Autonomous and Mobile Robotics

Prof. Giuseppe Oriolo

Introduction to V-REP with Applications to Motion Planning

Michele Cipriano

Dipartimento di Ingegneria Informatica Automatica e Gestionale Antonio Ruberti

Sapienza Università di Roma

outline

- introduction to V-REP
  - basic elements
  - dynamic modeling
  - C++ plugins
  - Matlab/Simulink interface

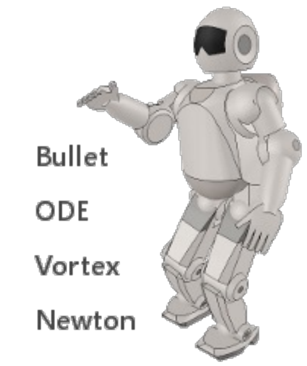
- applications to task-constrained motion planning
  - with moving obstacles
  - in the presence of soft task constraints
  - for humanoid robots

the V-REP simulator

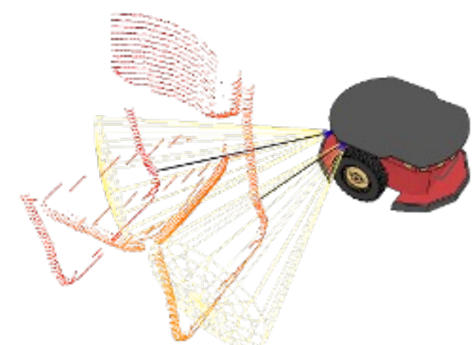
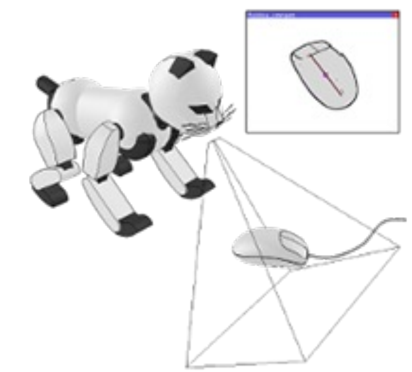
- V-REP = Virtual Robot Experimentation Platform
- a robotic simulator: a software environment aimed at generic robotic applications (not only motion planning)
- relatively new (2014), produced by Coppelia Robotics
- free and open source
- available on Windows, Linux and Mac
- example of applications
  - fast prototyping and verification
  - fast algorithm development
  - hardware control
  - etc

the V-REP simulator

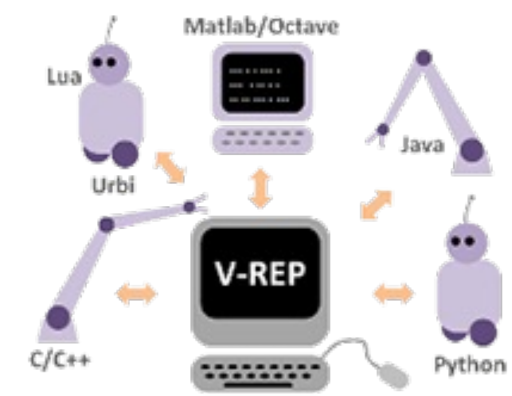
- provides physical engines for dynamic simulations



- allows the simulation of sensors

- its functionalities can be easily extended using many programming languages (C/C++, Python, Java, Lua, MATLAB, Octave, Urbi) and programming approaches (remote clients, plugins, ROS nodes,…)

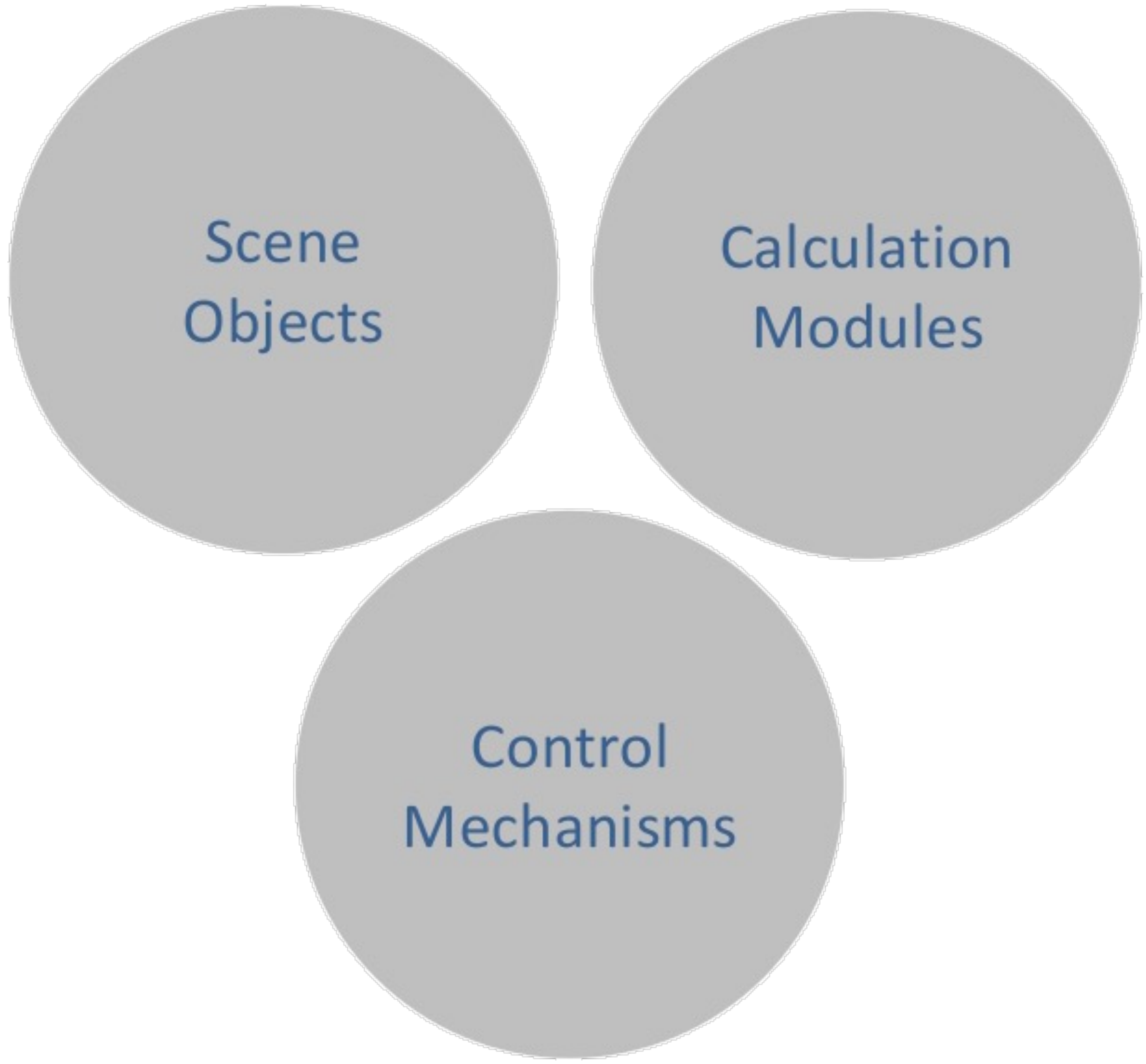


- provides a large and continuously growing library of robot models




three central elements

how to build
a robot

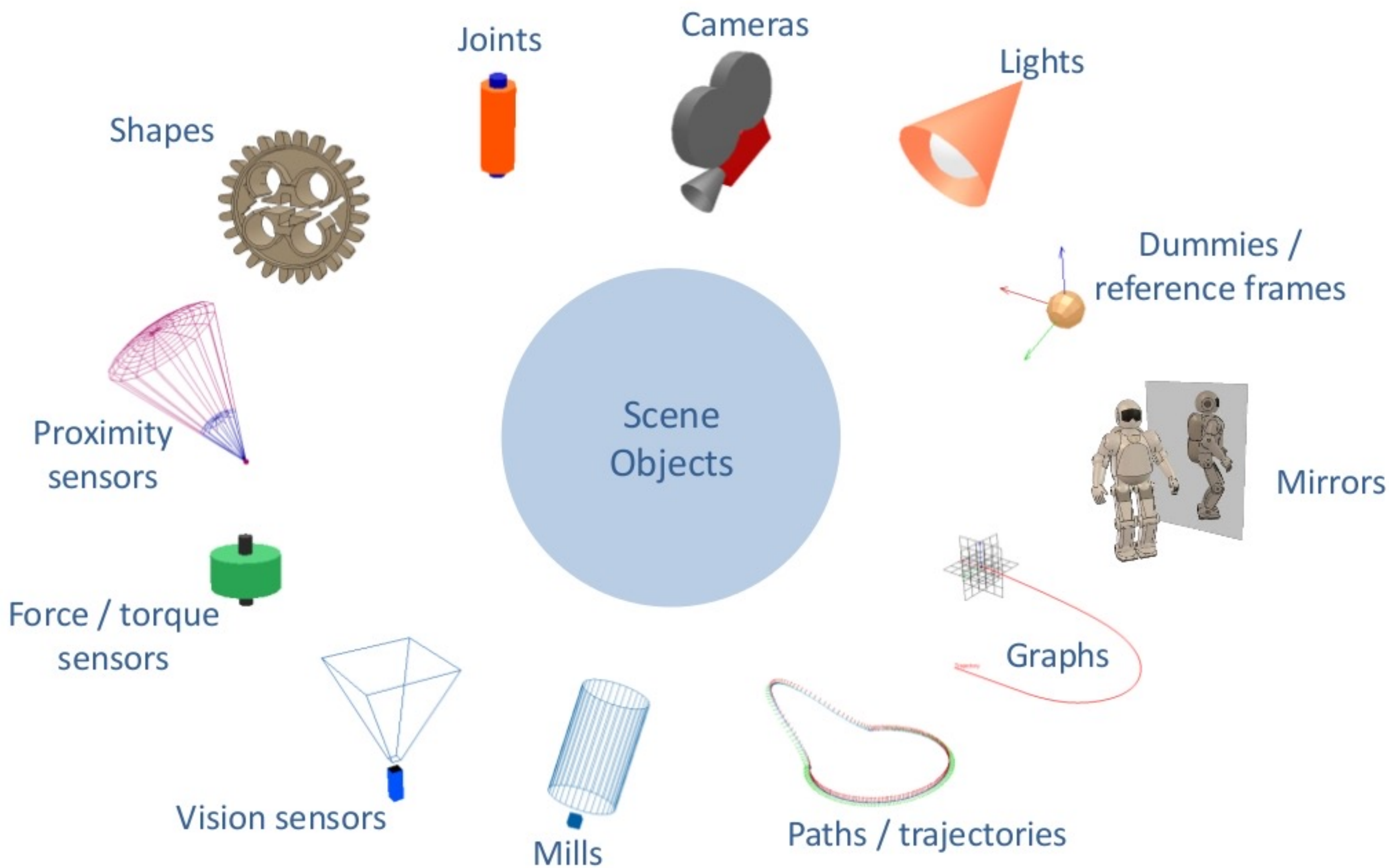


how to simulate
a robot

how to control
a robot

scene objects

- how to build a robot

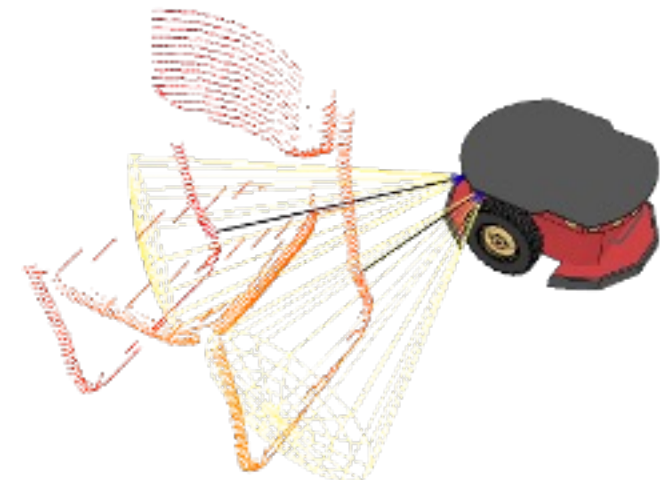


scene objects: basic components

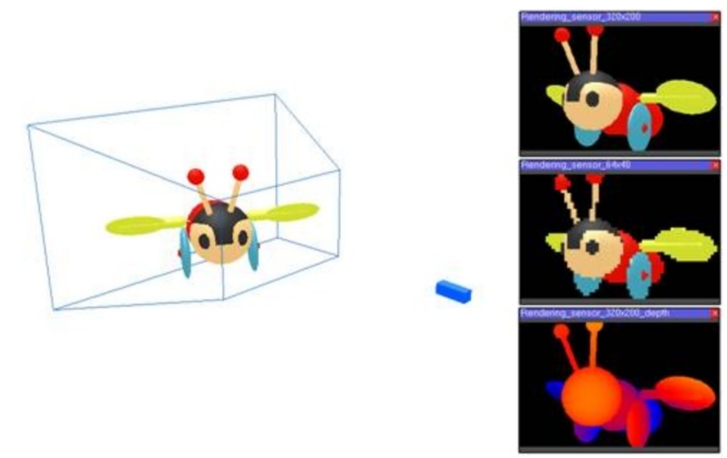
- shapes
  - rigid mesh objects that are composed of triangular faces
  - can be grouped/ungrouped
  - different types (random, convex, pure shapes)
- joints
  - revolute: rotational movement
  - prismatic: translational movement
  - screw: translational while rotational movement
  - spherical: three rotational movements
- a robot model can be created through a hierarchical structure including (at least) shapes and joints

scene objects: sensors

- proximity sensors
  - more than simple ray-type detection
  - configurable detection volume
  - fast minimum distance calculation within volume

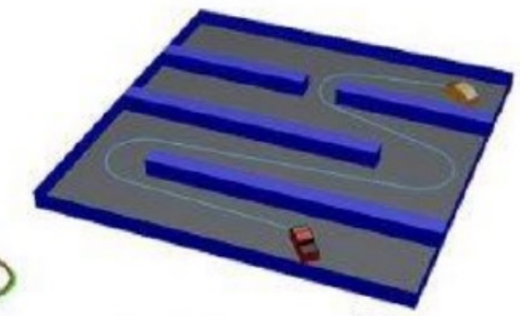


- vision sensors
  - render the objects that are in their field of view
  - embedded image processing
  - two different types: orthographic projection-type (e.g., close-range infrared or laser range finders) and perspective projection-type (e.g., camera-like sensors)

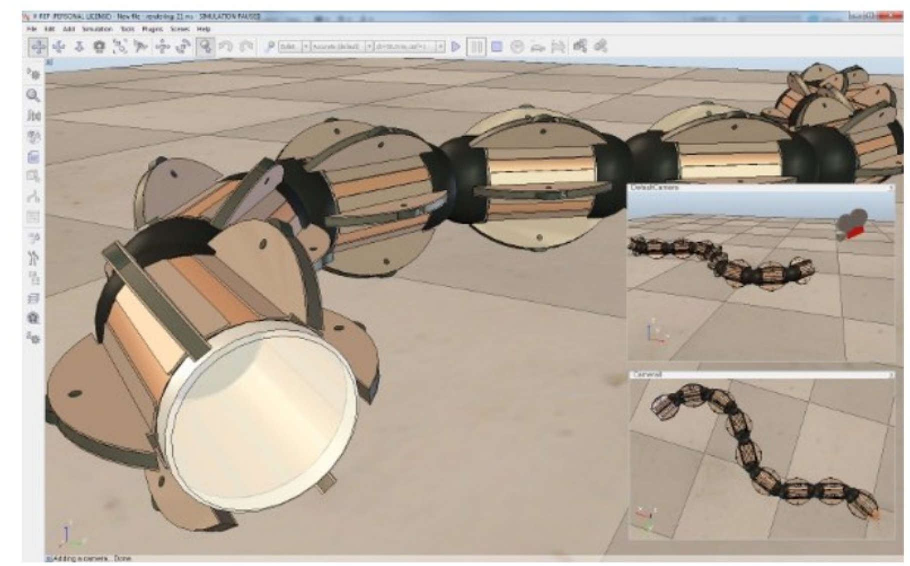


- torque/force sensors
  - measure applied force/torque (on 3 principal axes)

scene objects: other components



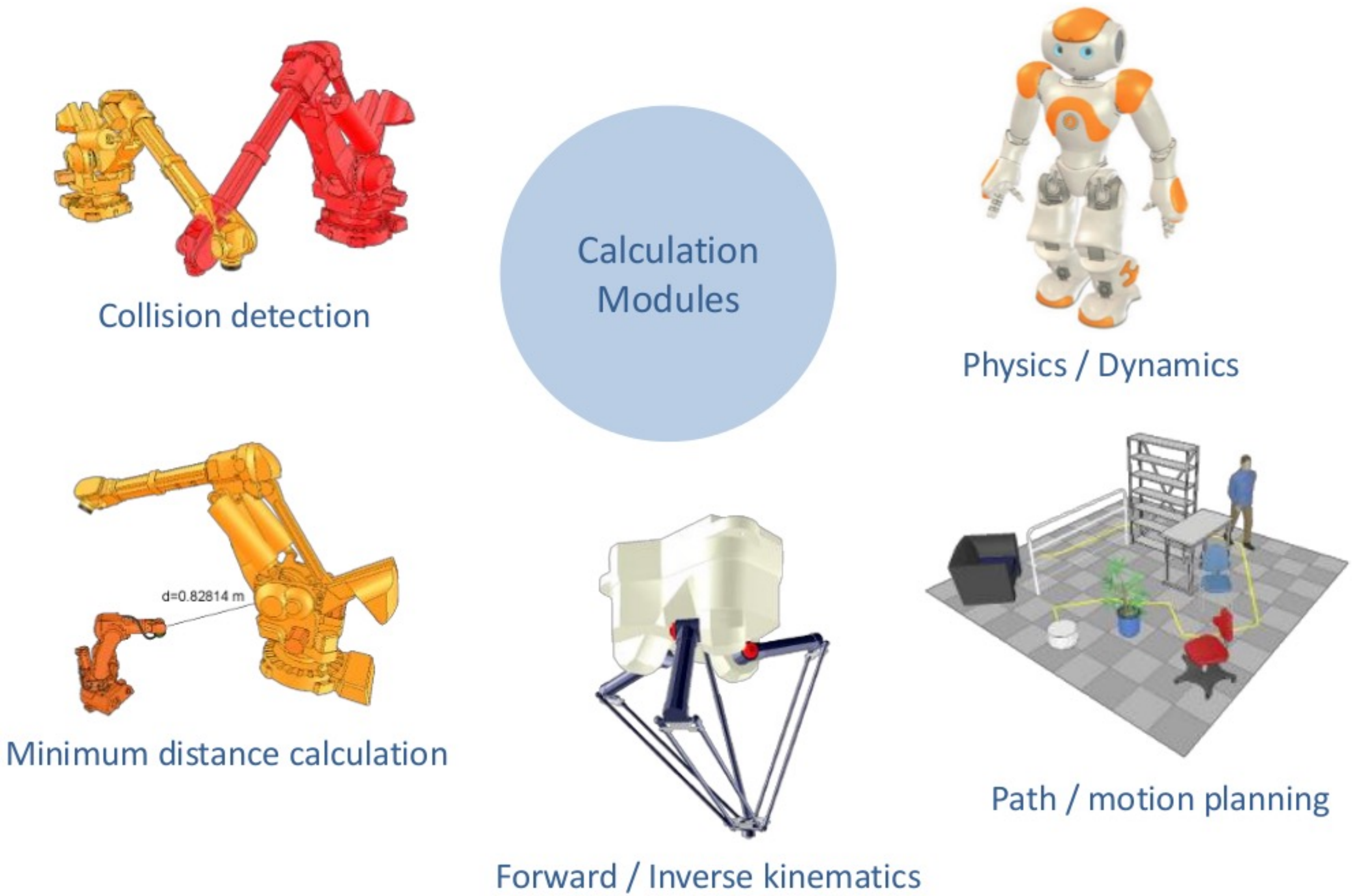
- paths/trajectories
  - allow 6D definitions
  - can be easily created
    (by importing a file or defining control points)



- cameras
  - perspective/orthographic projection
  - can track an object while moving
- lights: omnidirectional, spotlight, directional
- mirrors: reflect images/light, auxiliary clipping frame
- graphs: draw 3D curves, easily exportable
- dummies: auxiliary reference frame

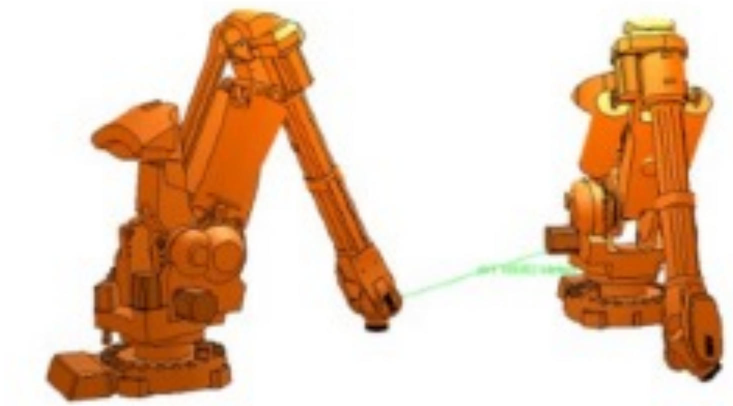
calculation modules

- how to simulate a robot



calculation modules

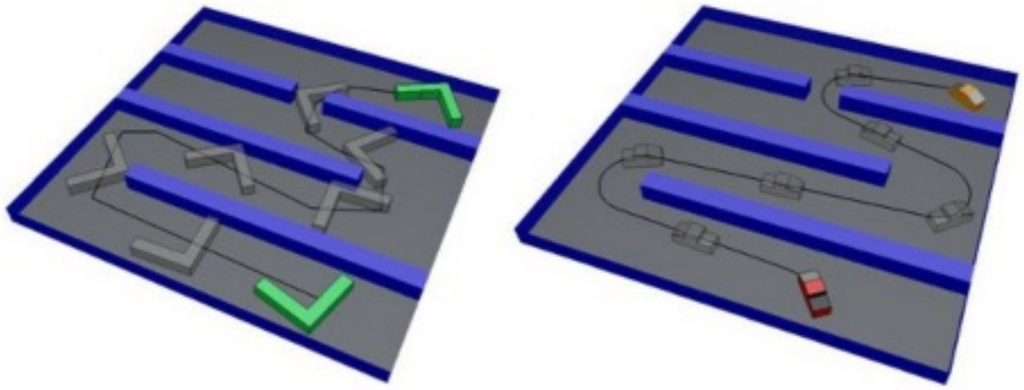
- forward/inverse kinematics
  - can be used for any kinematic chain (closed, redundant, …)
  - build inverse kinematic (IK) group
  - different techniques for inverse kinematics (pseudoinverse, damped least square)
  - accounts for joint limits and obstacle avoidance
- minimum distance computation
  - can be used between any pair of meshes
  - very fast and optimized



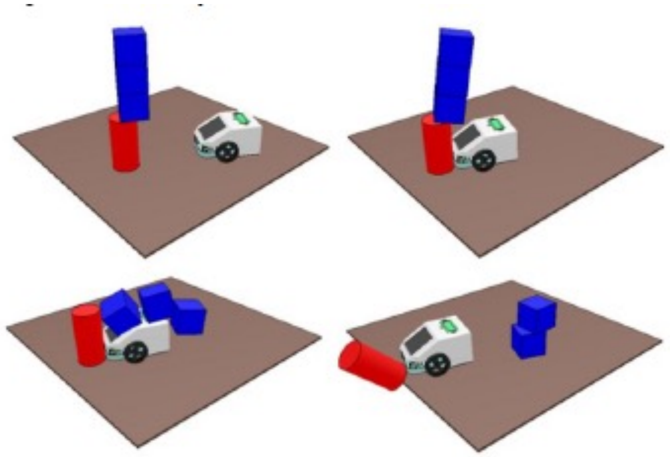
- collision detection
  - can be used between any pair of meshes
  - scene objects can be defined as collidable or not

calculation modules

- path/motion planning
  - can be performed for any kinematic chain
  - holonomic/non holonomic path planning
  - requires the specification of: start/goal configuration, obstacles, robot model
  - uses the OMPL library

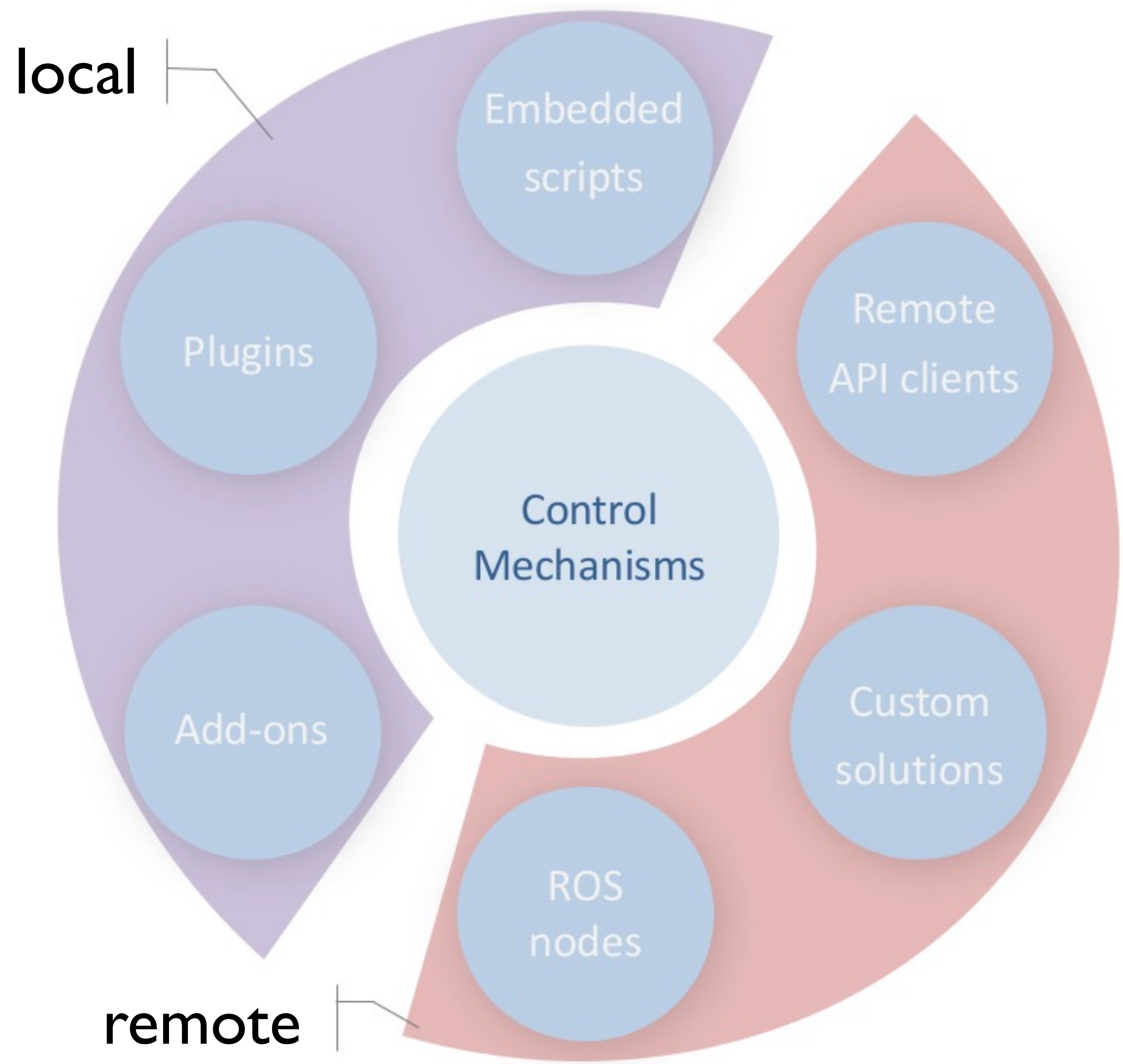


- physics/dynamics
  - enable dynamic simulations (gravity, friction, …)
  - four different physical engines: Bullet, ODE, Vortex, Newton (ordered by computational demand)
  - dynamic particles to simulate air or water jets



control mechanisms

- how to **control** a robot

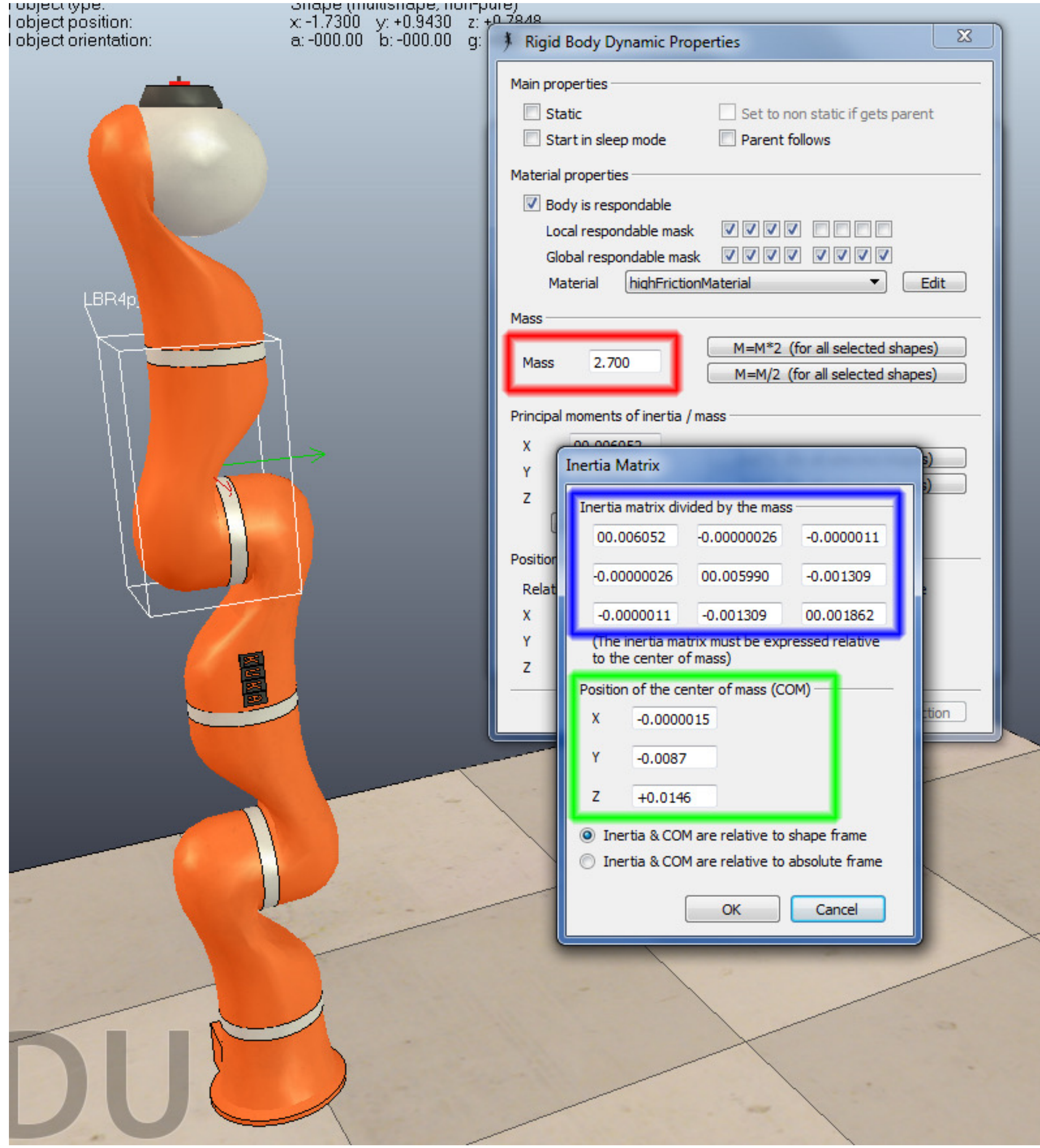


control mechanisms

- local approach: control entity is internal
  - embedded script: associated to a single robot
  - add-on: can only execute minimalistic code
  - plugin: most general tool, fast computation, written in C++
- remote approach: control entity is external
  - ROS node: bridge between V-REP and ROS
  - custom solution: client/server paradigm using the BlueZero framework
  - remote API client: communication between VREP and an external application (e.g., Matlab/Simulink)

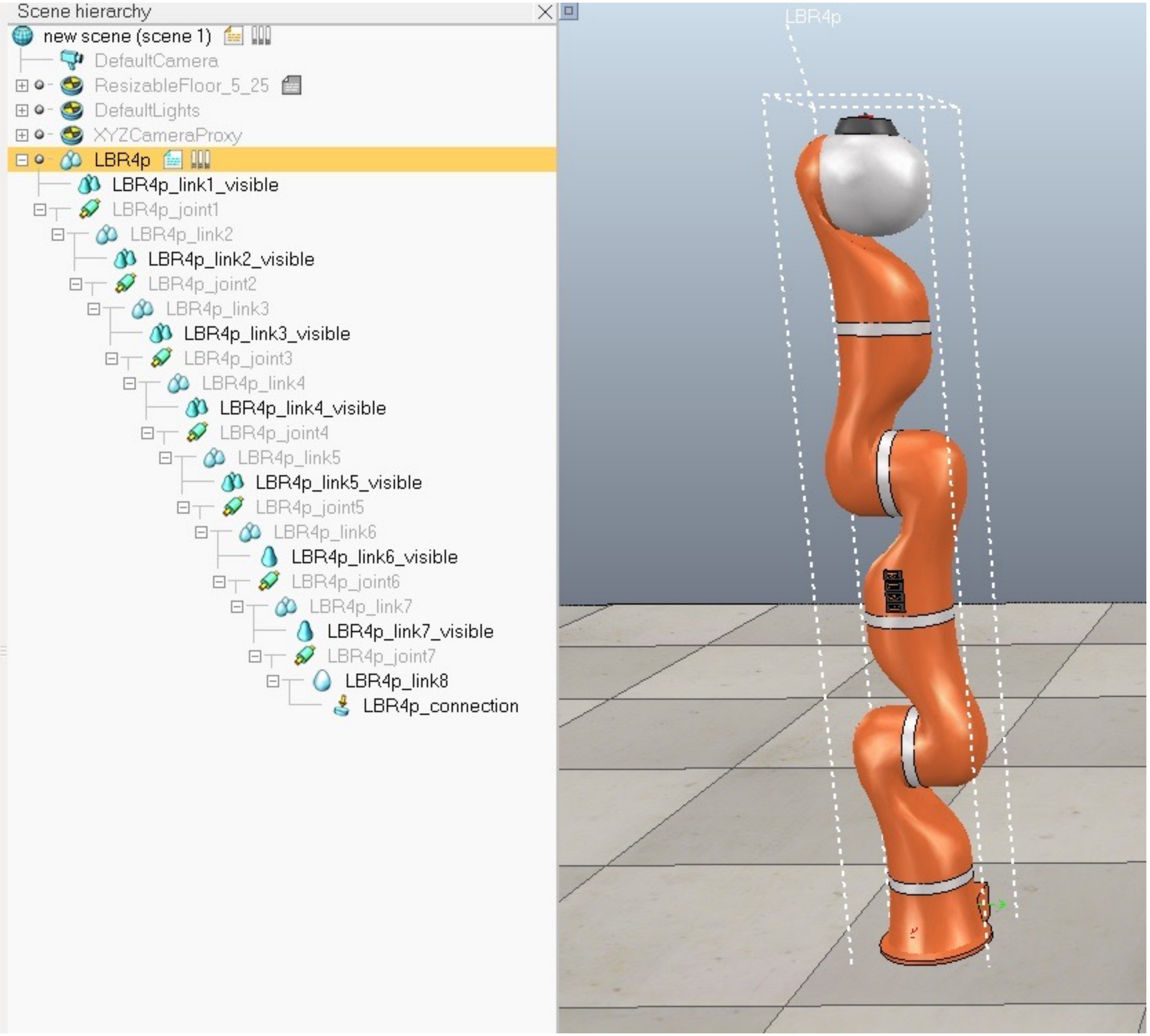
dynamic modeling of a robot

- building a dynamic model in V-REP is very easy: no equations are needed
- it requires a few simple steps:
  1. import a CAD model of the robot
  2. associate to each body of the robot its dynamic parameters: mass, center of mass, inertia matrix



dynamic modeling of a robot

3. build a model tree: a tree that represents all hierarchical information of the kinematic chains (links and joints)



C++ plugins

- uses the V-REP regular APIs (more than 450 functions available)
- produces a shared library (e.g., .so for Linux and .dll for Windows)
- automatically loaded by V-REP at program start-up
- can be integrated with other C++ libraries (e.g., Eigen, Octomap, etc)
- two main applications
  - extend V-REP's functionality through user-written functions (e.g., motion planning algorithms, controllers, …)
  - used as a wrapper for running code written in other languages
- a single plugin can manage more than one robot
- fast execution (particularly suited for motion planning)

C++ plugins

- at each event in the V-REP interface, a corresponding message is sent to the plugin
- each message triggers the execution of a particular portion of the code in the plugin

before starting the simulation ("offline" functions)

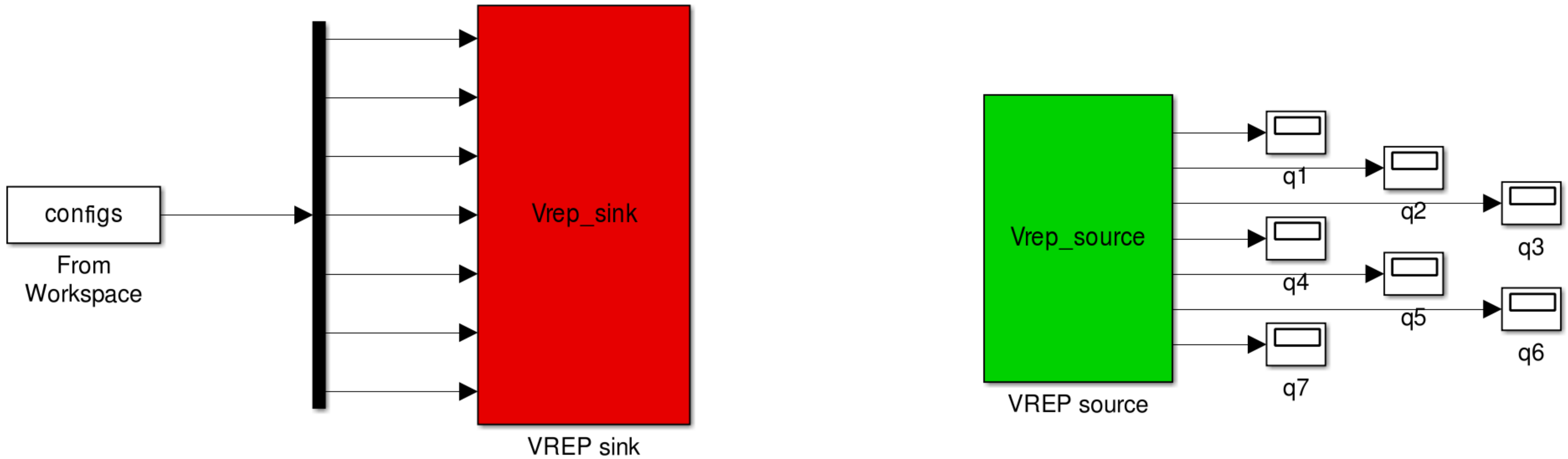
simulation is stopped

```
if (message == sim_message_eventcallback_simulationabouttostart) { // Simulation is about to start
    Planning();
}
if (message == sim_message_eventcallback_simulationended) { // Simulation just ended
    std::cout << "Simulation Concluded" << std::endl;
}
if (message == sim_message_eventcallback_modulehandle) { // Simulation is running
    Execution();
}
```

during the simulation ("online" functions)

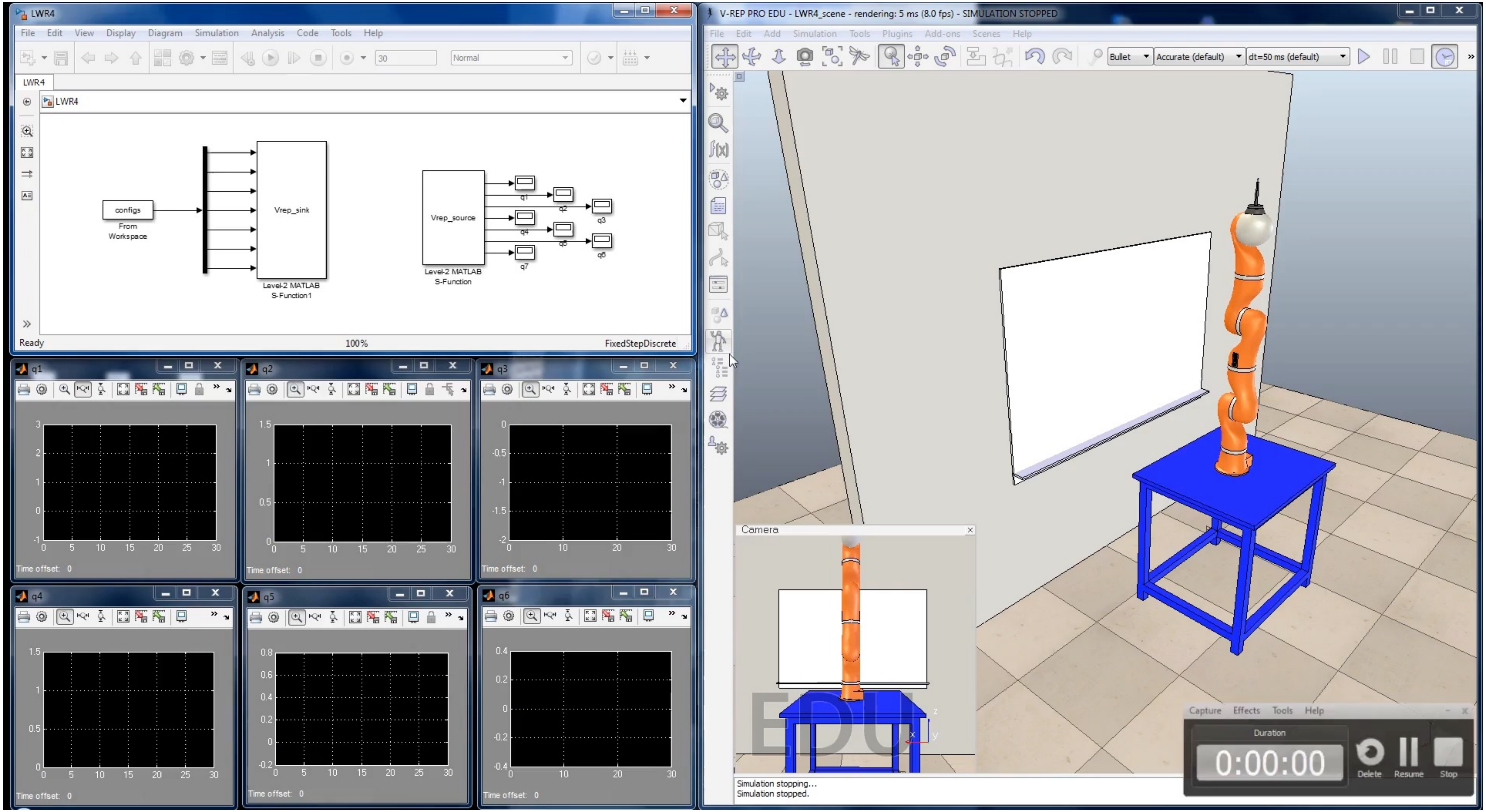
Matlab/Simulink interface

- uses the V-REP remote APIs (more than 100 functions available)
- an interface for sending/receiving commands to/from V-REP
- two main blocks: V-REP sink and V-REP source respectively sends/reads values from V-REP joints



- V-REP and Matlab/Simulink times automatically synchronized
- Matlab/Simulink commands start/stop V-REP simulations

Matlab/Simulink interface

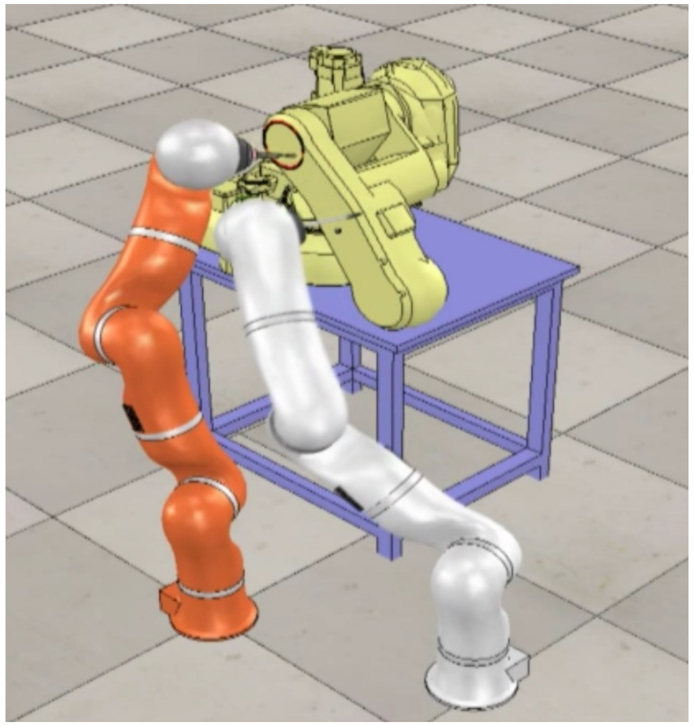


wrap up

- V-REP main advantages
  - very good online documentation
  - four different physics engines
  - most complete open source software for dynamic simulations
  - very good set of APIs and control mechanisms
  - very fast software development when one gets the V-REP structure
- V-REP main drawbacks
  - vision sensors have high computational payload
  - since it is a huge software, it is not so friendly at the beginning
  - multi-robot simulations have high computational payload
  - collision checking library is slow

task-constrained motion planning with moving obstacles (TCMP-MO)

- problem: find feasible, collision-free motions for a robot that is assigned a task constraint in an environment containing moving obstacles
- the robot configuration $\boldsymbol{q}$ takes values in a n_q-dimensional configuration space
- the task is defined by a desired task path $\boldsymbol{y}_d(s), s \in [0,1]$, that takes values in an n_y-dimensional task space
- assumptions
  - the robot is kinematically redundant w.r.t. the task ($n_q > n_y$)
  - the obstacle trajectories are known



TCMP-MO

- in the kinematic case

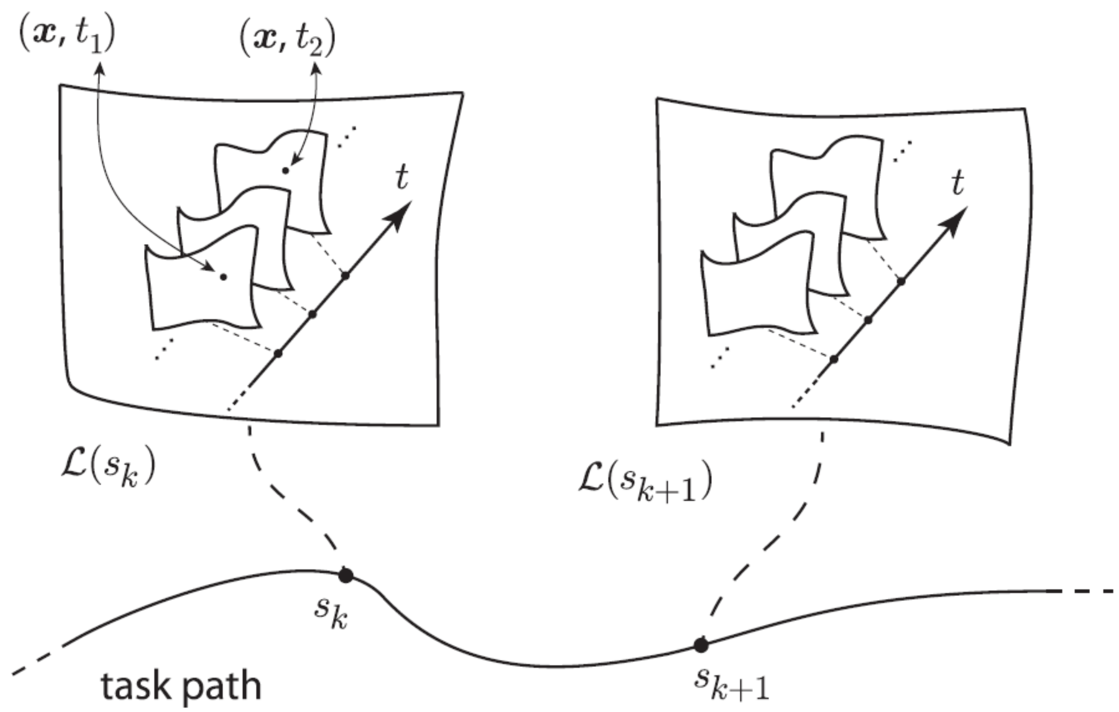
  robot described by the kinematic-level model $\dot{\boldsymbol{q}} = \boldsymbol{v}$
  with generalized velocities $\dot{\boldsymbol{q}} = \dot{s}\boldsymbol{q}'$, and state $\boldsymbol{x} = \boldsymbol{q}$

- in the dynamic case

  robot described in the Euler-Lagrange form $\boldsymbol{B}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{n}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \boldsymbol{\tau}$
  with generalized accelerations $\ddot{\boldsymbol{q}} = \dot{s}^2\boldsymbol{q}'' + \ddot{s}\boldsymbol{q}'$, and state $\boldsymbol{x} = (\boldsymbol{q}, \dot{\boldsymbol{q}})$

- solution to the TCMP-MO problem: a configuration-space trajectory $\boldsymbol{q}(t)$, i.e., a path $\boldsymbol{q}(s)$ + a time history $s(t)$, such that:
  - the assigned task path is continuously satisfied
  - collisions are always avoided
  - joint velocity limits (kinematic case) or joint velocity/torque limits (dynamic case) are respected

search space



- in general, a state may be admissible at a certain time instant and not admissible at another due to the movements of the obstacles
- task-constrained state-time space (STS) $\mathcal{S}_{\text{task}}$ decomposes as a foliation, i.e., $\mathcal{S}_{\text{task}} = \bigcup_{s\in[0,1]} \mathcal{L}(s)$ with $\mathcal{L}(s)$ the generic leaf
  – in the kinematic case

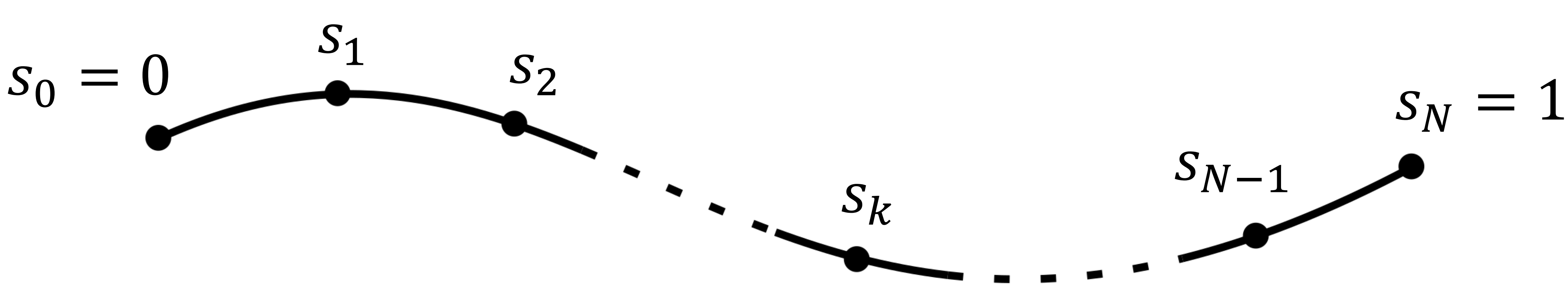
$$\mathcal{L}(s) = \{(\boldsymbol{q}, t) \in STS: \boldsymbol{f}(\boldsymbol{q}) = \boldsymbol{y}_d(s)\}$$

  – in the dynamic case

$$\mathcal{L}(s) = \{(\boldsymbol{q}, \dot{\boldsymbol{q}}, t) \in STS: \boldsymbol{f}(\boldsymbol{q}) = \boldsymbol{y}_d(s), \boldsymbol{J}(\boldsymbol{q})\dot{\boldsymbol{q}} = \dot{s}\boldsymbol{y}_d'(s)\}$$

approach

- an offline randomized algorithm
- builds a tree, similarly to the RRT, in $\mathcal{S}_{\text{task}} \cap \mathcal{S}_{\text{free}}$
  - a vertex contains a state of the robot and an associated time instant
  - an edge between two adjacent vertexes represents a feasible collision-free subtrajectory in the configuration space
- makes use of $N+1$ samples of the desired task path $\boldsymbol{y}_d(s)$, corresponding to the equispaced sequence $\{s_0 = 0, s_1, \dots, s_N = 1\}$



the kinematic case

- root the tree at $(\boldsymbol{q}_{\text{ini}}, 0)$
- iteratively
  - randomly select a sample $\boldsymbol{y}_{\text{rand}}$
  - compute an IK solution $\boldsymbol{q}_{\text{rand}} = \boldsymbol{f}^{-1}(\boldsymbol{y}_{\text{rand}})$
  - randomly assign to $\boldsymbol{q}_{\text{rand}}$ a time instant t_{rand}
  - search the closest vertex $(\boldsymbol{q}_{\text{near}}, t_{\text{near}})$ to $(\boldsymbol{q}_{\text{rand}}, t_{\text{rand}})$, and extract s_k associated to $\boldsymbol{q}_{\text{near}}$
  - generate forward/backward motions
    - time histories: randomly choose two values of $\dot{s}$ (pos/neg)
    - subpaths: numerically integrate using random $\widetilde{\boldsymbol{w}}$

$$\boldsymbol{q}' = \boldsymbol{J}^{\#}\left(\pm\boldsymbol{y}_d' + k_p\boldsymbol{e}_y\right) + (\boldsymbol{I} - \boldsymbol{J}^{\#}\boldsymbol{J})\widetilde{\boldsymbol{w}}$$

  - if no violation occurs, add new vertices and edges to the tree

the dynamic case

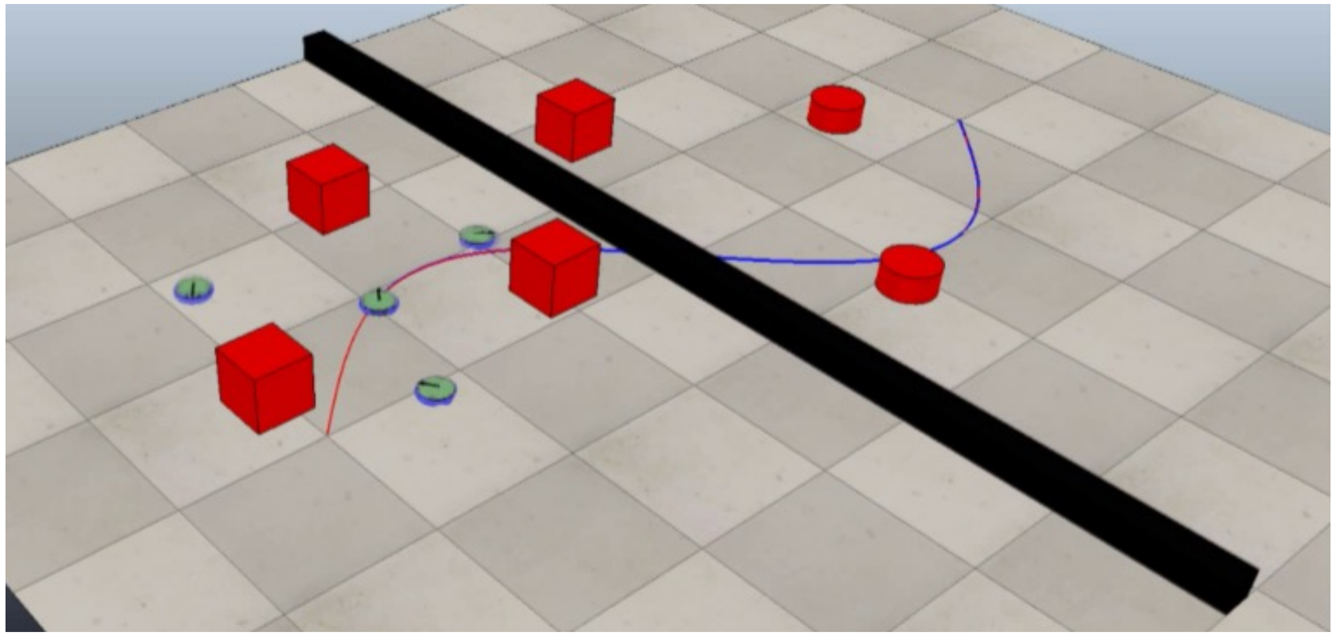
- root the tree at $(\boldsymbol{q}_{\text{ini}}, \dot{\boldsymbol{q}}_{\text{ini}}, 0)$
- iteratively
  - randomly select a sample $\boldsymbol{y}_{\text{rand}}$
  - compute an IK solution $\boldsymbol{q}_{\text{rand}} = \boldsymbol{f}^{-1}(\boldsymbol{y}_{\text{rand}})$
  - randomly assign to $\boldsymbol{q}_{\text{rand}}$ a time instant t_{rand} and a velocity $\dot{\boldsymbol{q}}_{\text{rand}}$
  - search the closest vertex $(\boldsymbol{q}_{\text{near}}, \dot{\boldsymbol{q}}_{\text{near}}, t_{\text{near}})$ to $(\boldsymbol{q}_{\text{rand}}, \dot{\boldsymbol{q}}_{\text{rand}}, t_{\text{rand}})$, and extract s_k associated to $\boldsymbol{q}_{\text{near}}$
  - generate accelerating/decelerating motions
    - time histories: randomly choose two values of $\ddot{s}$ (pos/neg)
    - subpaths: numerically integrate using random $\tilde{\boldsymbol{z}}$
      $$\boldsymbol{q}'' = \boldsymbol{J}^{\#}\left(\boldsymbol{y}_d'' - \boldsymbol{J}'\boldsymbol{q}' + k_p \boldsymbol{e}_y + k_d \boldsymbol{e}_y'\right) + (\boldsymbol{I} - \boldsymbol{J}^{\#}\boldsymbol{J})\tilde{\boldsymbol{z}}$$
  - if no violation occurs, add new vertices and edges to the tree

exploration-exploitation

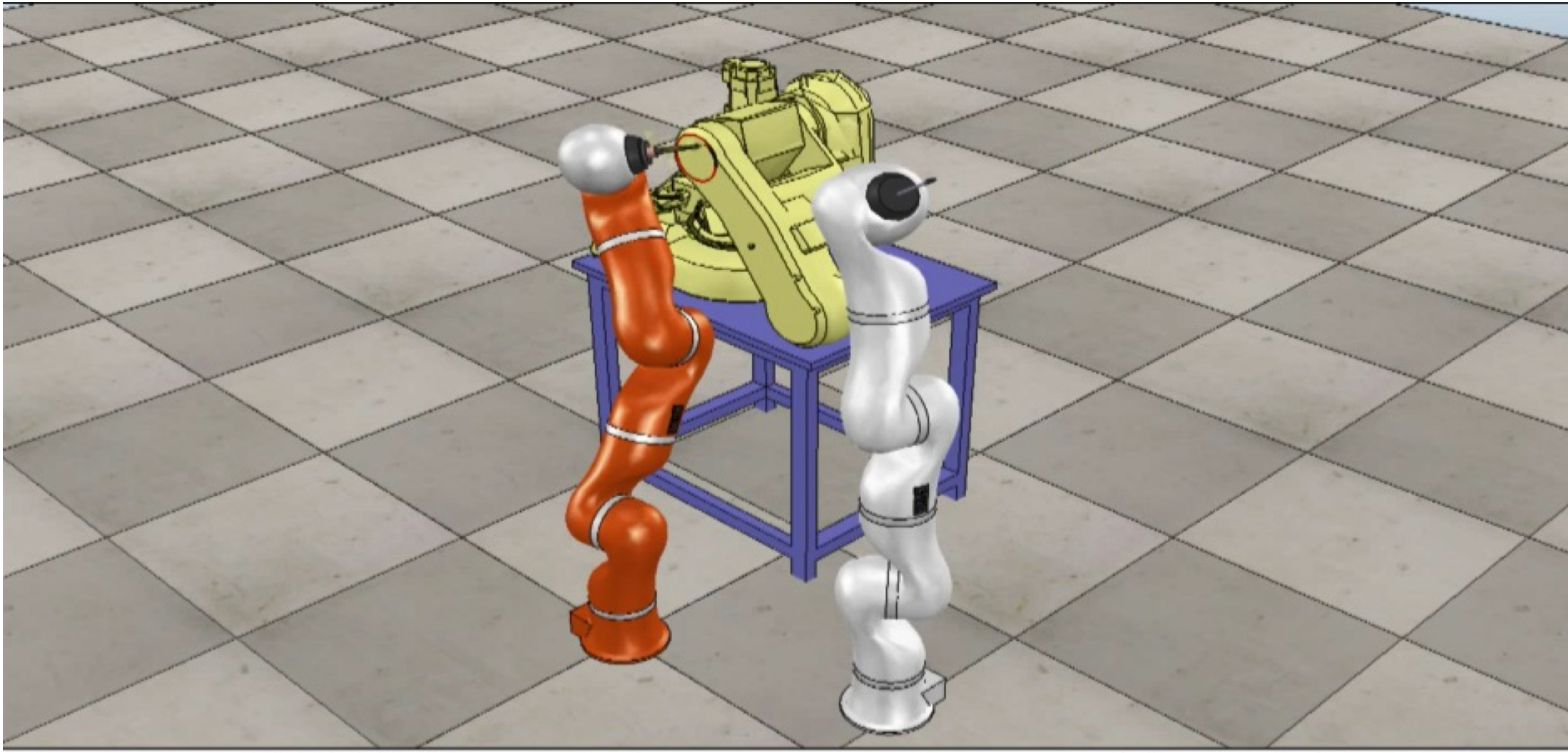
improve the search efficiency and/or the quality of the solution by alterating

- exploration
  - residual input ($\tilde{\boldsymbol{w}}$ or $\tilde{\boldsymbol{z}}$) chosen randomly
  - essential for guaranteeing probabilistic completeness
- exploitation
  - residual input ($\tilde{\boldsymbol{w}}$ or $\tilde{\boldsymbol{z}}$) chosen deterministically with the objective of minimizing a certain state-dependent cost function

example: minimize the centroidal variance of a formation of mobile robots



V-REP simulations



the orange KUKA LWR manipulator must perform a welding task (red circle) while avoiding another LWR (white) moving in its workspace

soft task-constrained motion planning (STCMP)

- problem: find feasible, collision-free motions for a robot that is assigned a soft task constraint in an environment containing obstacles

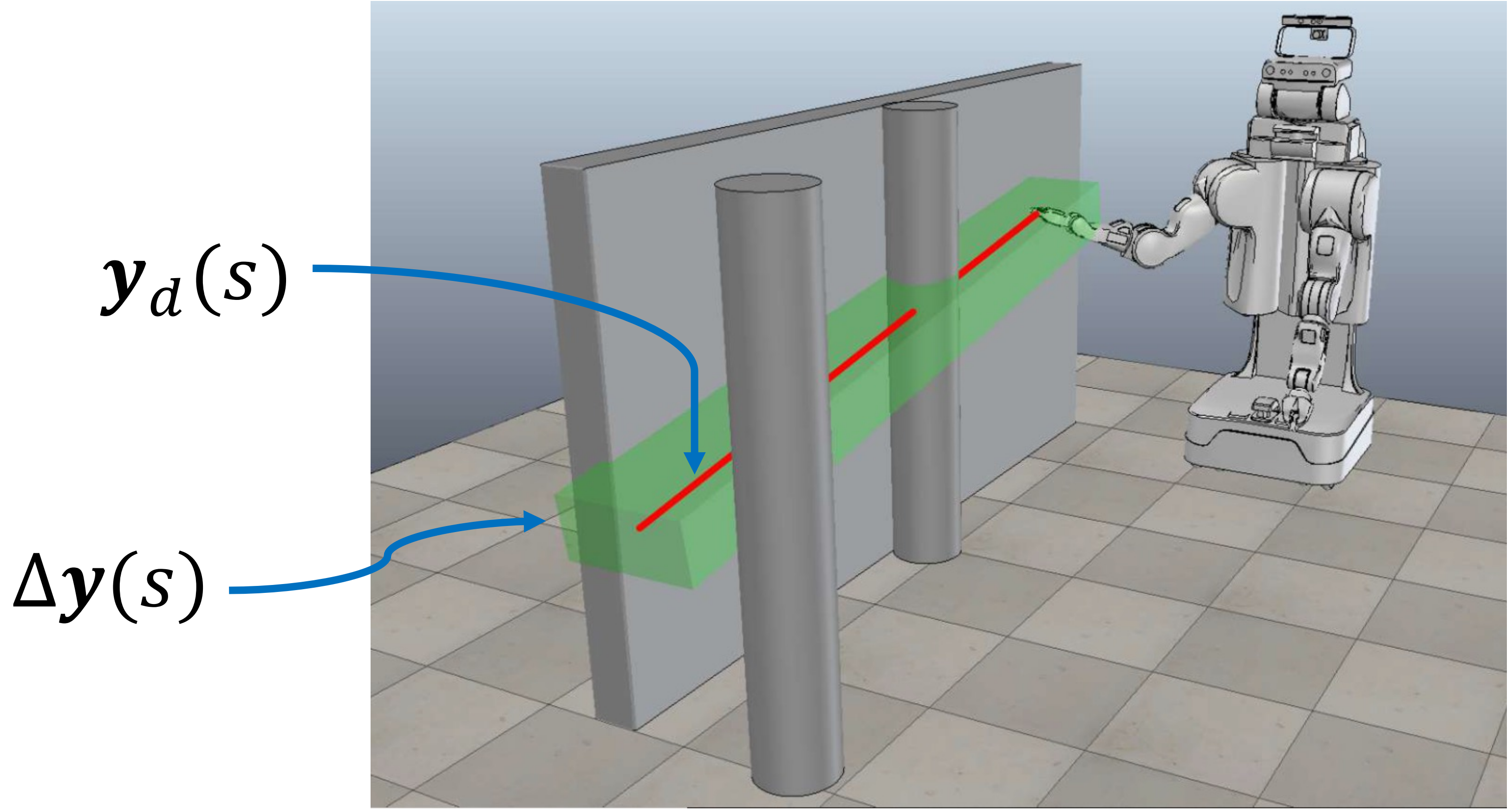
- the robot configuration $\boldsymbol{q}$ takes values in a n_q-dimensional configuration space $\mathcal{C}$, containing a free part $\mathcal{C}_{\text{free}}$

- the task is described in coordinates $\boldsymbol{y}$ taking values in a n_y-dimensional task space $\mathcal{Y}$

- assumptions
  - the robot is kinematically redundant w.r.t. the task ($n_q > n_y$)
  - obstacles are fixed
  - the robot is free-flying in $\mathcal{C}$, then its kinematic model consists of simple integrators and is expressed in geometric form as $\boldsymbol{q}' = \tilde{\boldsymbol{v}}$

soft task constraints

- the soft task constraint is defined by
  - a desired task path $\boldsymbol{y}_d(s), s \in [0,1]$
  - a tolerance $\Delta\boldsymbol{y}(s), s \in [0,1]$, a positive n_y-vector that represents for each component the max admissible deviation of $\boldsymbol{y}$ from $\boldsymbol{y}_d$ at s



- the tolerance can be exploited whenever realizing the desired task exactly is difficult or impossible (narrow or closed passages in $\mathcal{C}$)

STCMP

- the task error associated to $\boldsymbol{q}$ at s is $\boldsymbol{e}(\boldsymbol{q}, s) = \boldsymbol{y}_d(s) - \boldsymbol{f}(\boldsymbol{q}(s))$
- $\boldsymbol{q}$ is compliant with the hard task at s if $\boldsymbol{e}(\boldsymbol{q}, s) = 0$

$$\mathcal{L}(s) = \{\boldsymbol{q} \in \mathcal{C}: \boldsymbol{e}(\boldsymbol{q}, s) = 0\} \Longrightarrow \mathcal{C}_{\text{hard}} = \bigcup_{s \in [0,1]} \mathcal{L}(s)$$

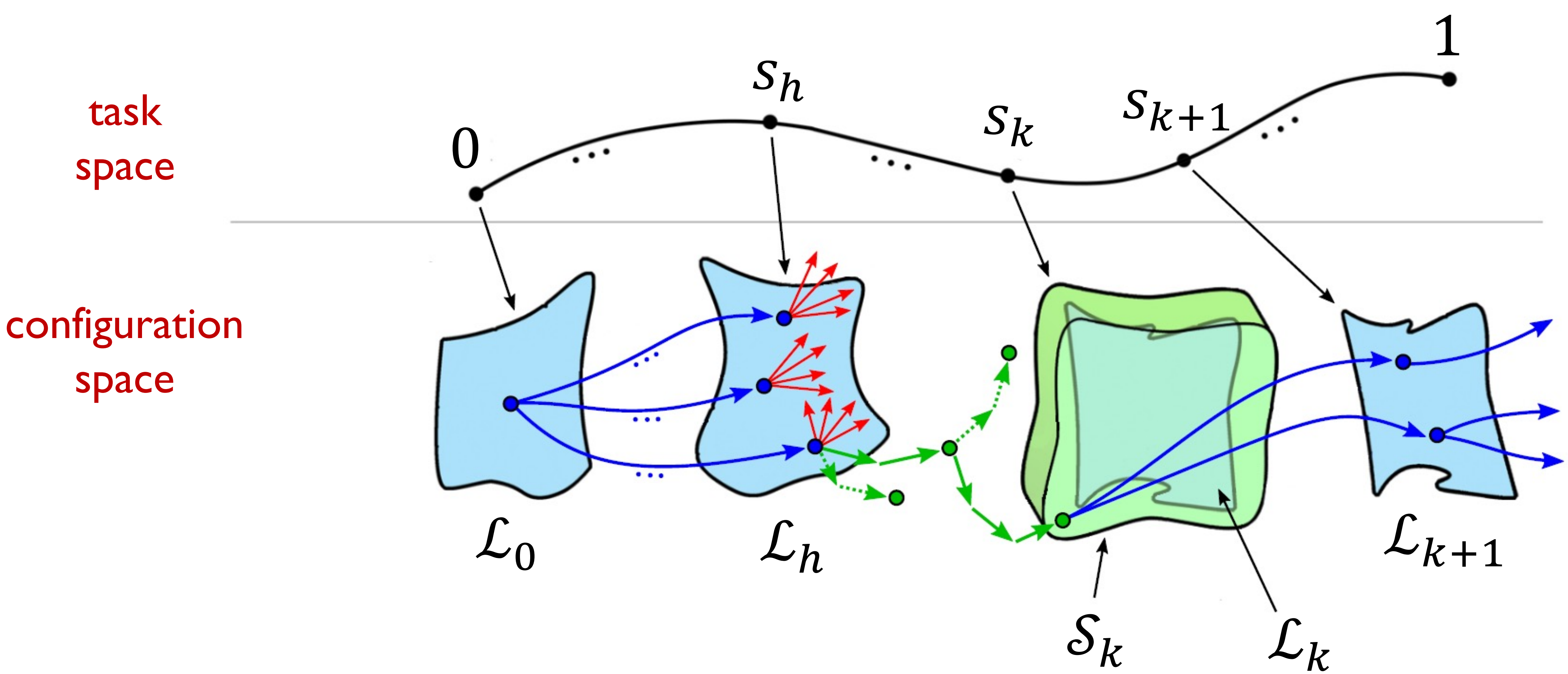
- $\boldsymbol{q}$ is compliant with the soft task at s if $|\boldsymbol{e}(\boldsymbol{q}, s)| \leq \Delta\boldsymbol{y}(s)$

$$\mathcal{S}(s) = \{\boldsymbol{q} \in \mathcal{C}: \boldsymbol{e}(\boldsymbol{q}, s) \leq \Delta\boldsymbol{y}(s)\} \Longrightarrow \mathcal{C}_{\text{soft}} = \bigcup_{s \in [0,1]} \mathcal{S}(s)$$

- solution: a configuration-space path $\boldsymbol{q}(s), s \in [0,1]$, such that for all s:
  - $\boldsymbol{q}(s)$ is compliant with the soft task
  - (self-)collisions, singularities and joint limits violations are avoided
- but: the solution should comply with the hard task as much as possible

opportunistic planner

- builds a tree $\mathcal{T}$ in $\mathcal{C}_{\text{soft}} \cap \mathcal{C}_{\text{free}}$ alternating two (sub)planners
  - the hard planner (HP) extends $\mathcal{T}$ as much as possible in $\mathcal{C}_{\text{hard}}$
  - when HP identifies an obstruction, the soft planner (SP) extends $\mathcal{T}$ in $\mathcal{C}_{\text{soft}}$ until extension by HP is viable again



- frontier index h: index of the largest sample of s for which there exists a vertex $\boldsymbol{q}$ in $\mathcal{T}$ on $\mathcal{L}_h$ or $\mathcal{S}_h$

hard planner

generic iteration

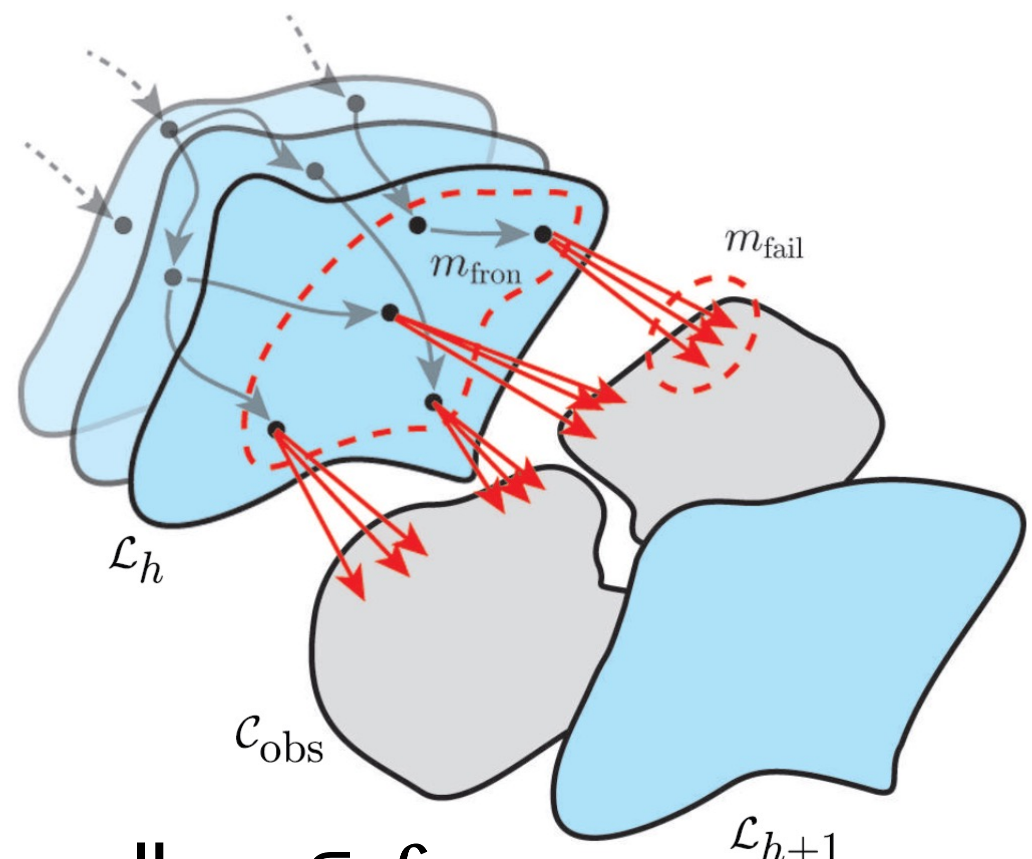
1. generate a random configuration $\boldsymbol{q}_{\text{rand}}$ in $\mathcal{C}_{\text{hard}}$
2. select in $\mathcal{T}$ the closest vertex $\boldsymbol{q}_{\text{near}}$ to $\boldsymbol{q}_{\text{rand}}$
3. produce a subpath from $\boldsymbol{q}_{\text{near}} \in \mathcal{L}_j$ to $\boldsymbol{q}_{\text{new}} \in \mathcal{L}_{j+1}$ by integrating

$$\tilde{\boldsymbol{v}} = \boldsymbol{J}^{\#}(\boldsymbol{q})\big(\boldsymbol{y}_d' + \boldsymbol{K}\boldsymbol{e}(\boldsymbol{q})\big) + \Big(\boldsymbol{I} - \boldsymbol{J}^{\#}(\boldsymbol{q})\boldsymbol{J}(\boldsymbol{q})\Big)\,\tilde{\boldsymbol{\omega}}$$

4. if the subpath is valid
   - extend $\mathcal{T}$
   - if $\boldsymbol{q}_{\text{new}} \in \mathcal{L}_{h+1}$, update h

   else
   - increase failure counter $m_{\text{fail}}(\boldsymbol{q}_{\text{near}})$
   - detect presence of obstruction if

     $m_{\text{fron}} \geq m_{\text{fron}}^{\max}$ and $m_{\text{fail}}(\boldsymbol{q}_j) \geq m_{\text{fail}}^{\max}$ for all $\boldsymbol{q}_j \in \mathcal{L}_h$

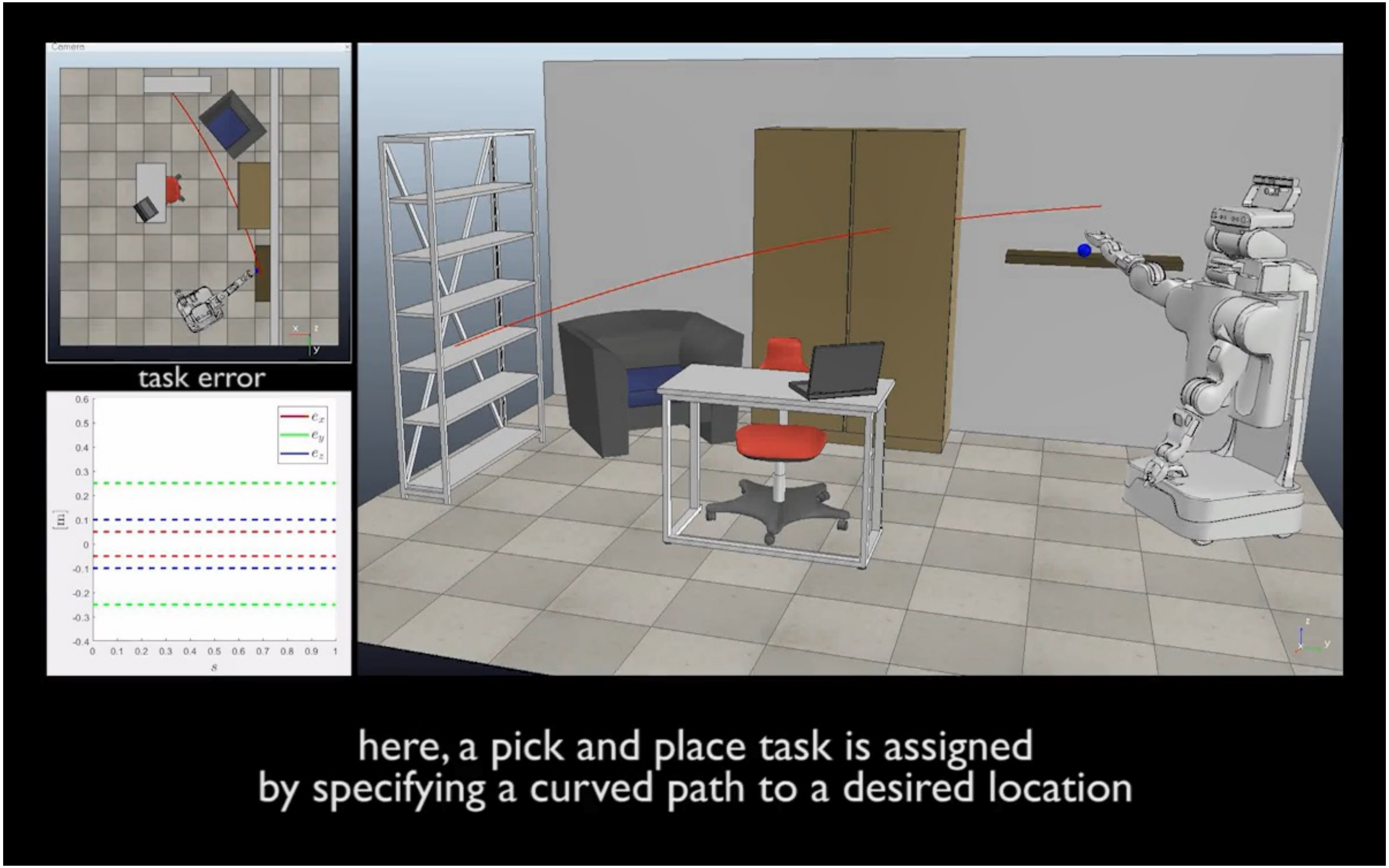


soft planner

- identify s_k where the obstruction disappears as the first sample beyond s_h where $m_{\rm free} \geq m_{\rm free}^{\rm min}$ collision-free IK solutions exist
- grow a tree $\mathcal{T}_{\rm soft}$ in $\mathcal{C}_{\rm soft}$ to find a subpath connecting $\mathcal{L}_h$ to $\mathcal{S}_k$
- root $\mathcal{T}_{\rm soft}$ at a random vertex $\boldsymbol{q}_h$ of $\mathcal{T}$ lying on $\mathcal{L}_h$
- generic iteration

1. generate a random configuration $\boldsymbol{q}_{\rm rand}$ in $\mathcal{C}_{\rm free}$
2. select in $\mathcal{T}_{\rm soft}$ the closest vertex $\boldsymbol{q}_{\rm near}$ to $\boldsymbol{q}_{\rm rand}$
3. generate $\boldsymbol{q}_{\rm curr}$ moving of η from $\boldsymbol{q}_{\rm near}$ towards $\boldsymbol{q}_{\rm rand}$
4. iteratively compute
$$\boldsymbol{q}_{\rm curr} = \boldsymbol{q}_{\rm curr} - \eta \frac{\boldsymbol{J}^T(\boldsymbol{q}_{\rm curr})\boldsymbol{e}(\boldsymbol{q}_{\rm curr}, s_{\rm curr}+\delta s)}{\|\boldsymbol{J}^T(\boldsymbol{q}_{\rm curr})\boldsymbol{e}(\boldsymbol{q}_{\rm curr}, s_{\rm curr}+\delta s)\|}$$
until $\mathcal{S}_k$ is reached or $\boldsymbol{q}_{\rm curr}$ not valid/compliant with the soft task
5. extend $\mathcal{T}_{\rm soft}$

V-REP simulations



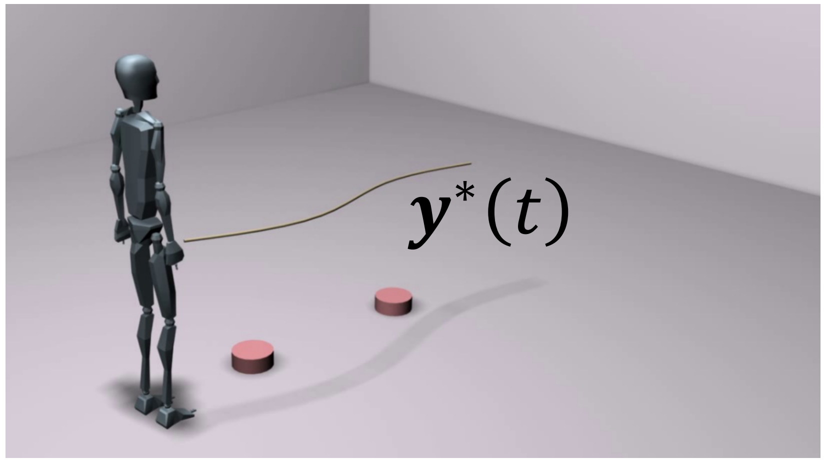
whole-body motion planning for humanoid robots

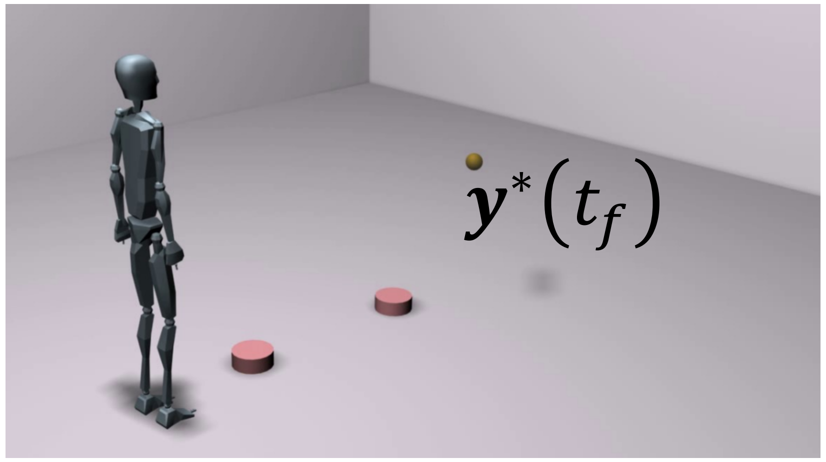
- problem: find feasible, collision-free whole-body motions for a humanoid that is assigned a task whose execution may require walking

- the generic humanoid configuration is

$$\boldsymbol{q} = \begin{pmatrix} \boldsymbol{q}_{\mathrm{CoM}} \\ \boldsymbol{q}_{\mathrm{jnt}} \end{pmatrix} \quad \begin{array}{l} \text{pose of the CoM frame} \\ n\text{-vector of joint angles} \end{array}$$

- the task is defined by a desired trajectory $\boldsymbol{y}^*(t)$ (or a geometric path $\boldsymbol{y}^*(s)$ or a single point $\boldsymbol{y}^*(t_f)$) for a specific point of the humanoid

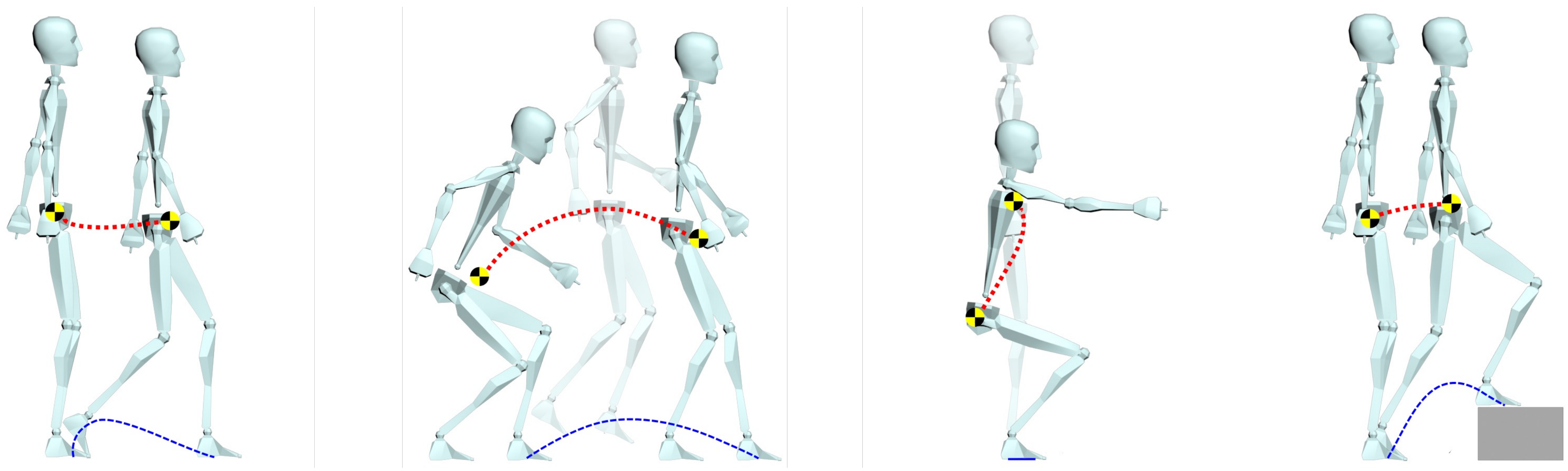




- solution to the planning problem: a whole-body motion, i.e., a configuration-space trajectory $\boldsymbol{q}(t)$, such that
  - the assigned task is realized
  - collisions with workspace obstacles and self-collisions are avoided
  - position and velocity limits on the robot joints are satisfied
  - the robot is in equilibrium at all times
- most approaches rely on a separation between locomotion and task execution, thus failing to exploit the rich humanoid motion capabilities
- our approach
  - does not separate locomotion from task execution, taking advantage of the whole-body structure of the humanoid
  - walking emerges naturally from the solution

planning based on CoM movement primitives

- main idea: build a solution by concatenating various feasible whole-body motions that realize CoM movement primitives contained in a precomputed catalogue



- a CoM movement primitive $\boldsymbol{u}_{\text{CoM}}$
  - represents an elementary humanoid motion (e.g., walking, jumping)
  - is characterized by a duration T, a CoM reference trajectory $\boldsymbol{z}^*_{\text{CoM}}(t)$ and a swing foot reference trajectory $\boldsymbol{z}^*_{\text{swg}}(t)$
  - does not specify a whole-body joint motion

motion generation

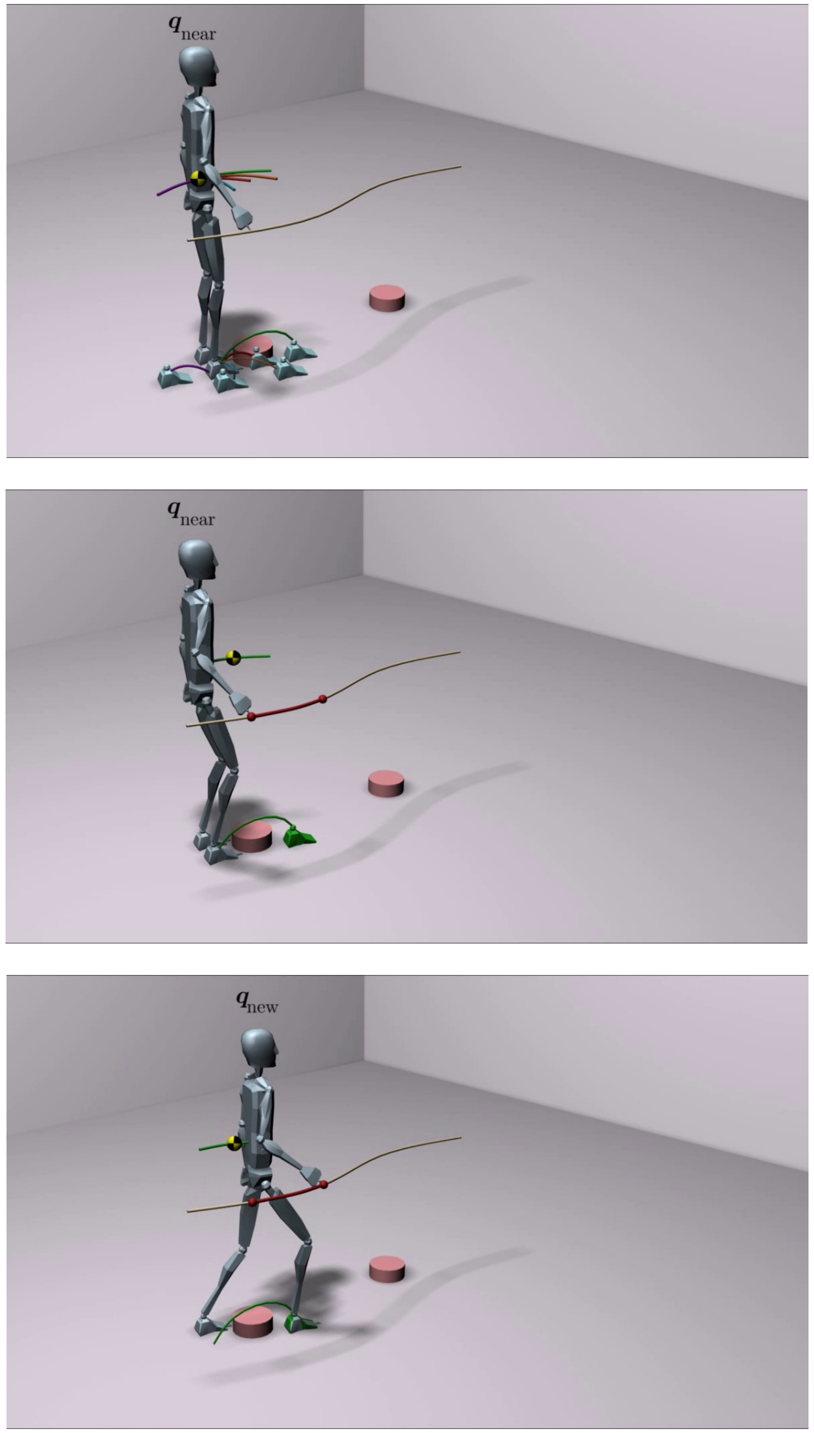
- the planner works iteratively, by repeated calls to a motion generator
- it is invoked from a certain configuration $\boldsymbol{q}_k$ at time t_k

1. CoM primitive selection: select a CoM primitive by randomly picking from the catalogue

$$U = \left\{U_{\text{CoM}}^{S} \cup U_{\text{CoM}}^{D} \cup \text{free\_CoM}\right\}$$

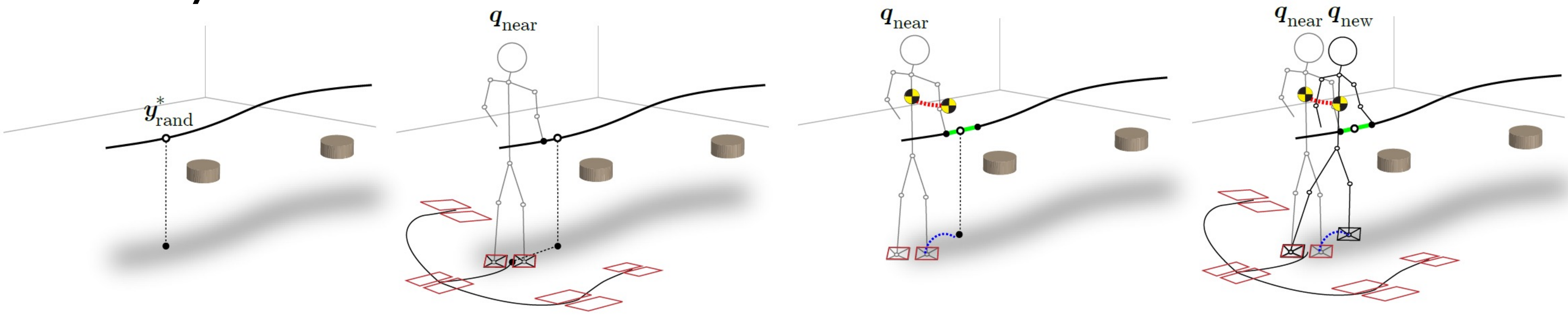
2. joint motion generation: produce joint motion from $\boldsymbol{q}_k$ at t_k that realizes the selected CoM primitive and the portion of the assigned task in $[t_k, t_{k+1}]$ by integrating

$$\dot{\boldsymbol{q}}_{\text{jnt}} = \boldsymbol{J}_a^{\#}(\dot{\boldsymbol{y}}_a^{*} + \boldsymbol{K}\boldsymbol{e}) + \left(\boldsymbol{I} - \boldsymbol{J}_a^{\#}\boldsymbol{J}_a\right)\boldsymbol{\omega}$$



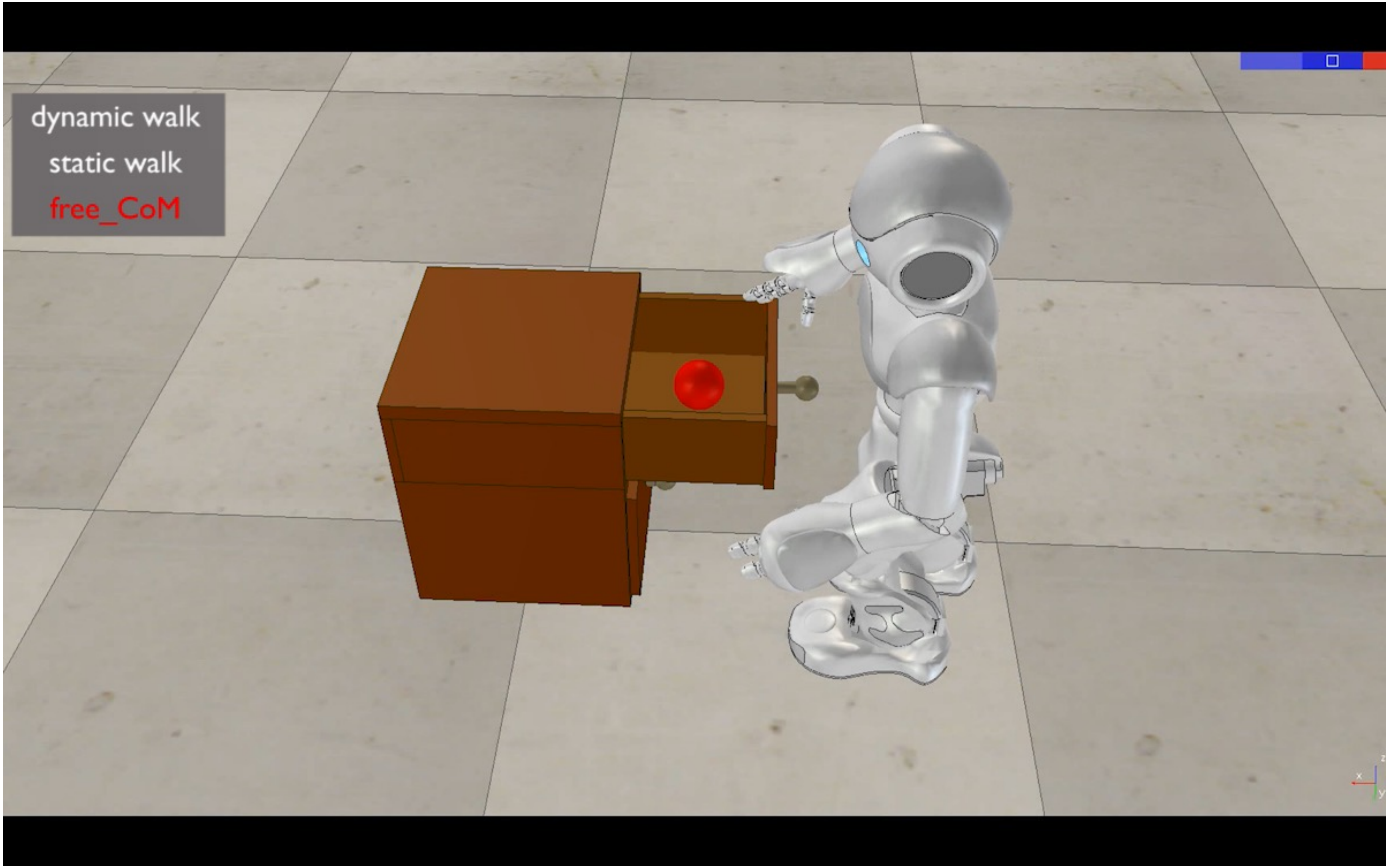
planner overview

iteratively build a tree



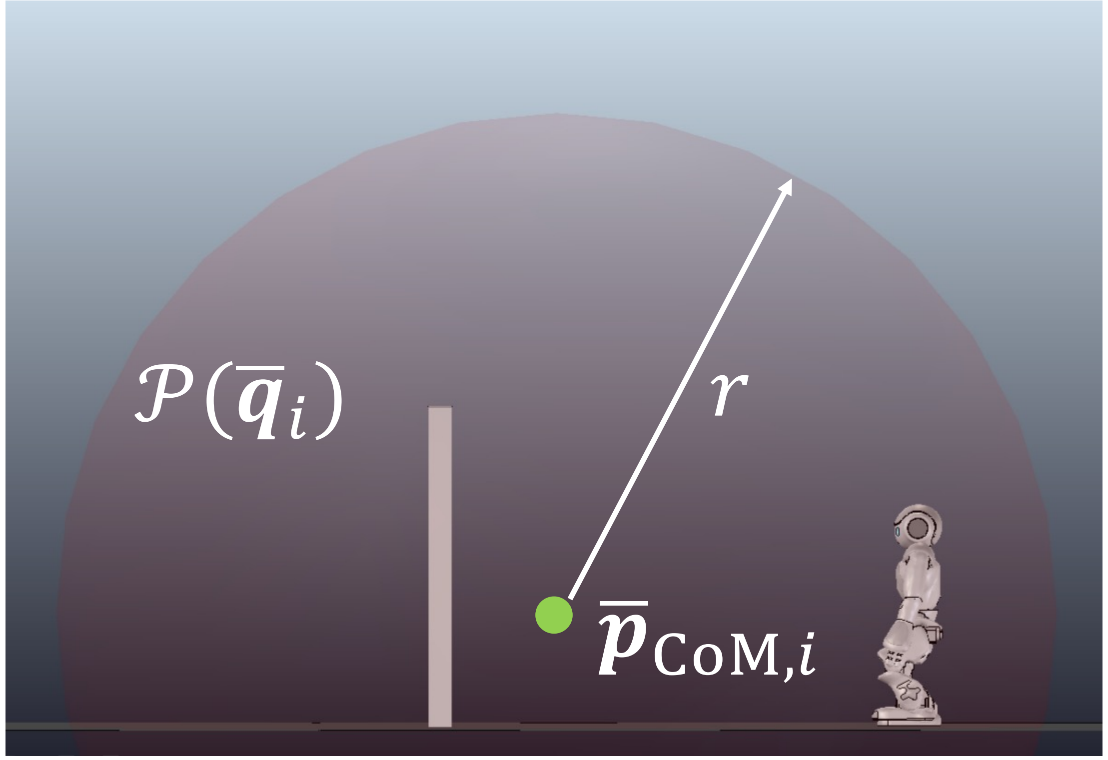
1. choose a random sample $\boldsymbol{y}^*_{\mathrm{rand}}$ on the assigned task trajectory
2. extract a configuration $\boldsymbol{q}_{\mathrm{near}}$ with probability proportional to a task compatibility function $\gamma(\cdot, \boldsymbol{y}^*_{\mathrm{rand}})$
3. select a random CoM primitive and extract the portion of the task trajectory to be executed
4. generate the joint motion to realize the CoM primitive and the portion of the task
5. if the motion is feasible and collision-free, add the final configuration $\boldsymbol{q}_{\mathrm{new}}$ to the tree

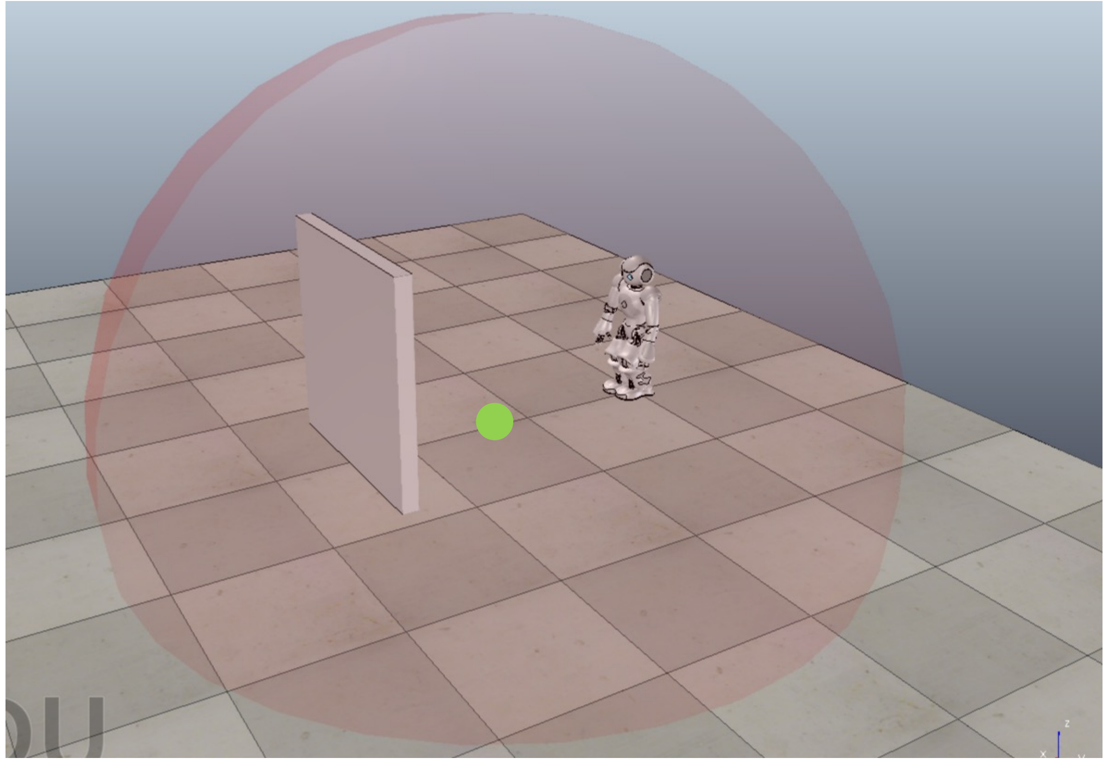
V-REP simulations



anytime whole-body planning

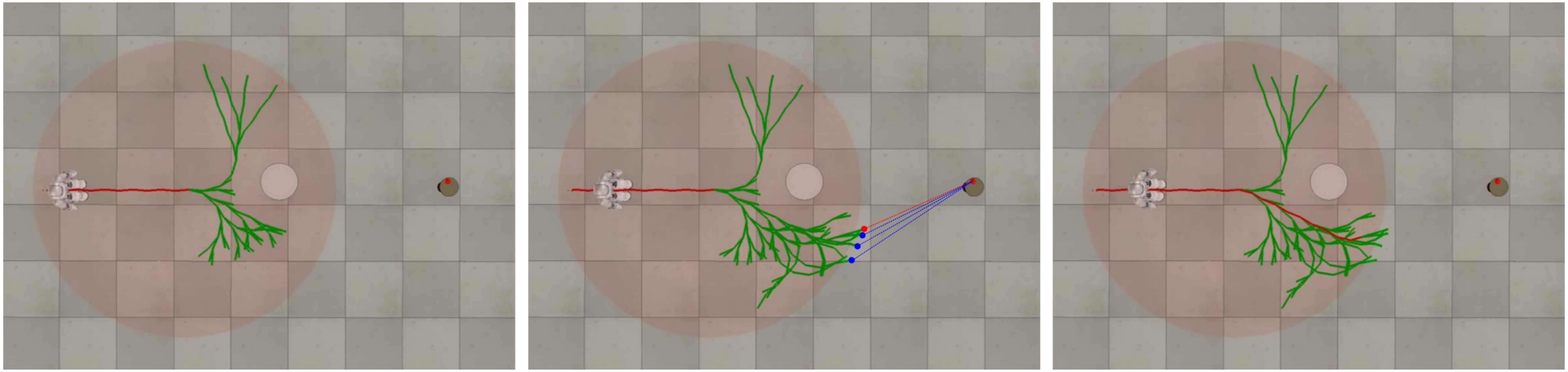
- problem: plan the humanoid motion under time limitations
- approach: simultaneously perform planning and execution of local plans:
  - execute the current local plan $\boldsymbol{q}_{i-1}(t), t \in [t_{i-1}, t_i]$
  - plan the next local plan $\boldsymbol{q}_i(t), t \in [t_i, t_{i+1}]$, that
    - starts at $\overline{\boldsymbol{q}}_i = \boldsymbol{q}_{i-1}(t_i)$
    - is feasible within a limited planning zone $\mathcal{P}(\overline{\boldsymbol{q}}_i)$



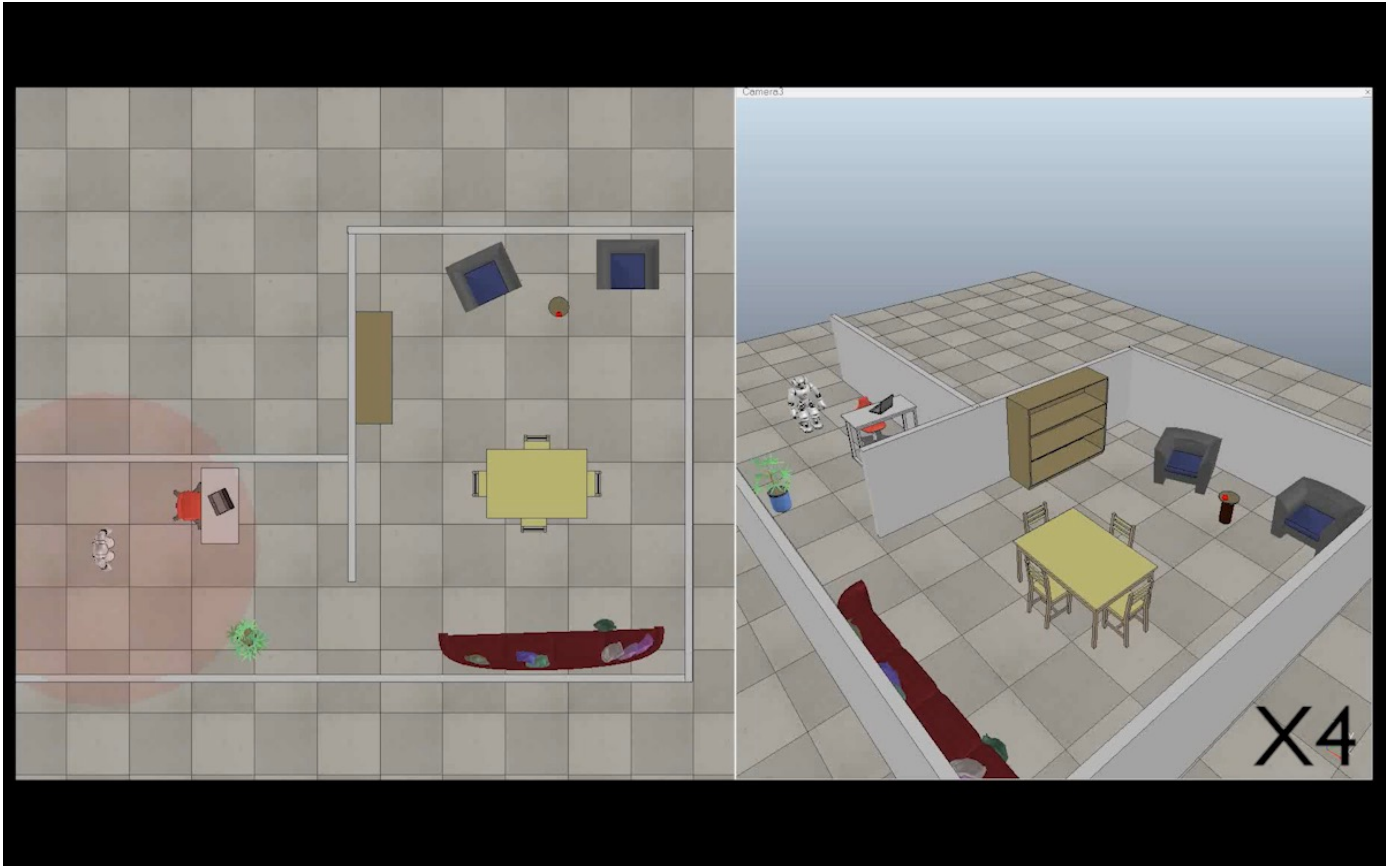


local motion planner

- runs for a given time budget ΔT_P
- works in two stages
  - lazy stage: quickly populate $\mathcal{T}$ with partial configurations $\boldsymbol{q} = (\boldsymbol{q}_{\rm CoM}, \emptyset)$, checking collisions only at vertexes using a simplified occupancy volume for the robot
  - validation stage: generate the joint motion $\boldsymbol{q}_{\rm jnt}(t)$ associated to the best candidate plan, checking collisions both at vertexes and along edges using the actual occupancy volume for the robot

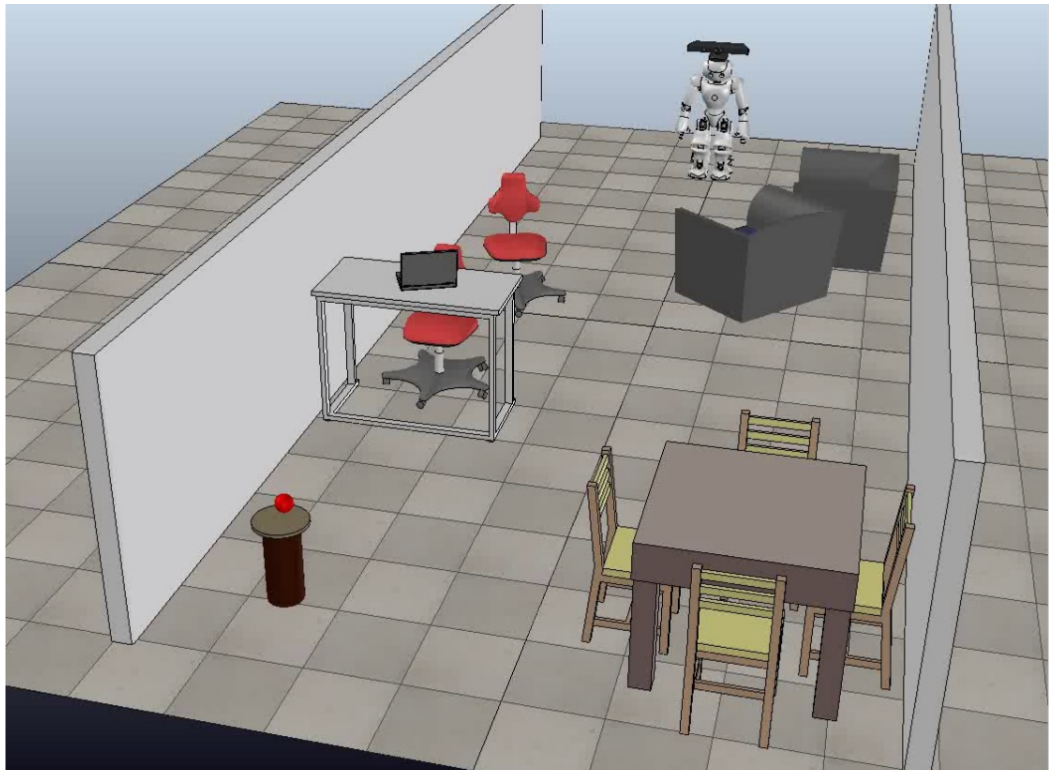


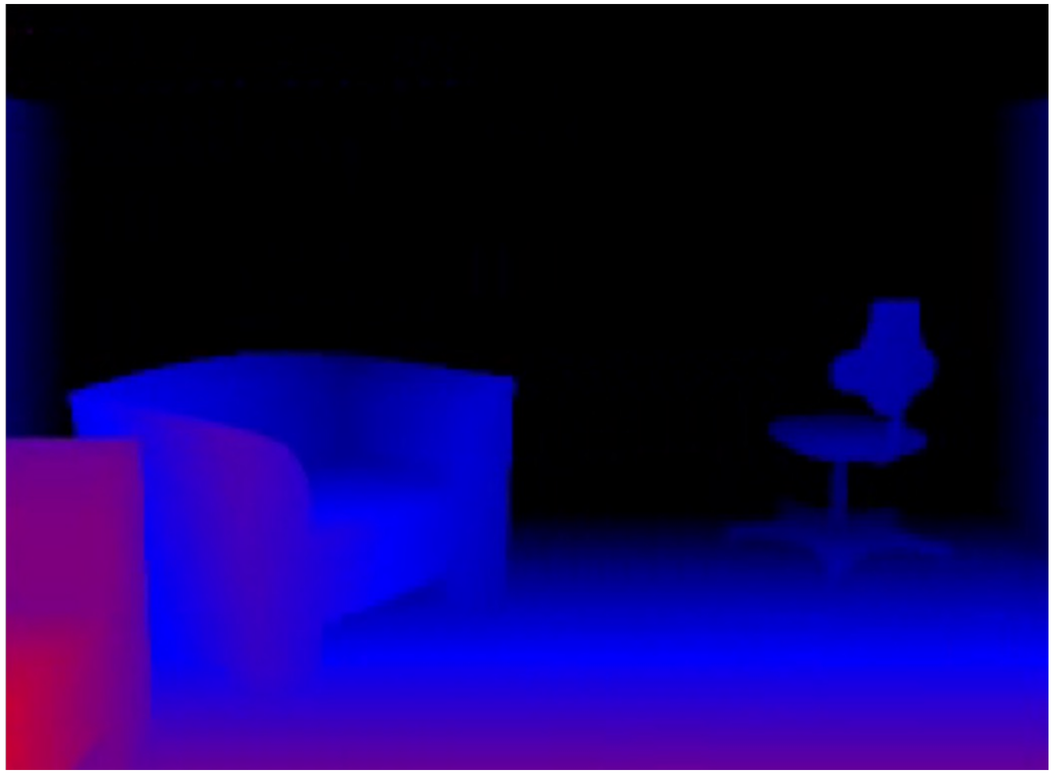
V-REP simulations



sensor-based whole-body planning

- problem: plan the humanoid motion in unknown environments
- the robot is equipped with a head-mounted depth camera, e.g., a Kinect
- it is assumed that the robot is localized via an external module

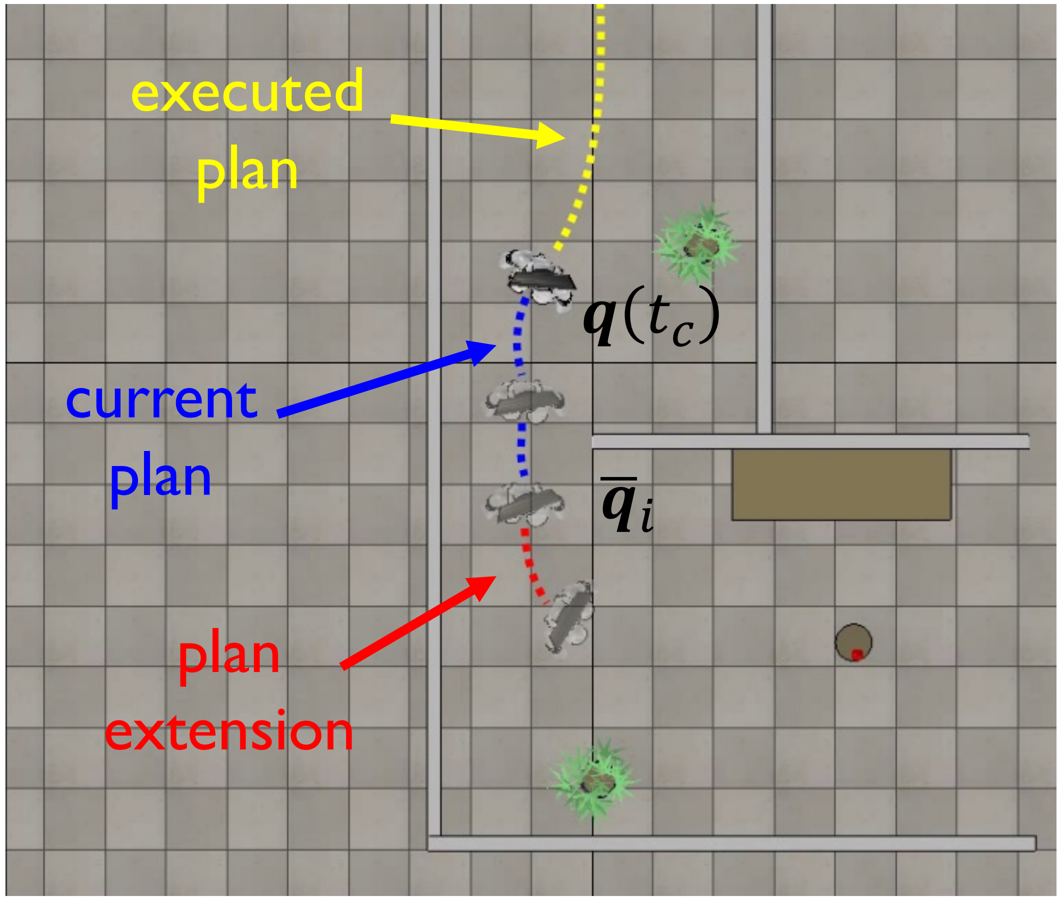




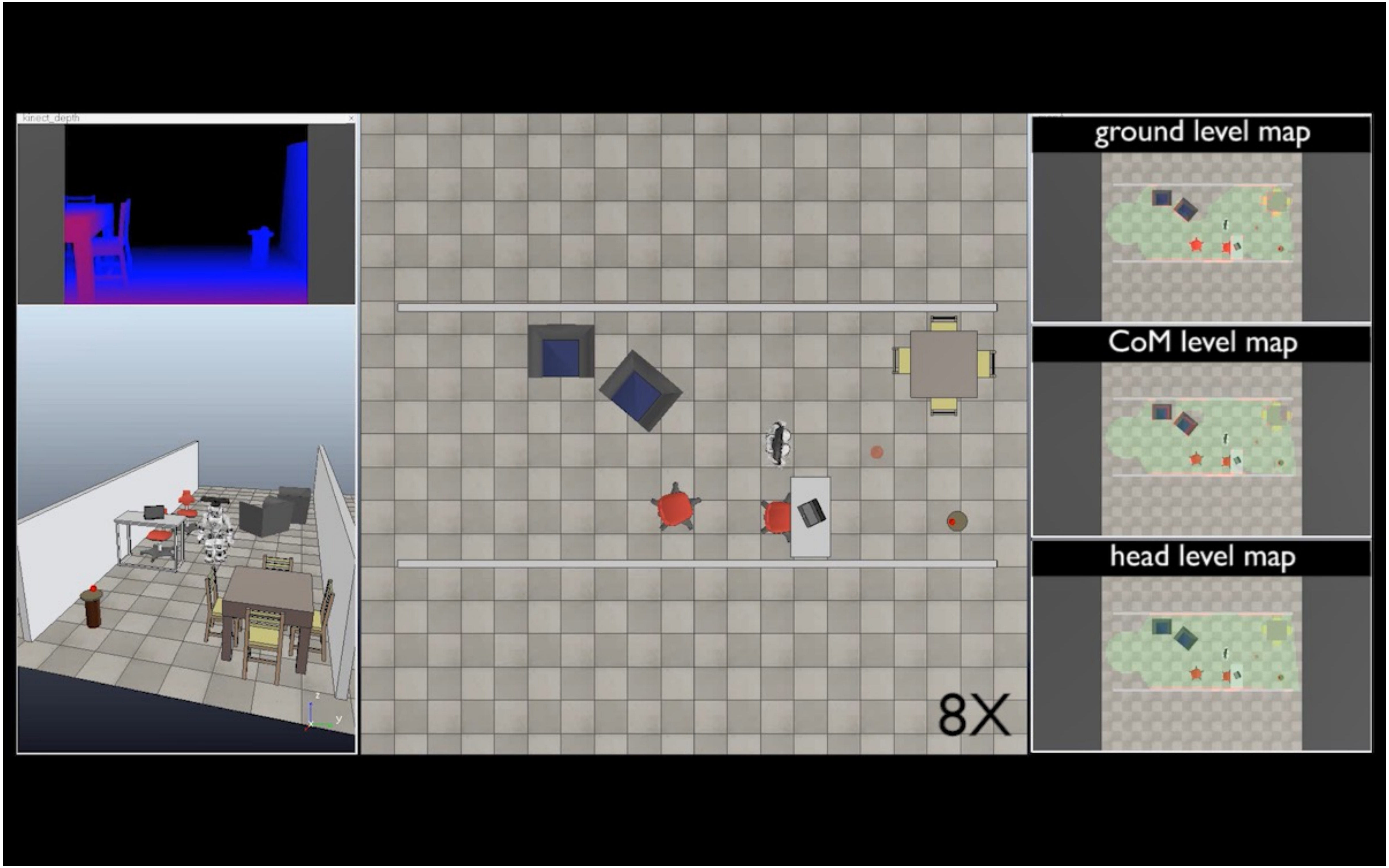
- approach: simultaneously perform mapping, planning and execution

framework overview

- mapping module: continuously integrates information gathered by the depth camera into a 3D occupancy grid map $\mathcal{M}$
- execution module: sends the commands to the humanoid actuators based on the current plan
- planning module: extends the current plan with feasible whole-body motions by repeatedly invoking the local motion planner providing
  - the final configuration $\bar{\boldsymbol{q}}_i$
  - the planning map $\mathcal{M}_{P,i} = \mathcal{M}(t_c)$
  - the time budget $\Delta T_{P,i} = \alpha_P (t_i - t_c)$

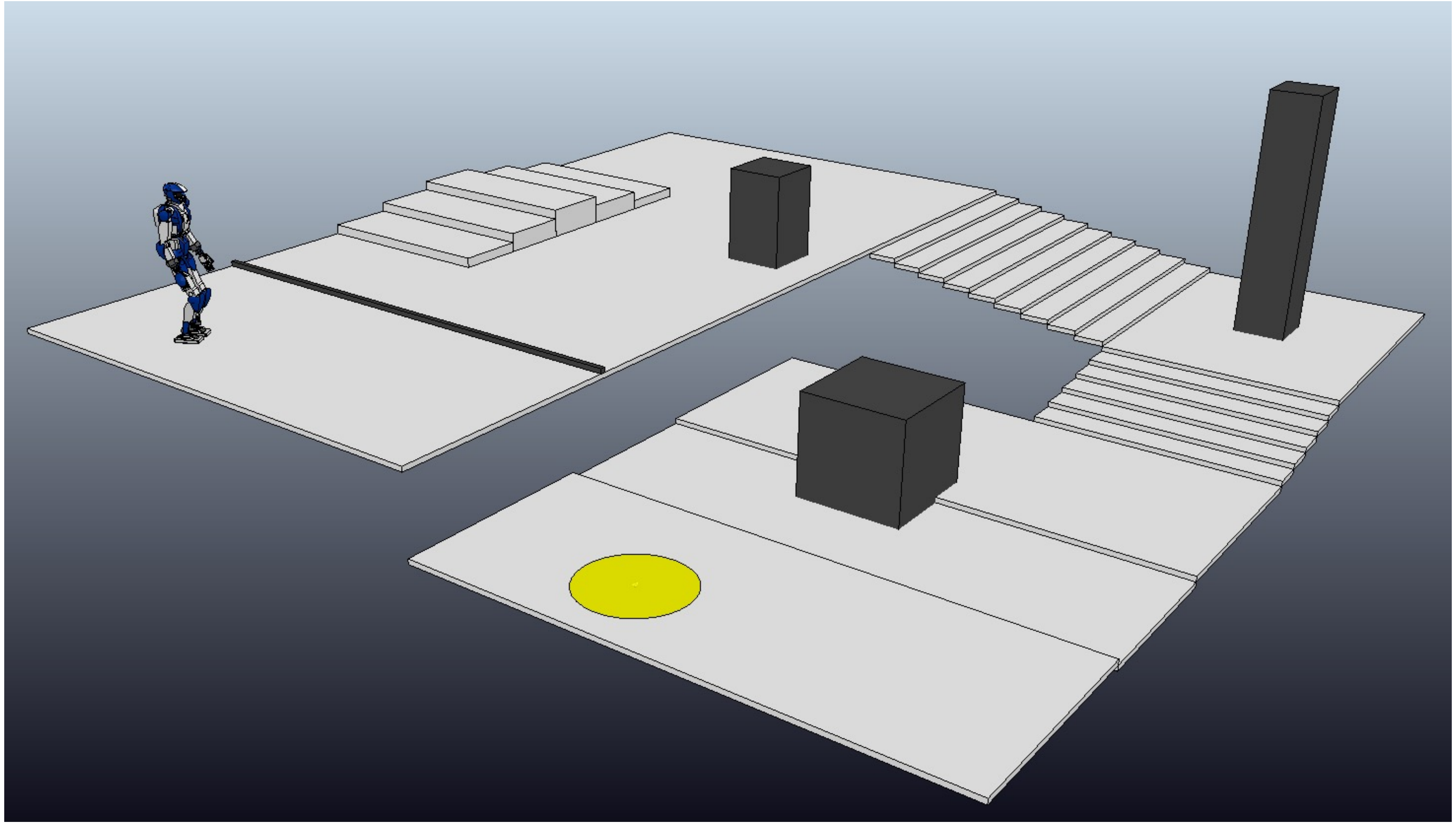


V-REP simulations



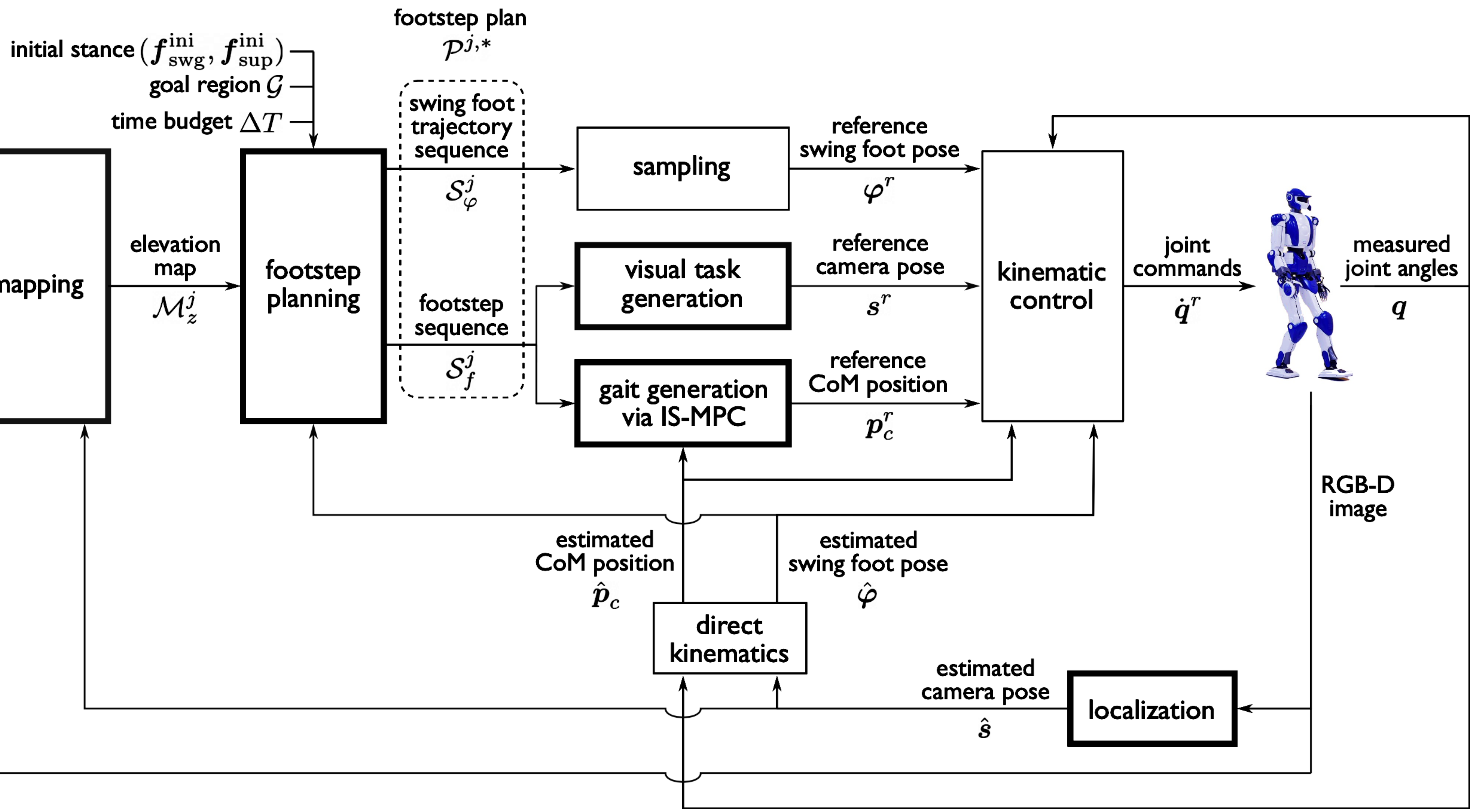
humanoid motion generation in a world of stairs

- problem: generate humanoid motion in an unknown world of stairs
- the robot is equipped with a head-mounted depth camera
- it is assumed that the robot is localized via an external module



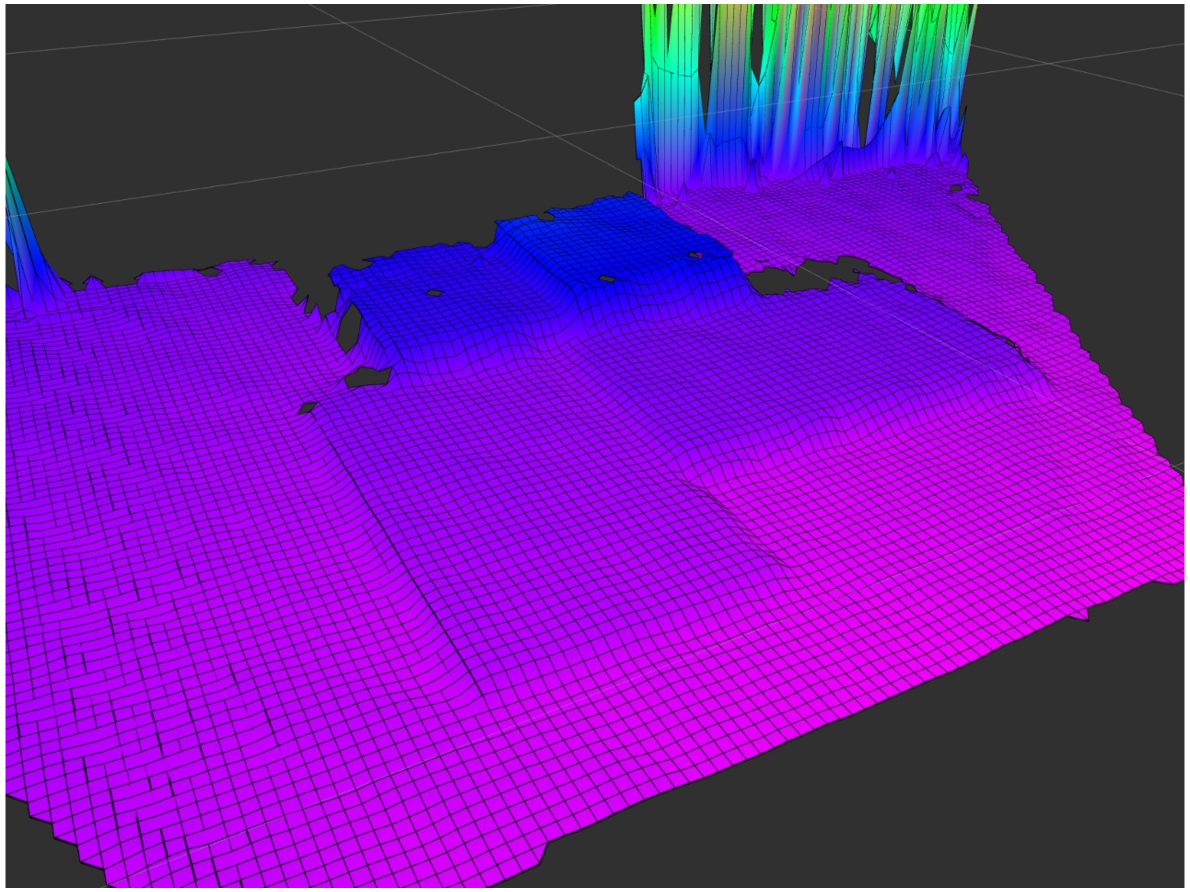
- approach: simultaneously perform mapping, planning and execution

general architecture



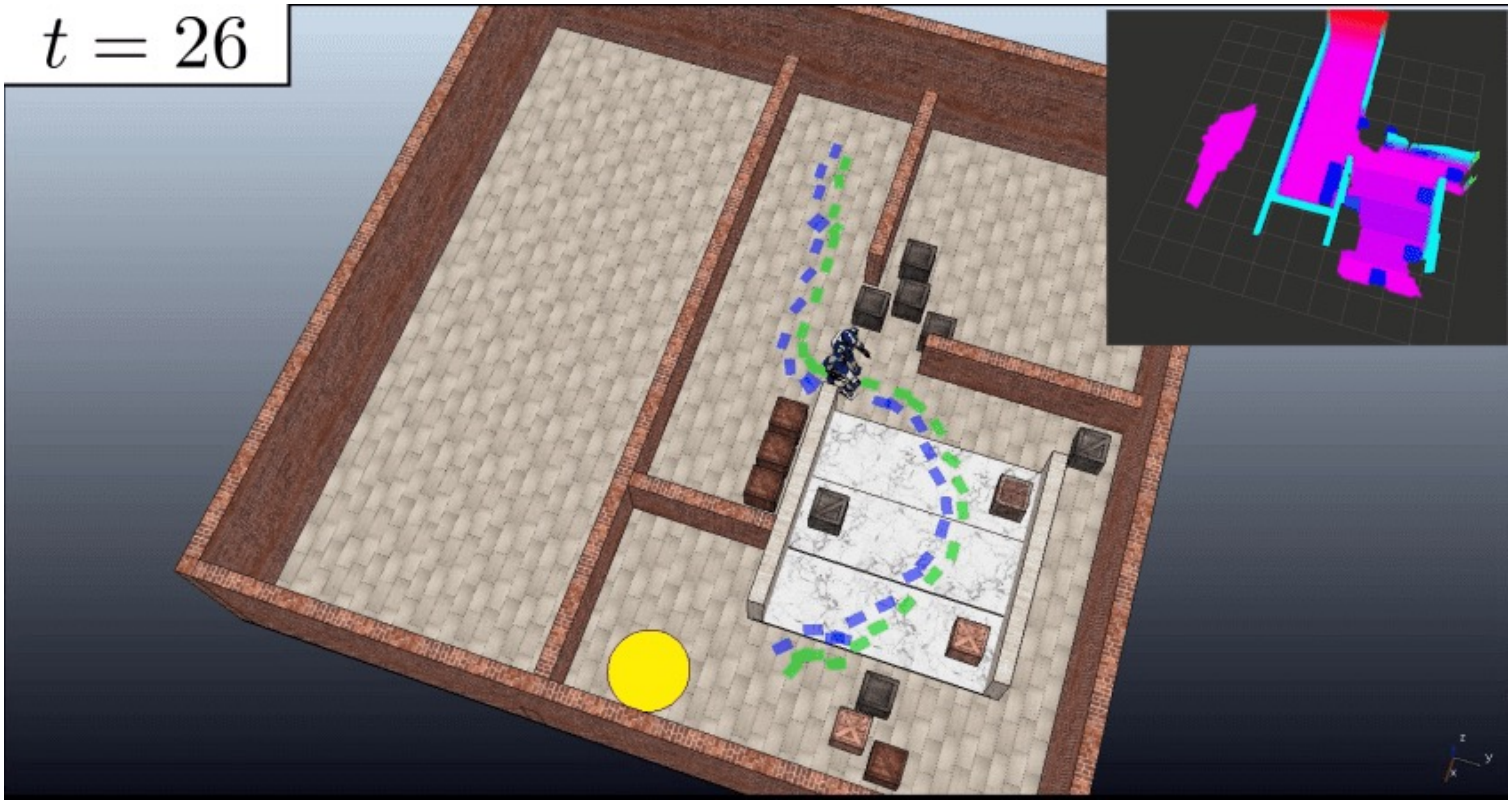
elevation mapping

- represent the world of stairs as an elevation map
- continuously update the map using on-board sensors
- dynamic environments using visibility check based on ray tracing



sensor-based footstep planning

- given current stance and current elevation map, generate a footstep plan in the direction of an unknown area of the environment
- take advantage of the motion duration to refine the footstep plan



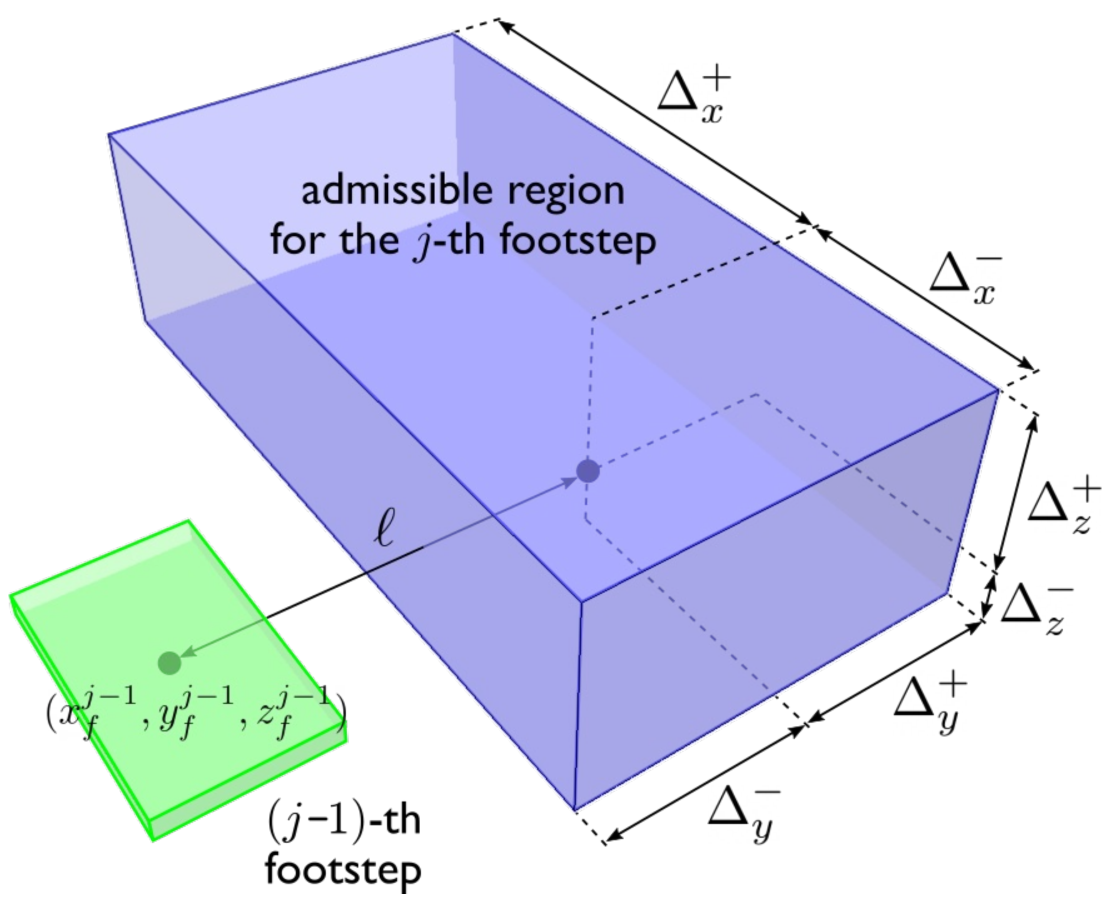
sensor-based footstep planning: feasibility

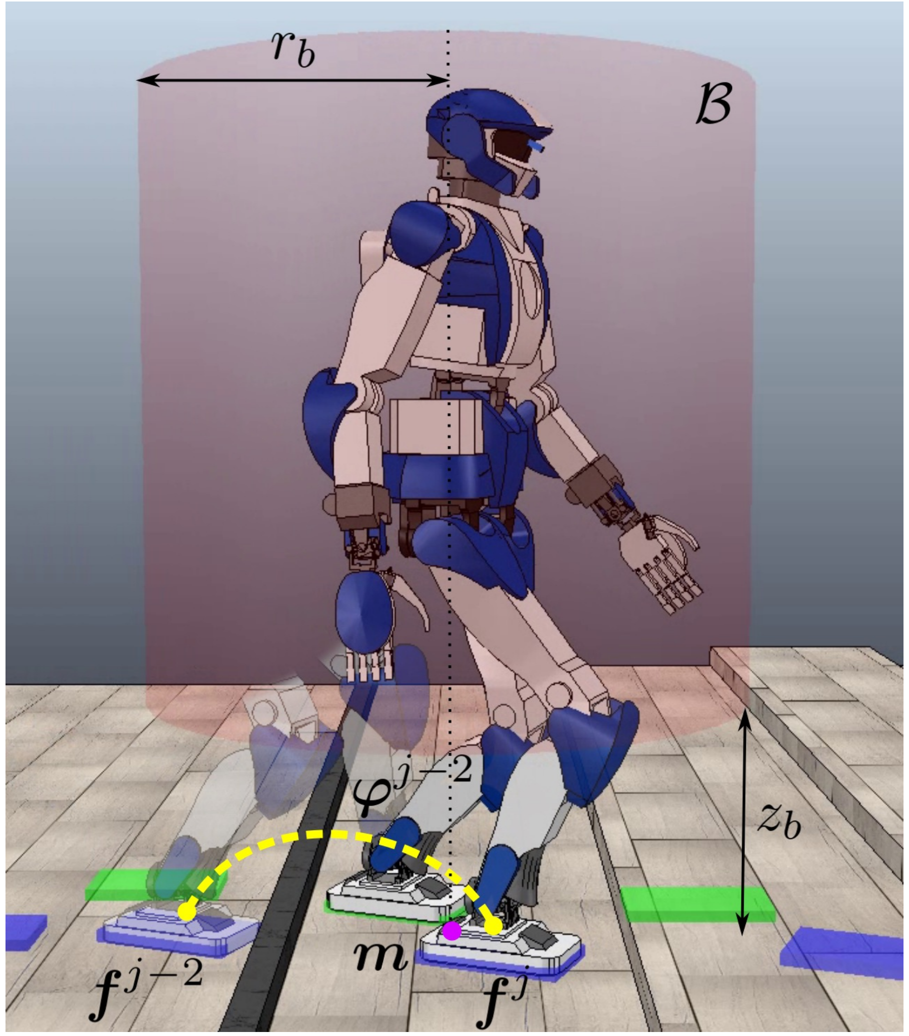
- requirements:

  R1: footstep fully in contact with a single horizontal patch

  R2: stance feasibility

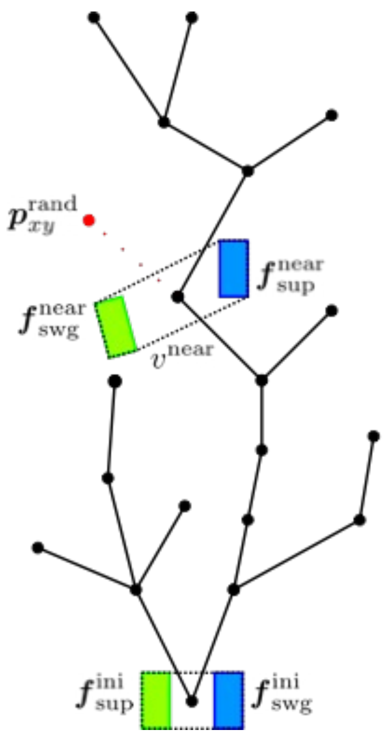
  R3: step feasibility



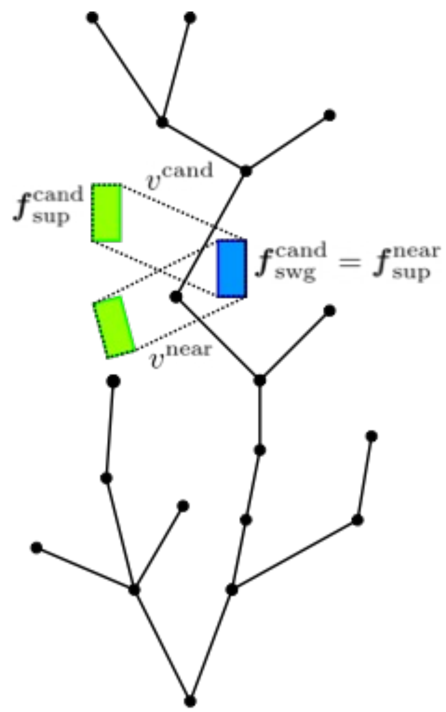


sensor-based footstep planning: algorithm

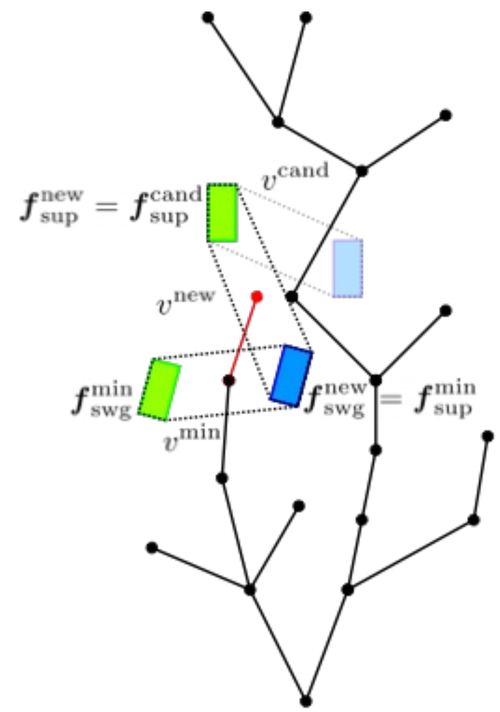
- RRT*-based footstep planner optimizing for number of steps, height variation or clearance



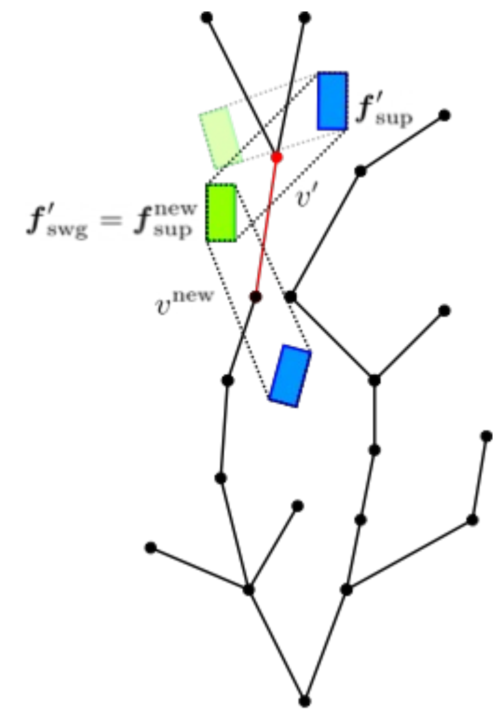
vertex selection



expansion



choose parent



rewire

gait generation via IS-MPC: 3D motion model

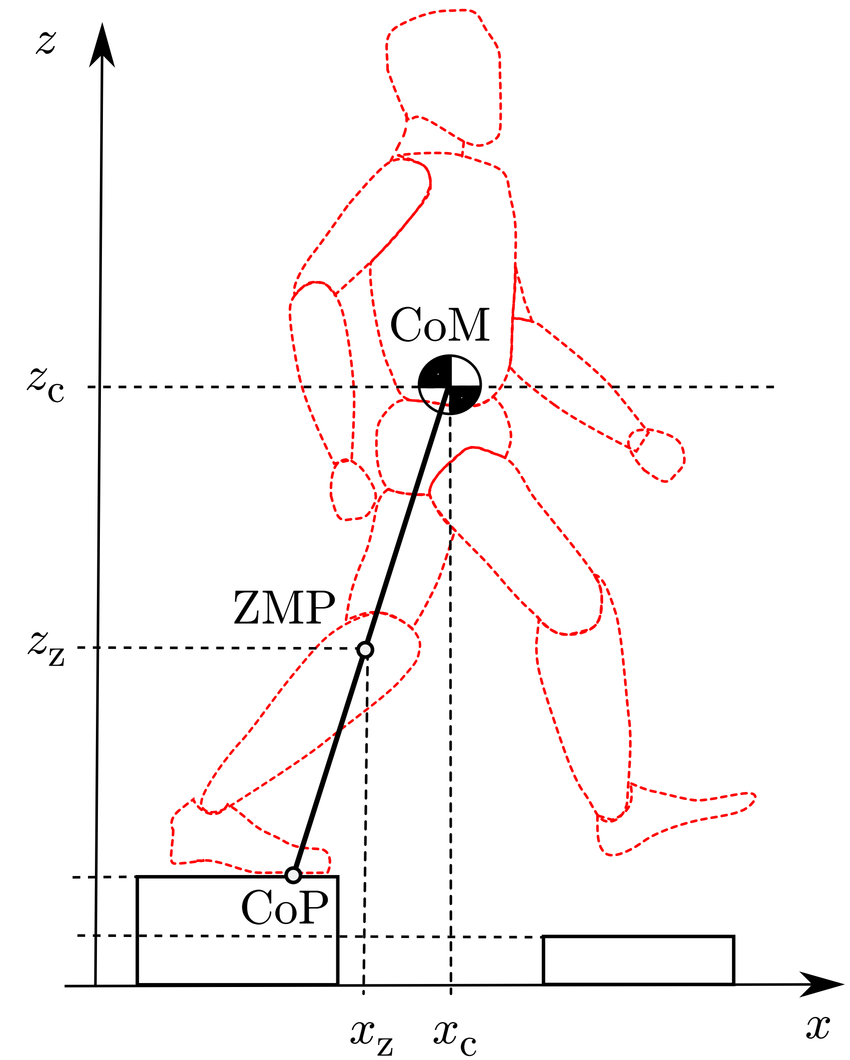
- LIPM not suitable for gait generation over uneven terrain due to constant height assumption
- linearity can be maintained by constraining vertical motion such that

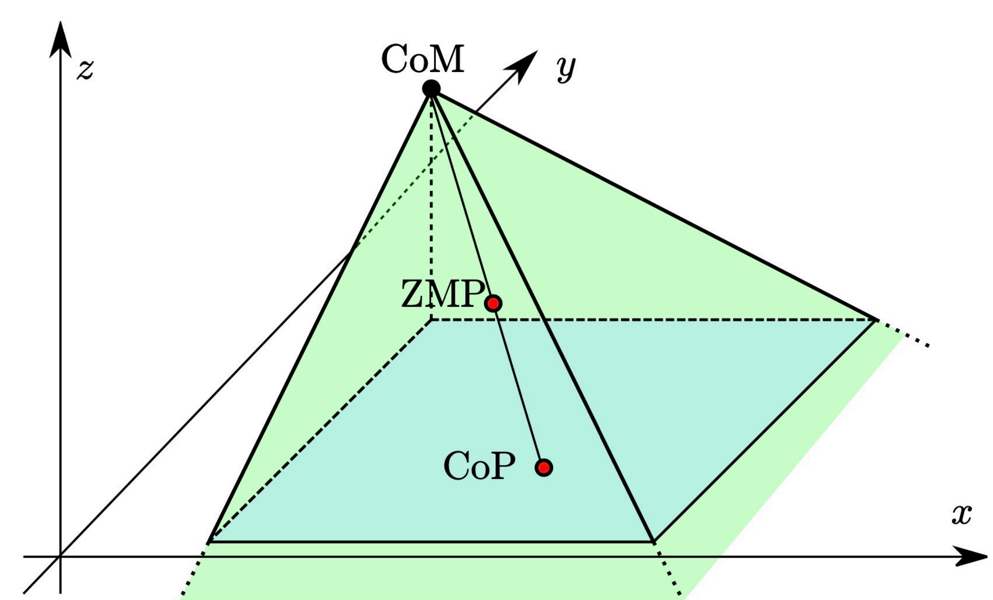
$$\frac{\ddot{z}_c + g}{z_c - z_z} = \eta^2$$

- CoM dynamics become

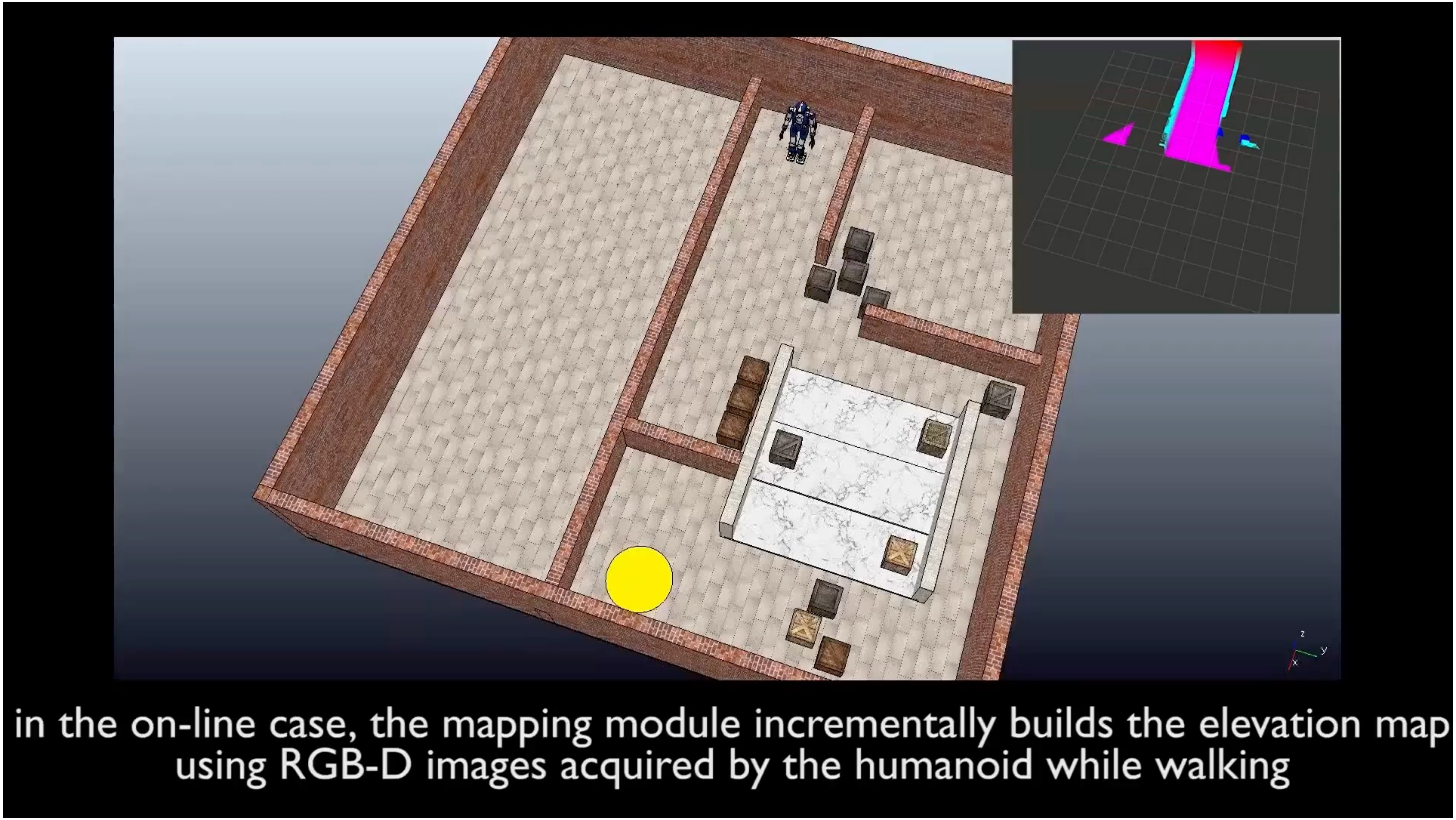
$$\ddot{p}_c = \eta^2 (p_c - p_z) - g$$

- solve QP problem using MPC scheme



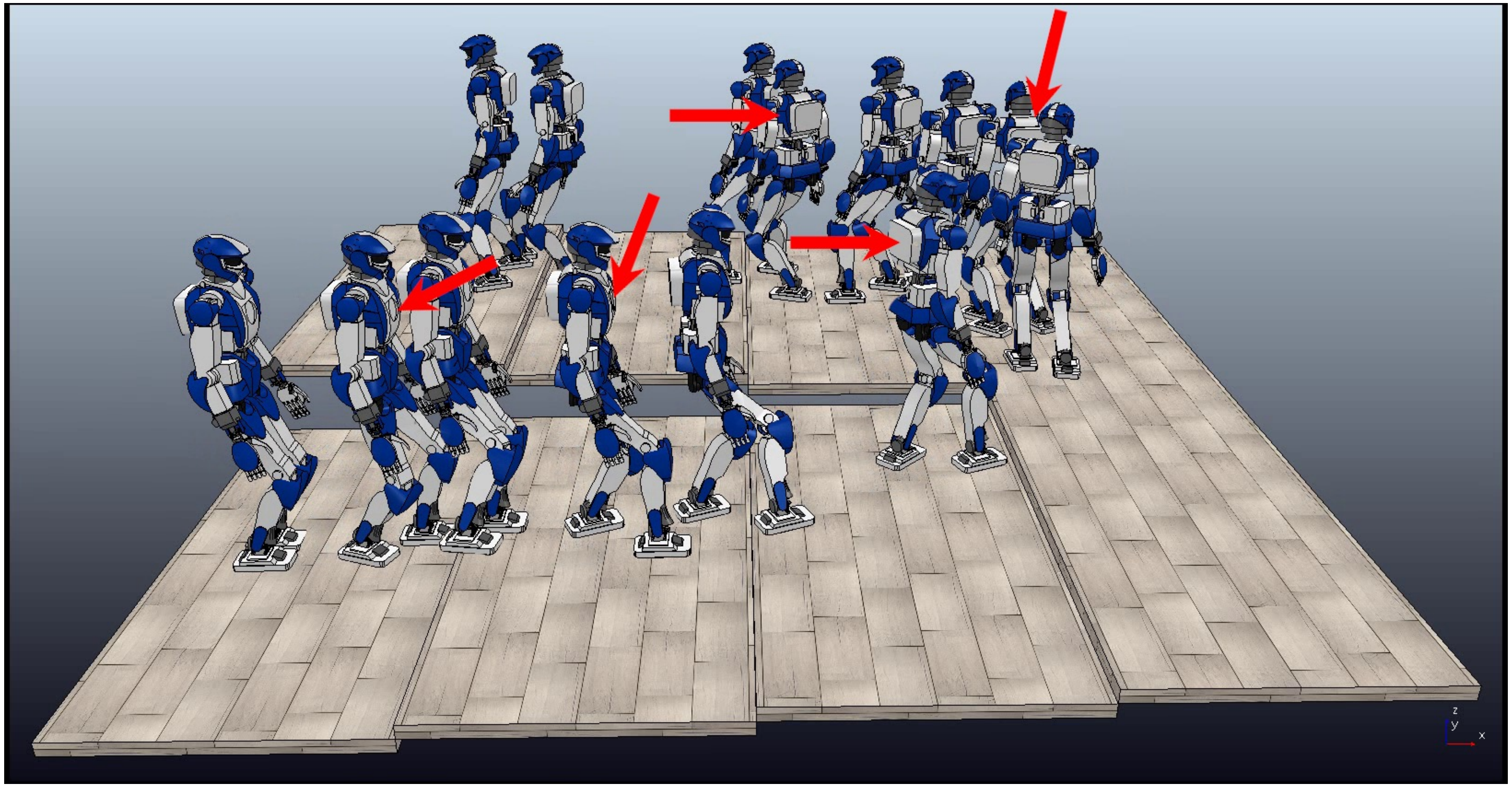


V-REP simulations



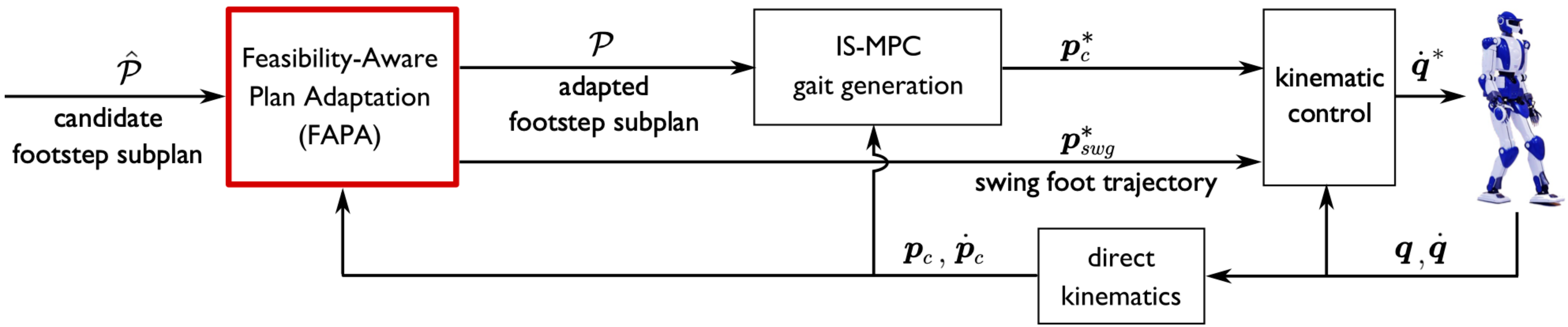
feasibility-aware plan adaptation in humanoid gait generation

- problem: generate a feasible gait, even in case of disturbances
- it is assumed that the state of the robot is known



- approach: adapt footstep plan (positions, orientations and timings)

architecture



- Feasibility-Aware Plan Adaptation (FAPA) module, it adapts the footstep plan before the IS-MPC stage
- being external to the MPC, it can enforce nonlinear constraints without a significant impact on the performance of the overall scheme

FAPA module

- the adaptation is done in such a way to always make the IS-MPC stage feasible (i.e., able to satisfy balance and stability constraints)
- the feasibility of the MPC can be studied analytically and the resulting conditions constitute gait feasibility constraints in FAPA
- this ties the adaptation to the robot dynamics and makes it able to react to pushes and disturbances
- F-FAPA is an NLP and V-FAPA is a mixed-integer NLP

FAPA module

- cost function minimizes displacement from the subplan subject to
    a. kinematic constraints
    b. timing constraints
    c. fixed (F-FAPA) or variable (V-FAPA) patch constraints
    d. current footsteps constraints
    e. gait feasibility constraints
- the gait feasibility constraints ensure that the state is in the feasibility region of the IS-MPC

$$x_u^k + b_x^k \leq s^T Z^{-1} \left(M X_f^l + m x_f^l + z \left(\frac{d_x}{2} - x_z^k \right) \right)$$

results and V-REP simulations

references

- V-REP User Manual, link: http://www.coppeliarobotics.com/helpFiles/index.html
- M. Cefalo, G. Oriolo, "*A general framework for task-constrained motion planning with moving obstacles*", Robotica, vol. 37, pp. 575-598, 2019
- M. Cefalo, P. Ferrari, G. Oriolo, "*An opportunistic strategy for motion planning in the presence of soft task constraints*", IEEE Robotics and Automation Letters, vol. 5, no. 4, pp. 6294-6301, 2020
- M. Cognetti, P. Mohammadi, G. Oriolo, "*Whole-body motion planning for humanoids based on CoM movement primitives*", 2015 IEEE-RAS International Conference on Humanoid Robots, Seoul, South Korea, pp. 1090-1095, 2015
- P. Ferrari, M. Cognetti, G. Oriolo, "*Anytime whole-body planning/replanning for humanoid robots*", 2018 IEEE-RAS International Conference on Humanoid Robots, Beijing, China, pp. 1-9, 2018
- P. Ferrari, M. Cognetti, G. Oriolo, "*Sensor-based whole-body planning/replanning for humanoid robots*", 2019 IEEE-RAS International Conference on Humanoid Robots, Toronto, Canada, pp. 535-541, 2019
- M. Cipriano, P. Ferrari, N. Scianca, L. Lanari, G. Oriolo, "*Humanoid Motion Generation in a World of Stairs*", Robotics and Autonomous Systems, vol. 168, 104495, 2023
- M. Cipriano, M. R.O.A. Maximo, N. Scianca, L. Lanari, G. Oriolo, "*Feasibility-Aware Plan Adaptation in Humanoid Gait Generation*", 2023 IEEE-RAS International Conference on Humanoid Robots, Austin, USA, 2023