazio libero attraverso il bias naturale verso regioni inesplorate, garantendo robustezza anche in ambienti complessi con ostacoli ravvicinati.

2.6 Integrazione nel Framework Proposto

Nel contesto del presente lavoro, gli algoritmi di motion planning sono incapsulati nello stream s-motion e resi disponibili a PDDLStream attraverso il meccanismo della stream_map:

```
stream_map = {
  's-motion': from_fn(plan_motion),
  ...
}
```

La funzione plan_motion riceve come argomenti le configurazioni iniziale e finale (q_1, q_2) e l'insieme dei fluents rappresentanti lo stato corrente del mondo. Restituisce una traiettoria $\tau = [q_1, \dots, q_n, q_2]$ priva di collisioni, oppure None qualora tale traiettoria non esista. Tale architettura consente la sostituzione dell'algoritmo di motion planning senza modifiche al dominio PDDL o alla logica di PDDLStream. Nel prosieguo, verrà mostrata questa proprietà per confrontare due implementazioni: una basata su traiettorie parametriche (continuous_tamp) e una basata sull'algoritmo RRT (continuous_tamp_RRT).

3. Descrizione del Dominio

3.1 Scenario Applicativo

Il dominio considerato nel presente lavoro è un problema di manipolazione bidimensionale. L'ambiente è costituito da:

- Un manipolatore robotico rappresentato come un punto con end-effector a ventosa
- Un insieme di blocchi rigidi di dimensioni uniformi
- Regioni planari identificate da intervalli sull'asse X
- Una piattaforma sopraelevata che funge da superficie di appoggio alternativa al piano di lavoro

L'obiettivo del sistema è pianificare ed eseguire una sequenza di azioni (movimento, presa, rilascio) per riposizionare i blocchi dalla configurazione iniziale a quella desiderata, rispettando i vincoli geometrici imposti dall'ambiente.

3.2 Regioni e Vincoli Spaziali

L'ambiente è suddiviso in tre regioni distinte:

- Ground (grigio): regione principale sul piano di lavoro, x in [-10, 10]
- Goal (rosso): sottoregione target, x in [5, 10]
- Platform (giallo): superficie sopraelevata con x in [-10, -5] e y in [2.5, 3.5], con top a y = 3.5

3.3 Convenzioni sulle Coordinate

Un aspetto da sottolineare nella modellazione geometrica riguarda la convenzione adottata per la rappresentazione delle pose dei blocchi. Nel presente lavoro sono state implementate due convenzioni distinte:

Convenzione centro-riferita (continuous_tamp):

$$\mathbf{p} = (\mathbf{p_x}, \mathbf{p_y}) \text{ dove } \mathbf{p_y} \text{ è l'ordinata del centro geometrico}$$

Convenzione fondo-riferita (continuous_tamp_RRT):

$$\mathbf{p} = (\mathbf{p_x}, \mathbf{p_y}) \text{ dove } \mathbf{p_y} \text{ è l'ordinata della faccia inferiore}$$

La convenzione fondo-riferita semplifica il ragionamento sulla stabilità e l'appoggio: un blocco è appoggiato su una superficie a quota y_s se e solo se p_y = y_s. Tale proprietà risulta vantaggiosa nella verifica dell'appoggio sulla piattaforma sopraelevata. Il bounding box di un blocco nelle due convenzioni è rispettivamente:

$$\text{Centro: } \mathcal{B}(\mathbf{p}) = \left[\mathbf{p_x} - \frac{\mathbf{w_b}}{2}, \mathbf{p_x} + \frac{\mathbf{w_b}}{2}\right] \times \left[\mathbf{p_y} - \frac{\mathbf{h_b}}{2}, \mathbf{p_y} + \frac{\mathbf{h_b}}{2}\right]$$

$$\text{Fondo: } \mathcal{B}(\mathbf{p}) = \left[\mathbf{p_x} - \frac{\mathbf{w_b}}{2}, \mathbf{p_x} + \frac{\mathbf{w_b}}{2}\right] \times \left[\mathbf{p_y}, \mathbf{p_y} + \mathbf{h_b}\right]$$

3.4 Definizione del Dominio PDDL

Il dominio è formalizzato in PDDL attraverso la specifica di predicati e azioni. I predicati si suddividono in tre categorie:

Predicati statici (invarianti durante l'esecuzione):

- (Robot ?r), (Block ?b), (Region ?s): definizione degli oggetti
- (Placeable ?b ?s): vincolo di posizionamento ammissibile

Predicati derivati (certificati dagli stream):

- (Pose ?b ?p), (Grasp ?b ?g), (Conf ?q), (Traj ?t): validità degli oggetti continui
- (Kin ?b ?q ?p ?g): relazione cinematica tra configurazione, posa e grasp
- (Motion ?q1 ?t ?q2): validità della traiettoria tra due configurazioni
- (CFree ?b1 ?p1 ?b2 ?p2): assenza di collisione tra due blocchi
- (Contain ?b ?p ?s): contenimento di una posa in una regione
- (PoseOnGround ?p), (PoseOnPlatform ?p): validità della superficie di appoggio
- (StackPose ?p_a ?p_b): relazione di impilamento

Predicati fluents (variabili durante l'esecuzione):

- (AtPose ?b ?p), (AtConf ?r ?q), (AtGrasp ?r ?b ?g): stato corrente
- (HandEmpty ?r), (Clear ?b): condizioni di manipolabilità
- (CanMove ?r), (CanManipulate ?r): abilitazioni alle azioni
- (In ?b ?s), (On ?b_a ?b_b): relazioni spaziali tra oggetti

3.5 Azioni del Dominio

Il dominio definisce cinque schemi d'azione che coprono l'intero repertorio di primitive robotiche necessarie.

3.5.1 Azione Move

L'azione move realizza il movimento del robot da una configurazione a un'altra seguendo una traiettoria t. Il predicato (Motion ?q1 ?t ?q2) certifica che la traiettoria τ connette q_1 a q_2 senza collisioni. I predicati CanMove e CanManipulate implementano un semplice vincolo di sequenzialità: dopo un movimento il robot può manipolare, dopo una manipolazione il robot può muoversi. Tale schema previene comportamenti degeneri quali pick e place consecutivi nella stessa posizione.

3.5.2 Azione Pick

L'azione pick realizza la presa di un blocco: La precondizione (Kin ?b ?q ?p ?g) garantisce la consistenza cinematica: la configurazione q del robot, combinata con il grasp g, consente di raggiungere la posa p del blocco. L'effetto condizionale rimuove il blocco da tutte le regioni, poiché durante il trasporto esso non appartiene ad alcuna di esse.

3.5.3 Azioni Place-Ground e Place-Platform

Il rilascio del blocco è suddiviso in due azioni distinte in base alla superficie di destinazione.

```
(:action place-ground
 :parameters (?r ?b ?p ?g ?q)
 :precondition (and (Robot ?r) (Kin ?b ?q ?p ?g) (AtConf ?r ?q)
            (AtGrasp ?r ?b ?g) (PoseOnGround ?p))
 :effect (and (AtPose ?b ?p) (HandEmpty ?r) (CanMove ?r) (Clear ?b)
        (not (AtGrasp ?r ?b ?g)) (not (CanManipulate ?r))
        (forall (?s) (when (Contain ?b ?p ?s) (In ?b ?s)))
        (increase (total-cost) (Cost))))

(:action place-platform
 :parameters (?r ?b ?p ?g ?q)
 :precondition (and (Robot ?r) (Kin ?b ?q ?p ?g) (AtConf ?r ?q)
            (AtGrasp ?r ?b ?g) (PoseOnPlatform ?p))
 :effect (...))  ; analogo a place-ground
```

La distinzione tra place-ground e place-platform è motivata dalla necessità di verificare la validità della superficie di appoggio attraverso i predicati PoseOnGround e PoseOnPlatform. Tale separazione rende esplicito nel piano simbolico il tipo di posizionamento effettuato.

3.5.4 Azione Stack

L'azione stack realizza l'impilamento di un blocco sopra un altro. Il predicato (StackPose ?p_a ?p_b) certifica che la posa $\mathbf{p_a}$ rappresenta un posizionamento stabile sopra un blocco in posa $\mathbf{p_b}$. L'effetto condizionale propaga l'appartenenza regionale: se il blocco inferiore è nella regione goal, anche quello superiore vi apparterrà dopo l'impilamento.

4. Implementazione degli Stream e del Motion Planner

4.1 Architettura del Sistema

L'architettura software segue il pattern di separazione proposto da PDDLStream, distinguendo tre livelli:

- Livello simbolico: file domain.pddl e stream.pddl
- Livello procedurale: modulo primitives.py con le implementazioni Python
- Livello di orchestrazione: modulo run.py con la configurazione di PDDLStream

La corrispondenza tra stream dichiarati e funzioni Python è stabilita attraverso la stream_map.

```
stream_map = {
   's-grasp': from_fn(lambda b: (GRASP,)),
   's-region': from_gen_fn(region_gen_wrapper),
   's-ik': from_fn(inverse_kin_fn),
   's-motion': from_fn(plan_motion),
   't-cfree': from_test(lambda *args: not collision_test(*args)),
   't-on-ground': from_test(get_on_ground_test()),
   't-on-platform': from_test(get_on_platform_test()),
   'dist': distance_fn,
}
```

4.2 Stream di Generazione

4.2.1 Generazione delle Pose (s-region)

Lo stream s-region genera pose ammissibili per un blocco all'interno di una regione specificata.

```
(:stream s-region
 :inputs (?b ?r)
 :domain (Placeable ?b ?r)
 :outputs (?p)
 :certified (and (Contain ?b ?p ?r) (Pose ?b ?p)))
```

L'implementazione adotta una strategia di campionamento uniforme sull'intervallo della regione.

```
def get_pose_gen(regions):
   def gen_fn(block, region):
      x1, x2 = regions[region]
      y = PLATFORM_TOP if region == YELLOW_NAME else 0.0
      for _ in range(100):
         x = np.random.uniform(x1, x2)
       yield (np.array([x, y]),)
   return gen_fn
```

La quota $\mathbf{y}$ è determinata dalla regione: $\mathbf{y} = \mathbf{0}$ per il piano di lavoro, $\mathbf{y} = \mathbf{y_{platform}}$ per la piattaforma sopraelevata.

4.2.2 Generazione del Grasp (s-grasp)

Nel dominio considerato, l'end-effector a ventosa consente un unico tipo di presa (top-grasp). Lo stream restituisce pertanto un valore costante. Il vettore grasp rappresenta l'offset tra la posizione del robot e quella del blocco afferrato. Il segno negativo indica che il robot si trova al di sopra del blocco.

4.2.3 Cinematica Inversa (s-ik)

Lo stream s-ik calcola la configurazione del robot necessaria per raggiungere una data posa con un dato grasp. La semplicità della cinematica inversa deriva dalla natura del dominio bidimensionale: il robot è modellato come un punto e la cinematica si riduce a una traslazione.

4.2.4 Generazione di Pose per Impilamento (s-stack)

Lo stream s-stack genera pose valide per l'impilamento di un blocco sopra un altro.

```
def get_stack_gen(x_radius=0.5, y_eps=0.02, method='uniform'):
  def gen_fn(b_above, b_below, p_below):
    if not (-0.1 <= p_below[1] <= 0.1):  # solo blocchi a terra
      return
    cx, target_y = float(p_below[0]), float(BLOCK_HEIGHT)
    while True:
      x = np.random.uniform(cx - x_radius, cx + x_radius)
      y = target_y + np.random.uniform(-y_eps, y_eps)
      yield (np.array([x, y]),)
  return gen_fn
```

Il parametro x_radius controlla la tolleranza orizzontale nell'allineamento dei blocchi, mentre y_eps introduce una piccola variazione verticale per migliorare la robustezza numerica. La condizione iniziale verifica che il blocco inferiore sia effettivamente posizionato a terra.

4.3 Stream di Test

4.3.1 Test di Collisione (t-cfree)

Lo stream t-cfree verifica l'assenza di collisione tra due blocchi.

```
def collision_test(b1, p1, b2, p2):
  if b1 == b2:
    return False
  return boxes_overlap(get_block_box(b1, p1),
                       get_block_box(b2, p2),
                       tol=0.01
```

Il predicato (CFree ?b1 ?p1 ?b2 ?p2) è certificato quando collision_test restituisce False.

4.3.2 Test di Superficie (t-on-ground, t-on-platform)

I test di superficie verificano la validità della quota di appoggio della posa.

```
def get_on_ground_test():
```

```
  def test(b, p):
    return abs(p[1] - 0.0) < 0.1
  return test
def get_on_platform_test():
  def test(b, p):
    return abs(p[1] - PLATFORM_TOP) < 0.1
  return test
```

La tolleranza di 0.1 unità consente di assorbire eventuali imprecisioni nella generazione delle pose.

4.4 Stream di Movimento

Lo stream s-motion costituisce il nucleo del sistema e presenta differenze sostanziali tra le due implementazioni considerate.

4.4.1 Implementazione Parametrica (continuous_tamp)

La prima implementazione adotta una strategia di traiettoria fissa di tipo lift–move–drop. La traiettoria è composta da quattro waypoint: posizione iniziale, sollevamento a quota $\mathbf{y_{carry}}$, traslazione orizzontale e discesa alla posizione finale. La funzione swept_collision_check verifica che lungo l'intero percorso il blocco trasportato non intersechi gli ostacoli statici.

Vantaggi

- semplicità implementativa;
- comportamento deterministico;
- tempo di calcolo costante $\mathbf{O(n)}$, dove $\mathbf{n}$ è il numero di ostacoli.

Limitazioni

- fallisce in presenza di ostacoli alla quota $\mathbf{y_{carry}}$;
- non minimizza la lunghezza del percorso.

4.4.2 Implementazione RRT (continuous_tamp_RRT)

La seconda implementazione adotta l'algoritmo RRT per la generazione di traiettorie collision-free in presenza di ostacoli arbitrari.

Parsing dei fluents

```
def plan_motion(q1, q2, fluents):
  carried_block, grasp = None, None
  placed_blocks = {}

  for fluent in fluents:
    name = fluent[0].lower()
    if name == 'atgrasp':
      carried_block, grasp = fluent[2], np.array(fluent[3])
    elif name == 'atpose':
      placed_blocks[fluent[1]] = np.array(fluent[2])
```

```
  if carried_block in placed_blocks:
    del placed_blocks[carried_block]
```

Funzione di collision checking

```
def collides_at(robot_conf):
  # Bounding box del robot
  robot_box = (robot_conf - [ROBOT_WIDTH/2, ROBOT_HEIGHT/2],
          robot_conf + [ROBOT_WIDTH/2, ROBOT_HEIGHT/2])

  # Check collisione robot-blocchi
  for obs_pos in placed_blocks.values():
    if boxes_overlap(robot_box, get_block_box(None, obs_pos)):
      return True

  # Check collisione robot-piattaforma
  if overlaps_platform(robot_box):
    return True

  # Check collisione blocco trasportato (se presente)
  if is_carrying:
    block_pos = robot_conf + grasp
    block_box = get_block_box(None, block_pos)
    for obs_pos in placed_blocks.values():
      if boxes_overlap(block_box, get_block_box(None, obs_pos)):
        return True
    if overlaps_platform(block_box):
      return True

  return False
```

La piattaforma gialla è modellata esplicitamente come ostacolo:

$$\mathcal{O}_{\mathbf{platform}} = [-\mathbf{10}, -\mathbf{5}] \times [\mathbf{2.5}, \mathbf{3.5}]$$

4.4.2 a - Strategia Gerarchica

L'algoritmo implementa una strategia a due livelli per bilanciare efficienza e completezza:

1. Percorso diretto: verifica se il segmento $\mathbf{q_1}\bar{\mathbf{q}}_\mathbf{2}$ è collision-free
2. RRT: se il metodo precedente fallisce, avvia l'esplorazione stocastica

Implementazione RRT

I parametri dell'algoritmo sono stati impostati in modo da bilanciare efficacia e costi computazionali: il limite max_iter = 2000 offre un numero di tentativi sufficiente per esplorare lo spazio delle configurazioni in 2D quando ci sono pochi ostacoli, garantendo un buon compromesso tra possibilità di trovare una soluzione e tempo di calcolo complessivo;
il passo di espansione step = 0.5 è scelto come equilibrio tra dettaglio della traiettoria e velocità di crescita dell'albero, evitando di generare nodi troppo ravvicinati o troppo distanti;
infine un goal_bias = 0.15 introduce una lieve preferenza verso il goal durante il

campionamento, un valore tipico adottato in letteratura per migliorare l'efficienza di RRT senza compromettere la proprietà di completezza probabilistica del planner.
Questo approccio conserva i principali vantaggi di RRT, come la capacità di affrontare ostacoli arbitrari esplorando lo spazio di configurazione in modo probabilisticamente completo e adattandosi alla configurazione corrente, ma mantiene anche alcune limitazioni tipiche: il tempo di calcolo può variare sensibilmente con la difficoltà dell'ambiente e i percorsi trovati possono essere subottimali.

5. Risultati Sperimentali e Confronto

5.1 Setup Sperimentale

Il problema di test new_problem prevede:
Due blocchi (A e B) posizionati casualmente nell'intervallo $x \in [-4, 5]$
L'obiettivo consiste nel:

- posizionare il blocco A nella regione gialla (piattaforma)
- posizionare il blocco B nella regione rossa

Configurazione iniziale del robot:

$q_0 = (-5, 8)$

5.2 Metriche di Valutazione

Sono state considerate le seguenti metriche:

- Tempo di pianificazione (T_{plan}): tempo totale richiesto per la generazione del piano
- Costo del piano (C): somma dei costi associati alle azioni pianificate
- Lunghezza delle traiettorie (L_{traj}): distanza totale percorsa dal robot
- Tasso di successo (ρ): percentuale di istanze risolte entro il timeout prefissato

5.3 Risultati Quantitativi

La Tabella 3 riporta i risultati medi ottenuti su 10 esecuzioni con semi casuali differenti.

Metrica	continuous_tamp	continuous_tamp_RRT
T_{plan} (s)	2.3 ± 0.8	3.5 ± 1.5
C	54.2 ± 5.1	55.3 ± 5.8
L_{traj}	48.5 ± 4.2	52.4 ± 7.3
ρ	100%	100%

5.4 Analisi dei Risultati

5.4.1 Tempo di Pianificazione

L'implementazione parametrica risulta mediamente più veloce del 34%. Questa differenza è attribuibile al costo computazionale dell'algoritmo RRT, che richiede in media tra 150 e 300 iterazioni per convergere verso una soluzione valida. L'approccio parametrico, invece, calcola direttamente una traiettoria di tipo lift–move–drop senza necessità di esplorare lo spazio delle configurazioni.

5.4.2 Qualità delle Traiettorie

Le traiettorie generate mediante RRT risultano mediamente più lunghe (circa +8%) e presentano una maggiore variabilità, come indicato dalla deviazione standard più elevata (7.3 contro 4.2). Questo comportamento è intrinseco alla natura stocastica dell'algoritmo: RRT esplora lo spazio delle configurazioni tramite campionamento casuale, producendo percorsi che non sono necessariamente ottimali in termini di lunghezza. L'implementazione parametrica, al contrario, genera traiettorie geometricamente prevedibili, composte da segmenti verticali e orizzontali che seguono un pattern deterministico di tipo lift–move–drop.

5.4.3 Generalizzabilità

Nonostante prestazioni leggermente inferiori nel dominio bidimensionale considerato, l'approccio basato su RRT offre vantaggi rilevanti in termini generali:

- Estensibilità dimensionale: RRT scala naturalmente a spazi di configurazione ad alta dimensionalità, come nel caso di bracci robotici con molti gradi di libertà
- Gestione di ambienti complessi: consente di trattare ostacoli di forma arbitraria senza ricorrere a strategie specifiche
- Completezza probabilistica: garantisce il ritrovamento di una soluzione, se esistente, dato un numero sufficiente di iterazioni

L'approccio parametrico, sebbene particolarmente efficiente nel caso 2D analizzato, richiederebbe una riprogettazione significativa per essere applicato a scenari più complessi.

6. Conclusioni

6.1 Sintesi del Lavoro

Il presente lavoro ha affrontato il problema dell'integrazione tra pianificazione simbolica e ragionamento geometrico nel contesto del Task and Motion Planning. Attraverso lo sviluppo e il confronto di due implementazioni del dominio Continuous TAMP, sono stati evidenziati i meccanismi fondamentali del framework PDDLStream e le sfide computazionali associate alla pianificazione di movimento in spazi continui.

L'implementazione continuous_tamp ha dimostrato come PDDLStream consenta di separare efficacemente il livello simbolico (espresso in PDDL) dal livello procedurale (implementato in Python), delegando la generazione di valori continui a stream specializzati. La strategia di motion planning parametrica adottata, sebbene geometricamente naive, si è rivelata sufficiente per illustrare i concetti chiave del framework: stream generators, test streams, fluents e il ciclo plan–fail–sample–replan dell'algoritmo Focused.

L'implementazione continuous_tamp_RRT ha esteso il sistema introducendo un motion planner basato sull'algoritmo Rapidly-exploring Random Trees. Tale estensione ha richiesto modifiche esclusivamente al livello procedurale, confermando l'efficacia dell'architettura modulare proposta da PDDLStream. L'algoritmo RRT, implementato senza dipendenze da librerie esterne, ha garantito completezza probabilistica e la capacità di gestire ostacoli arbitrari, inclusa la piattaforma sopraelevata modellata come corpo rigido.

6.2 Considerazioni sull'Approccio PDDLStream

L'esperienza maturata nel presente lavoro consente alcune riflessioni sull'approccio PDDLStream rispetto ad alternative quali PDDL 2.1 e PDDL+.

Vantaggi di PDDLStream:

- La generazione on-demand di valori continui evita l'esplosione combinatoria tipica della discretizzazione a priori
- L'incapsulamento delle procedure geometriche in stream garantisce modularità e riusabilità
- Il meccanismo dei fluents consente agli stream di accedere allo stato corrente, abilitando strategie di motion planning informate

Limitazioni osservate:

- Il debugging può risultare complesso a causa dell'interazione tra livello simbolico e procedurale
- Le prestazioni sono sensibili alla configurazione degli stream (strategie di deferimento, ordine di valutazione)

6.2.1 Confronto con PDDL+ e PDDL 2.1:

PDDL+ sarebbe teoricamente applicabile al problema considerato, modellando il movimento dei robot come processi continui con vincoli geometrici. Tuttavia, la praticabilità dipende criticamente da come sono gestite le pose e le traiettorie.

Scenario 1: Pose Predefinite su Griglia (n_blocchi = 2)

Se le pose iniziali e finali fossero vincolate a una griglia predefinita (es. 3×3 posizioni), allora PDDL+ diventerebbe fattibile. Le traiettorie potrebbero essere pre-calcolate offline durante l'inizializzazione del problema, generando waypoint discreti per ogni coppia di pose. I vincoli di collision checking verrebbero codificati nelle precondizioni "over all" come disuguaglianze lineari sui waypoint. Questo approccio manterrebbe la natura continua della pianificazione pur riducendo l'esplosione combinatoria. L'applicabilità è garantita per entrambi i problemi, se riconfigurati con pose discrete.

Scenario 2: Aumento di Complessità (n_blocchi = 3)

All'aumentare del numero di blocchi (e.g., da 2 a 3), la sostenibilità di PDDL+ comincia a degradarsi anche con pose predefinite. Con tre blocchi, il numero di verifiche di collision aumenta proporzionalmente: ogni waypoint deve essere verificato contro 3 blocchi statici anziché 2. Inoltre, le combinazioni di configurazioni dei blocchi sul tavolo crescono esponenzialmente (3! = 6 permutazioni), richiedendo potenzialmente azioni separate per ogni

configurazione. Oltre 4-5 blocchi, la codifica PDDL+ diventa intrattabile per i planner numerici attuali, anche con pose predefinite. PDDLStream, d'altra parte, scala maggiormente perché il collision checking rimane delegato a funzioni geometriche ottimizzate, indipendentemente dal numero di blocchi.

Scenario 3: Generazione Stocastica (Attuale)

Lo scenario attuale prevede una generazione stocastica di pose, e PDDL+ risulta immediatamente impraticabile perché le pose non sono predeterminate e le traiettorie variano dinamicamente tramite RRT. Ogni esecuzione del planificatore potrebbe generare nuove pose, rendendo impossibile pre-codificare i vincoli geometrici nel PDDL. PDDLStream gestisce bene questo caso delegando il collision checking a funzioni esterne (algoritmo RRT con discretizzazione locale a 0.05 unità), mantenendo il PDDL semplice e il vero problem-solving nel layer geometrico.

Infine, PDDL 2.1 non porterebbe alcun vantaggio significativo: le azioni durative, sua caratteristica principale, sono superflue quando il tempo non è una risorsa esplicita da pianificare, ma emerge implicitamente dai costi delle traiettorie.