Humanoid Path Planner

Florent Lamiraux and Joseph Mirabel

CNRS-LAAS, Toulouse, France

Humanoid Path Planner

Outline

Path Planning

Given

- A robot (kinematic chain),
- obstacles,
- constraints (non-holonomic, manipulation),
- an initial configuration and
- goal configurations,

Compute a collision-free path satisfying the constraints from the initial configuration to a goal configuration.

Path Planning

Given

- A robot (kinematic chain),
- obstacles,
- constraints (non-holonomic, manipulation),
- an initial configuration and
- goal configurations,

Compute a collision-free path satisfying the constraints from the initial configuration to a goal configuration.

Path Planning

Given

- A robot (kinematic chain),
- obstacles,
- constraints (non-holonomic, manipulation),
- an initial configuration and
- goal configurations,

Compute a collision-free path satisfying the constraints from the initial configuration to a goal configuration.

Path Planning

Given

- A robot (kinematic chain),
- obstacles,
- constraints (non-holonomic, manipulation),
- an initial configuration and
- goal configurations,

Compute a collision-free path satisfying the constraints from the initial configuration to a goal configuration.

Path Planning

Given

- A robot (kinematic chain),
- obstacles,
- constraints (non-holonomic, manipulation),
- an initial configuration and
- goal configurations,

Compute a collision-free path satisfying the constraints from the initial configuration to a goal configuration.

Historical perspective

- 1998: Move3D,
- 2001: Creation of Kineo-CAM, transfer of Move3D,
- 2006: Release of KineoWorks-2, development of HPP based on KineoWorks-2,
- 2013: kineo-CAM is bought by Siemens,
- December 2013: development of HPP open-source.

Historical perspective

- 1998: Move3D,
- 2001: Creation of Kineo-CAM, transfer of Move3D,
- 2006: Release of KineoWorks-2, development of HPP based on KineoWorks-2,
- 2013: kineo-CAM is bought by Siemens,
- December 2013: development of HPP open-source.

Historical perspective

- 1998: Move3D,
- 2001: Creation of Kineo-CAM, transfer of Move3D,
- 2006: Release of KineoWorks-2, development of HPP based on KineoWorks-2,
- 2013: kineo-CAM is bought by Siemens,
- December 2013: development of HPP open-source.

Historical perspective

- 1998: Move3D,
- 2001: Creation of Kineo-CAM, transfer of Move3D,
- 2006: Release of KineoWorks-2, development of HPP based on KineoWorks-2,
- 2013: kineo-CAM is bought by Siemens,
- December 2013: development of HPP open-source.

Historical perspective

- 1998: Move3D,
- 2001: Creation of Kineo-CAM, transfer of Move3D,
- 2006: Release of KineoWorks-2, development of HPP based on KineoWorks-2,
- 2013: kineo-CAM is bought by Siemens,
- December 2013: development of HPP open-source.

Main features

- Numerical constraints at the core of the model
  - quasi-static equilibrium
  - object grasp and placement
- no a priori discretization of paths
  - evaluation calls constraint projection
  - constrained path need to be checked for continuity (class `hpp::core::PathProjector`)

Main features

- Numerical constraints at the core of the model
  - quasi-static equilibrium
  - object grasp and placement
- no a priori discretization of paths
  - evaluation calls constraint projection
  - constrained path need to be checked for continuity (class `hpp::core::PathProjector`)

Outline

Overview of the architecture

Modular: collection of packages

- package dependencies tracked by `pkg-config`,
- installation managed by `cmake` and a `git` submodule:
  `git://github.com/jrl-umi3218/jrl-cmakemodules.git`,
- programmed in `C++`,
- controlled via `python`

Overview of the architecture

Modular: collection of packages

- package dependencies tracked by `pkg-config`,
- installation managed by `cmake` and a `git` submodule: `git://github.com/jrl-umi3218/jrl-cmakemodules.git`,
- programmed in `C++`,
- controlled via `python`

Overview of the architecture

Modular: collection of packages

- package dependencies tracked by `pkg-config`,
- installation managed by `cmake` and a `git` submodule:

  `git://github.com/jrl-umi3218/jrl-cmakemodules.git`,
- programmed in `C++`,
- controlled via `python`

Overview of the architecture

Modular: collection of packages

- package dependencies tracked by `pkg-config`,
- installation managed by `cmake` and a `git` submodule:

  `git://github.com/jrl-umi3218/jrl-cmakemodules.git`,
- programmed in `C++`,
- controlled via `python`

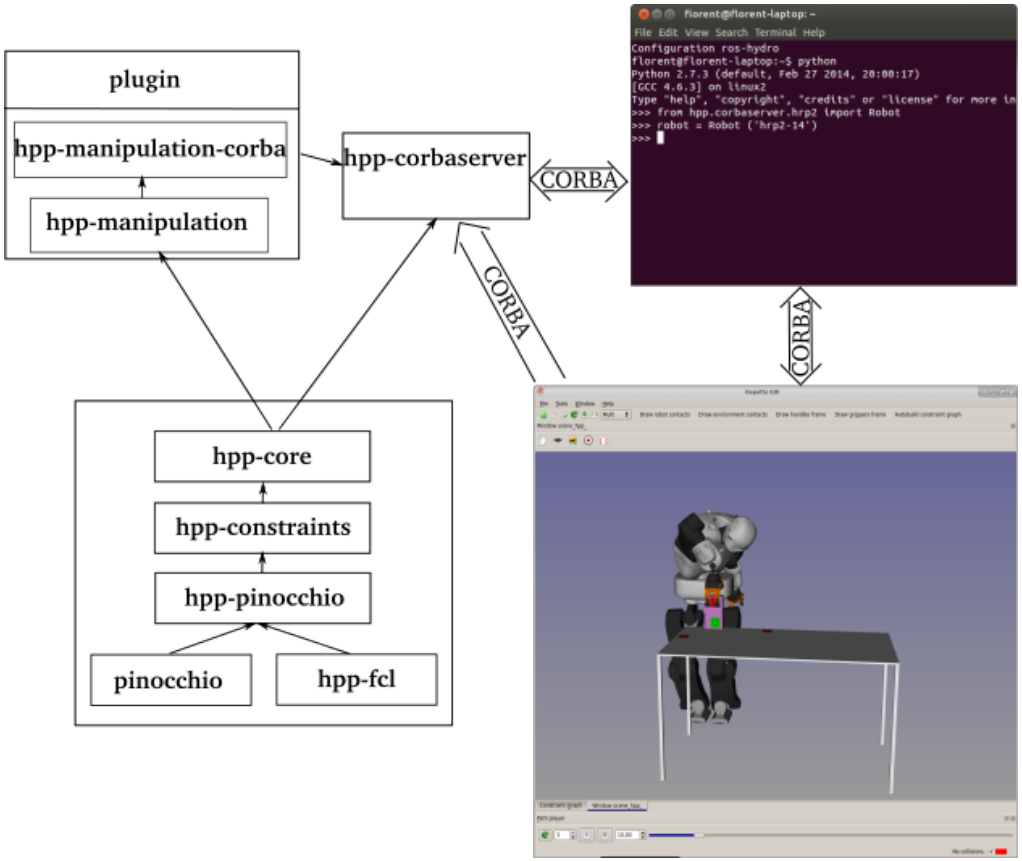
Overview of the architecture

Modular: collection of packages

- package dependencies tracked by `pkg-config`,
- installation managed by `cmake` and a `git` submodule:

  `git://github.com/jrl-umi3218/jrl-cmakemodules.git`,
- programmed in `C++`,
- controlled via `python`

Overview of the architecture



Software Development Kit

Packages implementing the core infrastructure

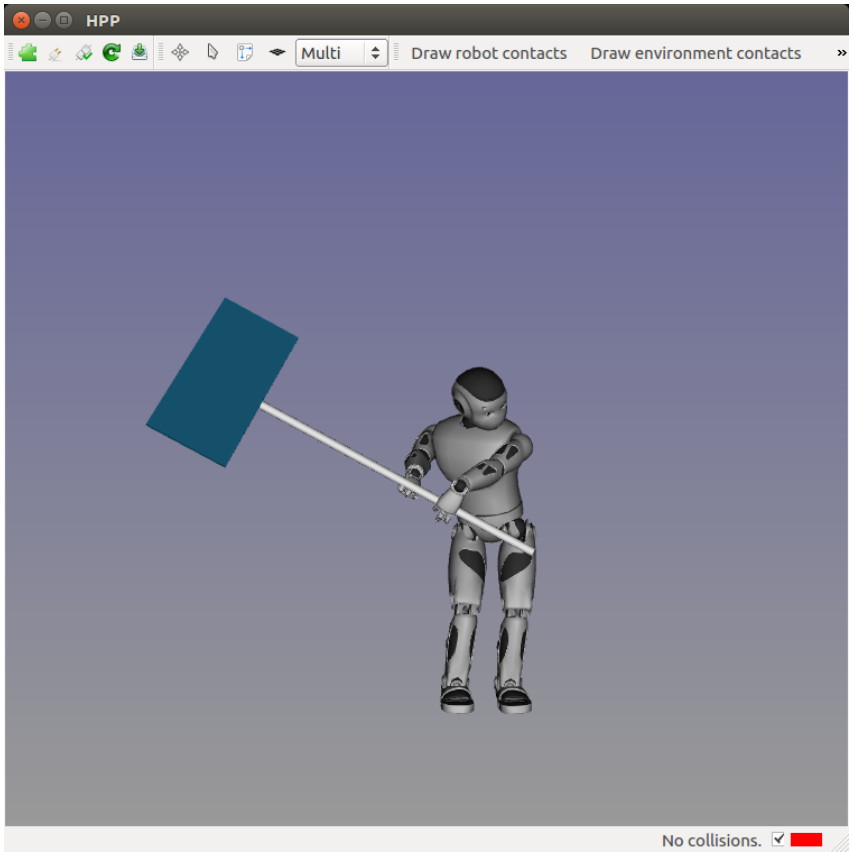
- Kinematic chain with geometry
  - `pinocchio`: implementation of kinematic chain with geometry,
    - tree of joints (Rotation, Translation, SE3: vector + unit-quaternions),
    - moving fcl::CollisionObjects,
    - forward kinematics,
    - joint Jacobians,
    - center of mass and Jacobian,
    - URDF parser.
- Numerical constraints
  - `hpp-constraints`: numerical constraints
    - implicit $f(\mathbf{q}) = (\leq)0$,
    - explicit $\mathbf{q}_{out} = f(\mathbf{q}_{in})$,
    - numerical solvers based on Newton-Raphson.

Software Development Kit

Packages implementing the core infrastructure

- Kinematic chain with geometry
  - `pinocchio`: implementation of kinematic chain with geometry,
    - tree of joints (Rotation, Translation, SE3: vector + unit-quaternions),
    - moving fcl::CollisionObjects,
    - forward kinematics,
    - joint Jacobians,
    - center of mass and Jacobian,
    - URDF parser.
- Numerical constraints
  - `hpp-constraints`: numerical constraints
    - implicit $f(\mathbf{q}) = (\leq)0$,
    - explicit $\mathbf{q}_{out} = f(\mathbf{q}_{in})$,
    - numerical solvers based on Newton-Raphson.

Newton-Raphson algorithm

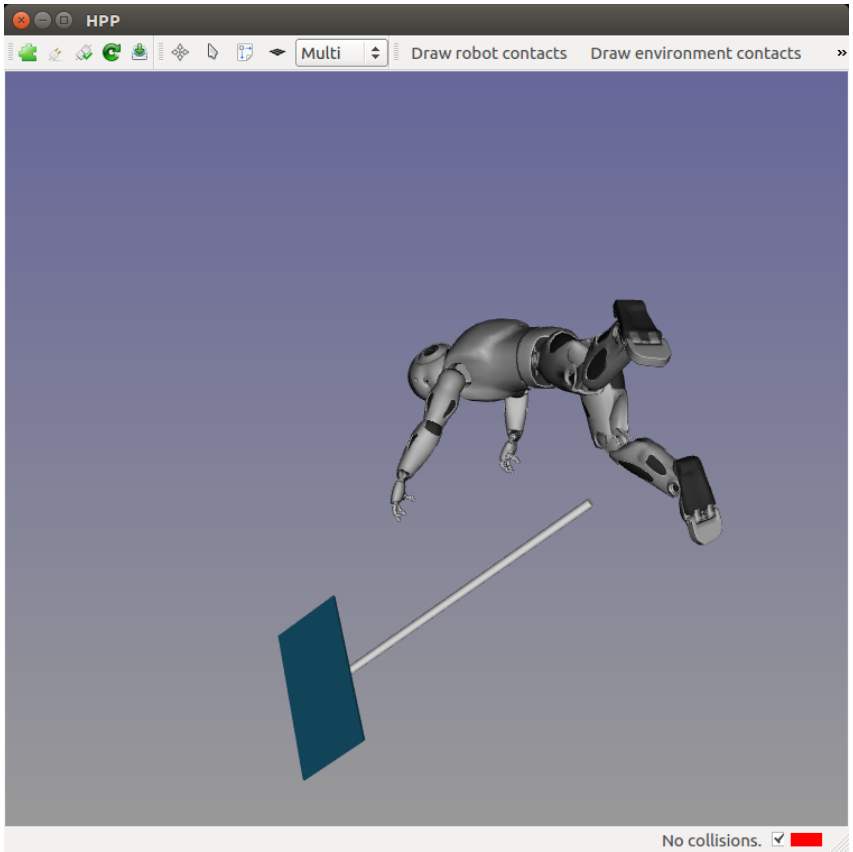


Constraints

- quasi-static equilibrium (15)
- both hands hold the placard (10)

Goal: Generate a configuration satisfying the constraints.

Newton-Raphson algorithm

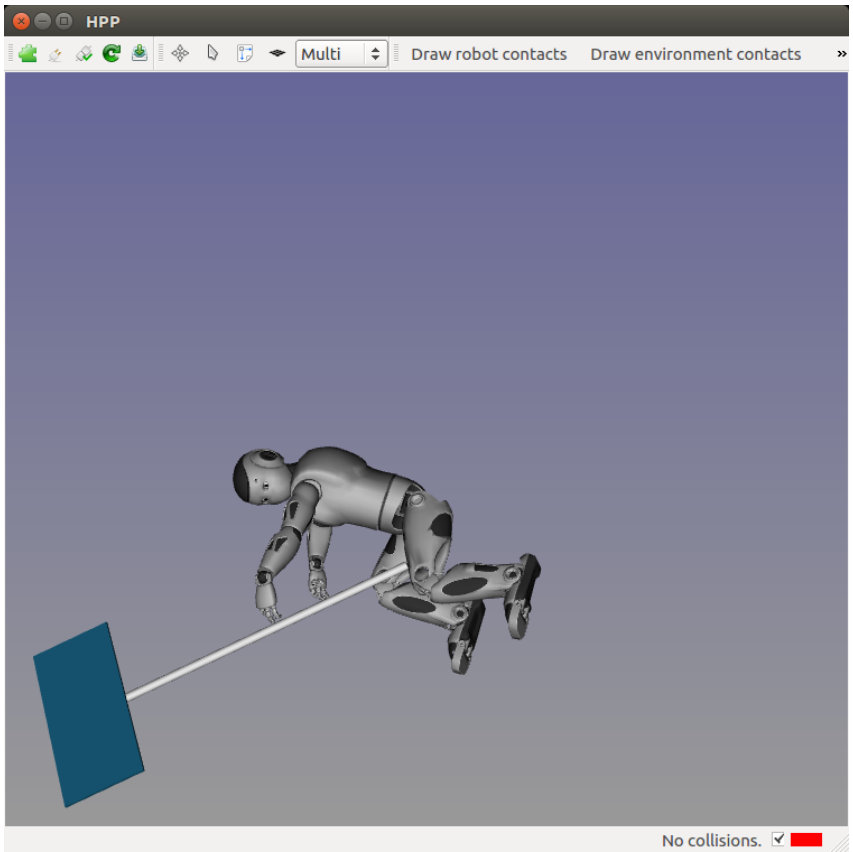


Constraints

- quasi-static equilibrium (15)
- both hands hold the placard (10)

Shoot random configuration

Newton-Raphson algorithm

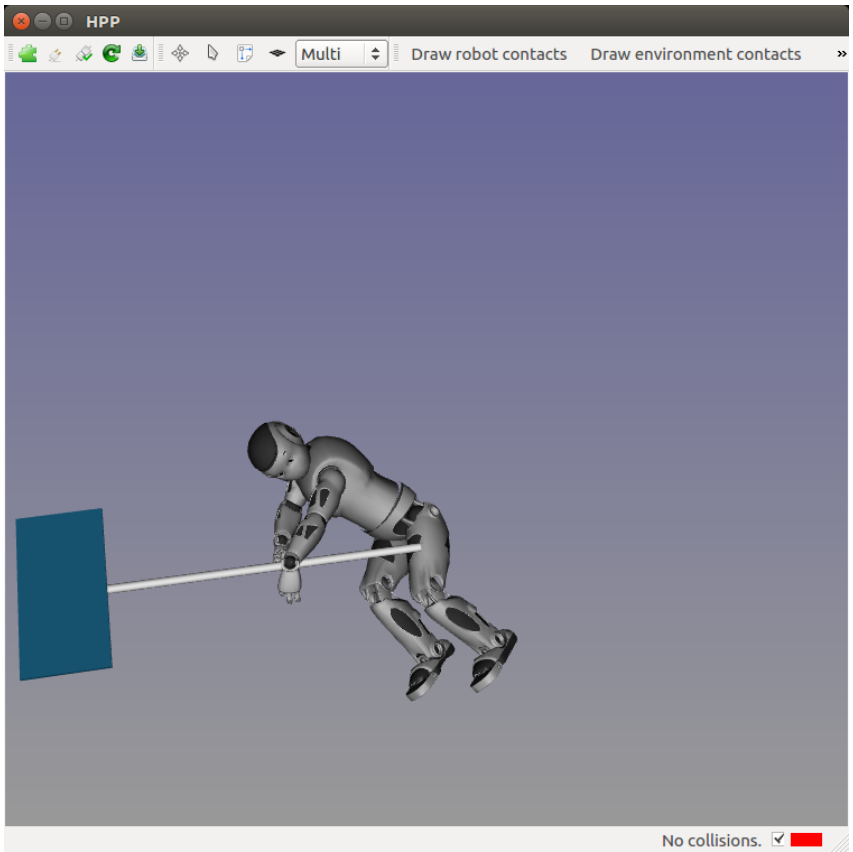


Constraints

- quasi-static equilibrium (15)
- both hands hold the placard (10)

Solve linearized system

Newton-Raphson algorithm

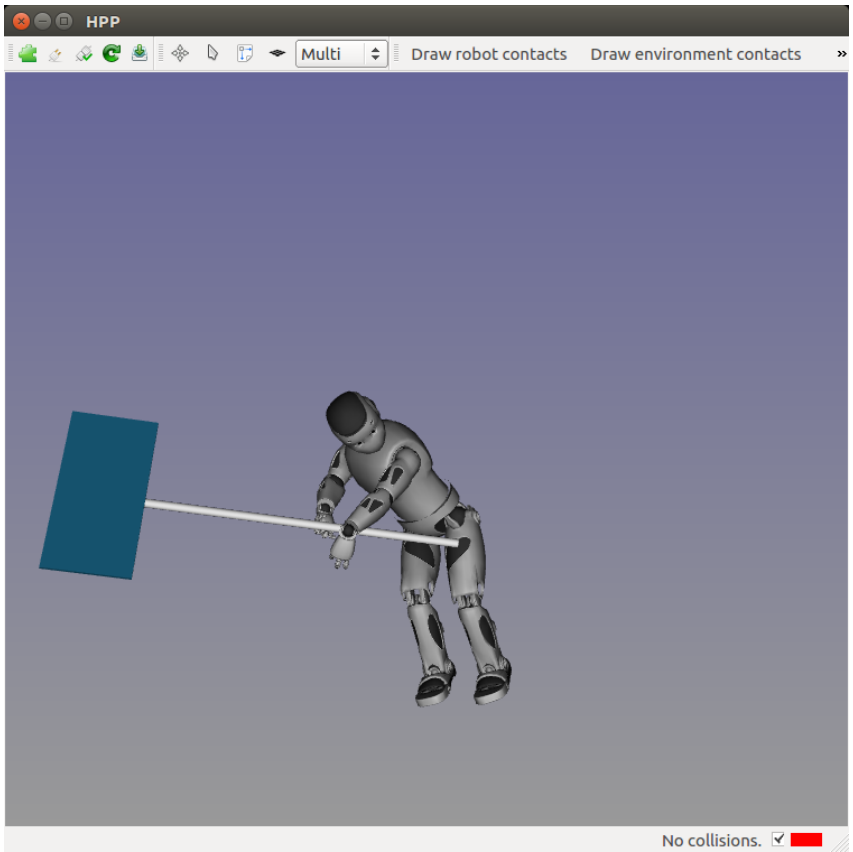


Constraints

- quasi-static equilibrium (15)
- both hands hold the placard (10)

Solve linearized system

Newton-Raphson algorithm

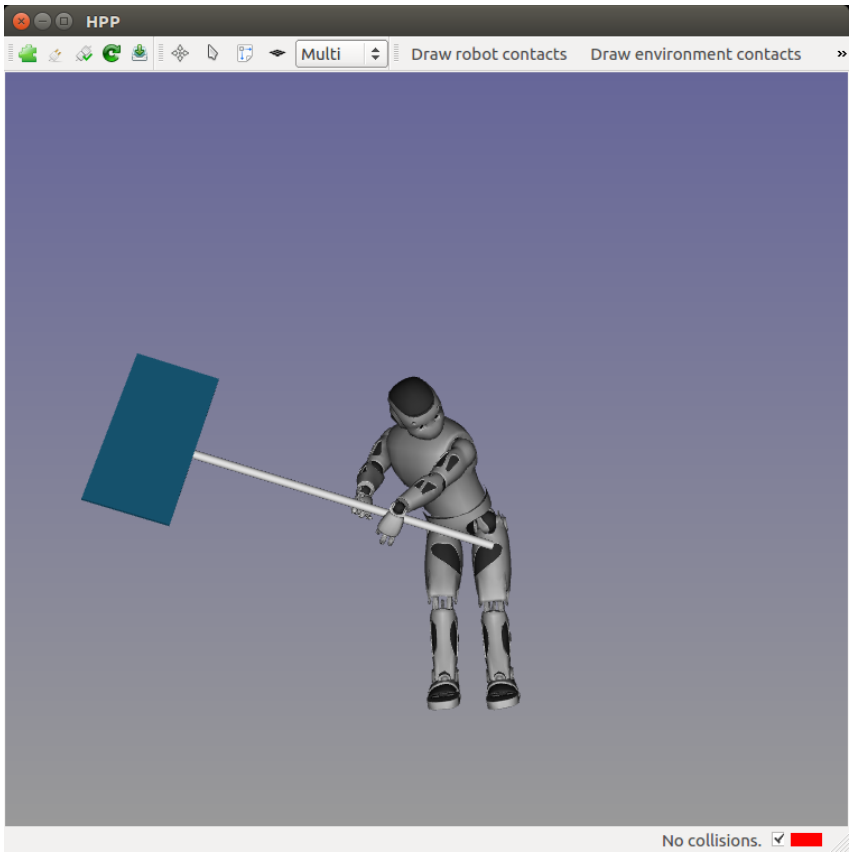


Constraints

- quasi-static equilibrium (15)
- both hands hold the placard (10)

Solve linearized system

Newton-Raphson algorithm

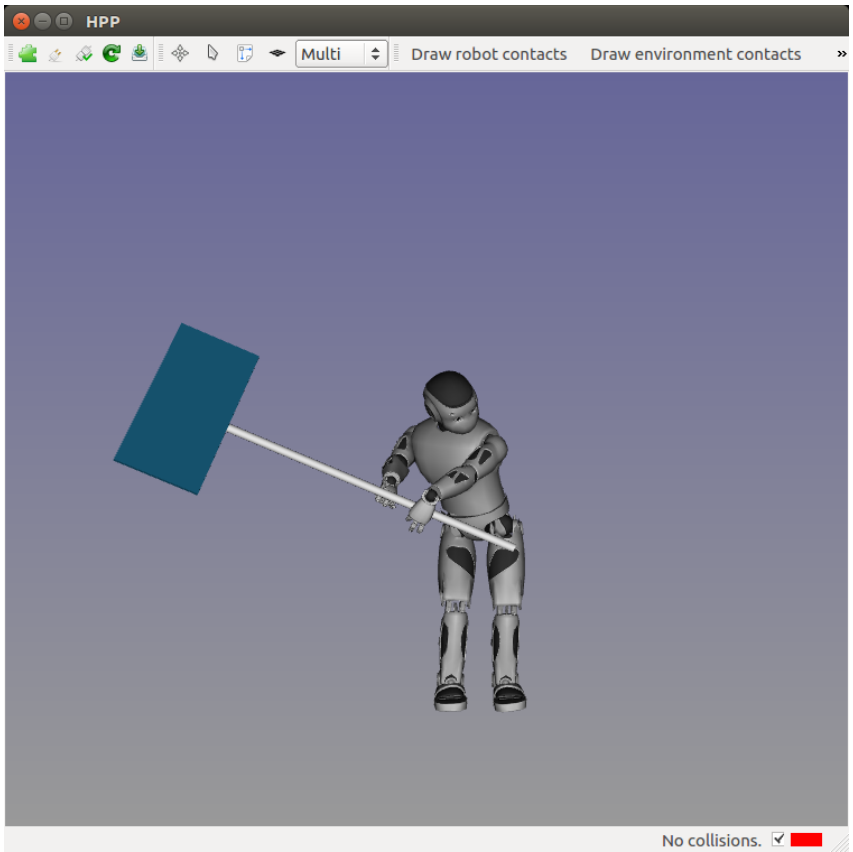


Constraints

- quasi-static equilibrium (15)
- both hands hold the placard (10)

Solve linearized system

Newton-Raphson algorithm

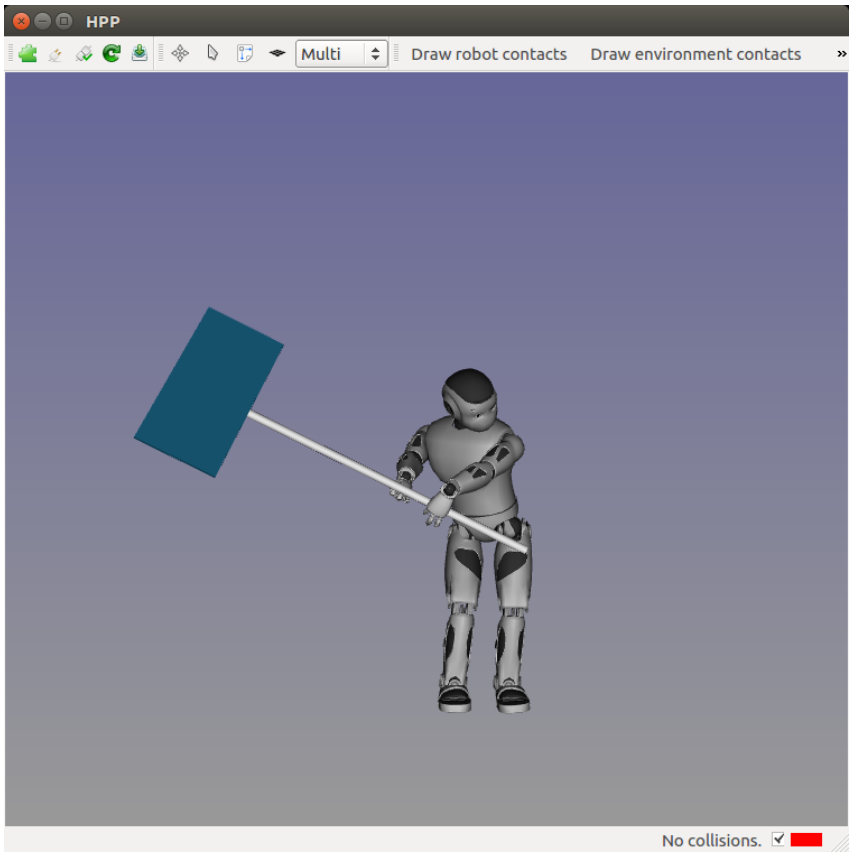


Constraints

- quasi-static equilibrium (15)
- both hands hold the placard (10)

Solve linearized system

Newton-Raphson algorithm

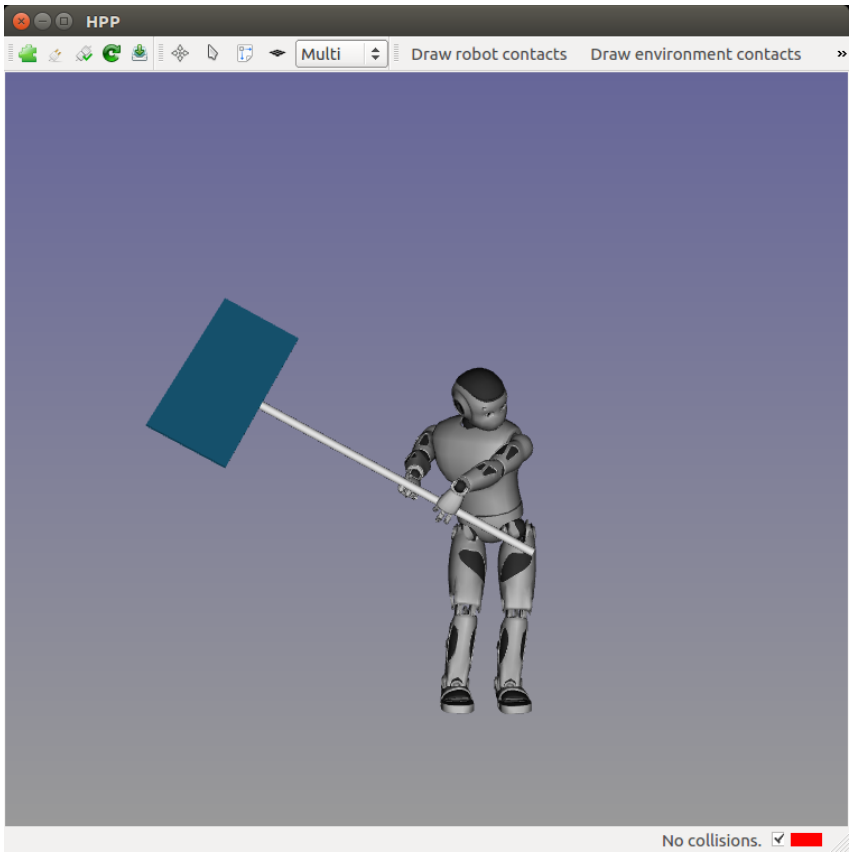


Constraints

- quasi-static equilibrium (15)
- both hands hold the placard (10)

Solve linearized system

Newton-Raphson algorithm



Constraints

- quasi-static equilibrium (15)
- both hands hold the placard (10)

Result: a configuration that satisfies the constraints.

Software Development Kit

Packages implementing the core infrastructure

- Path planning
  - `hpp-core`: definition of basic classes,
    - path planning problems,
    - path planning solvers (RRT),
    - constraints (locked dofs, numerical constraints)
    - path optimizers (random shortcut),
    - steering methods (straight interpolation)

Extensions

Packages implementing other algorithms

- `hpp-wholebody-step`: whole-body and walk planning using sliding path approximation,
- `hpp-manipulation`: manipulation planning (see next section),
- any extension for a given application.

Extensions

Packages implementing other algorithms

- `hpp-wholebody-step`: whole-body and walk planning using sliding path approximation,
- `hpp-manipulation`: manipulation planning (see next section),
- any extension for a given application.

Extensions

Packages implementing other algorithms

- `hpp-wholebody-step`: whole-body and walk planning using sliding path approximation,
- `hpp-manipulation`: manipulation planning (see next section),
- any extension for a given application.

Python control

`hpp-corbaserver`: python scripting through CORBA

- embed `hpp-core` into a CORBA server and expose services through 3 `idl` interfaces:
  - `Robot` load and initializes robot,
  - `Obstacle` load and build obstacles,
  - `Problem` define and solve problem.
- Implement python classes to help user call CORBA services
  - `Robot` automatize robot loading,
  - `ProblemSolver` definition problem helper.

Python control

`hpp-corbaserver`: python scripting through CORBA

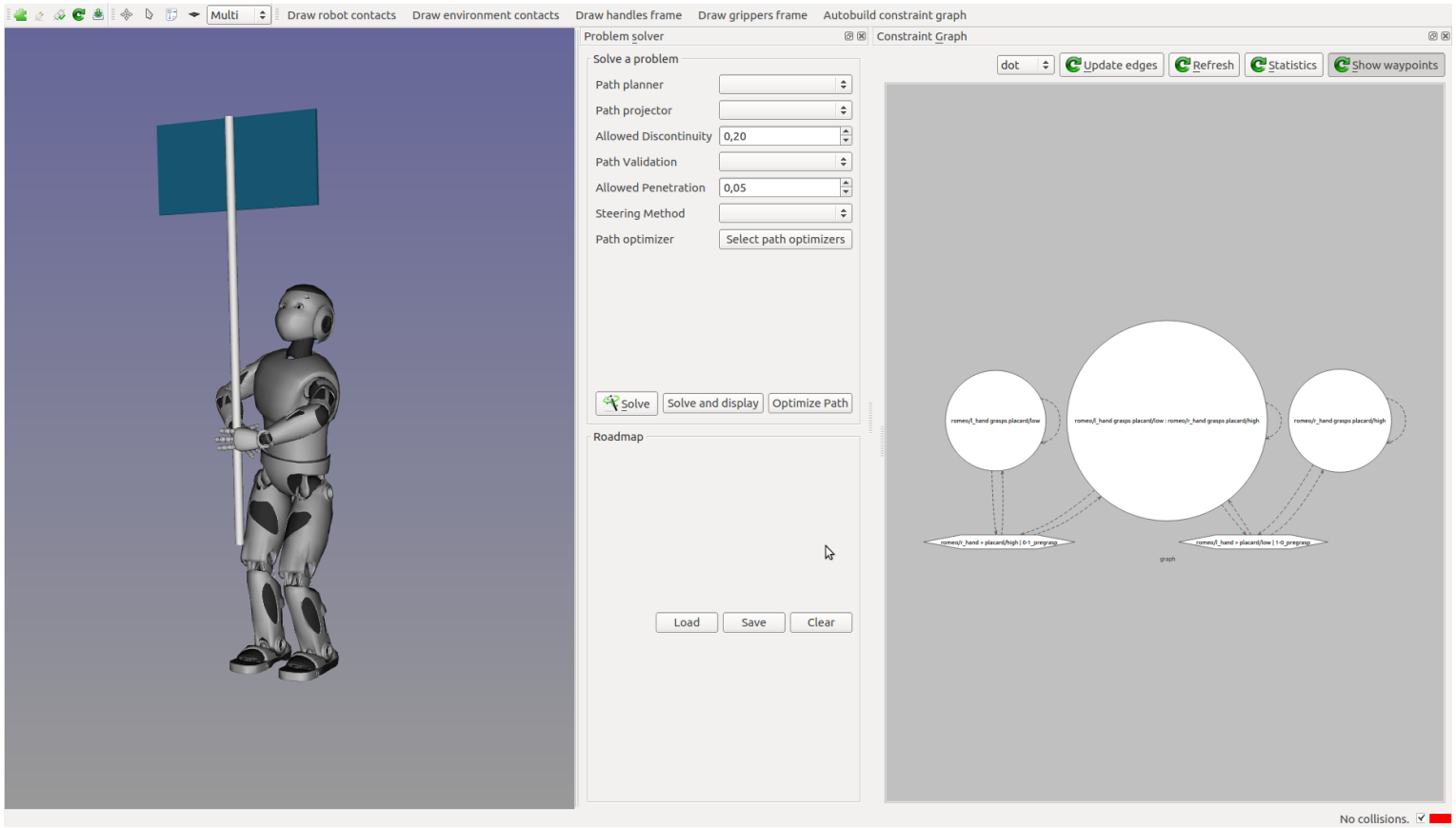
- embed `hpp-core` into a CORBA server and expose services through 3 `idl` interfaces:
  - `Robot` load and initializes robot,
  - `Obstacle` load and build obstacles,
  - `Problem` define and solve problem.
- Implement python classes to help user call CORBA services
  - `Robot` automatize robot loading,
  - `ProblemSolver` definition problem helper.

Python control

Extensions through plugins in `hpp-corbaserver`

- `hpp-manipulation-corba`: control of manipulation planning specific classes and algorithms.

Visualization through gepetto-gui



Implemented by package `hpp-gepetto-viewer`.

Outline

Manipulation

Class of problem containing:

- A robot: actuated DOFs
- Objects: unactuated DOFs

A solution will be a succession of motion of two types:

- The robot moves without constraints. Objects do not move.
- The robot moves while grasping the object.

Manipulation

Class of problem containing:

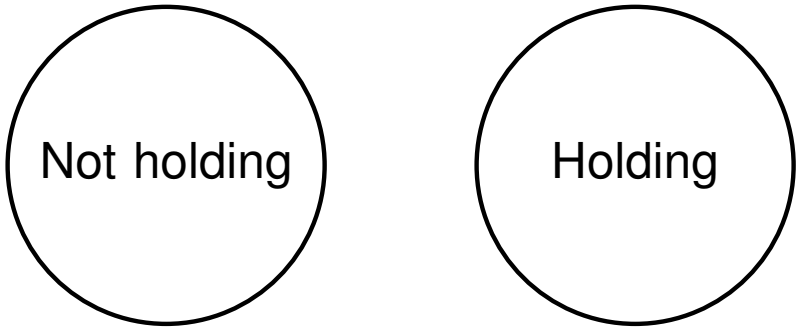
- A robot: actuated DOFs
- Objects: unactuated DOFs

A solution will be a succession of motion of two types:

- The robot moves without constraints. Objects do not move.
- The robot moves while grasping the object.

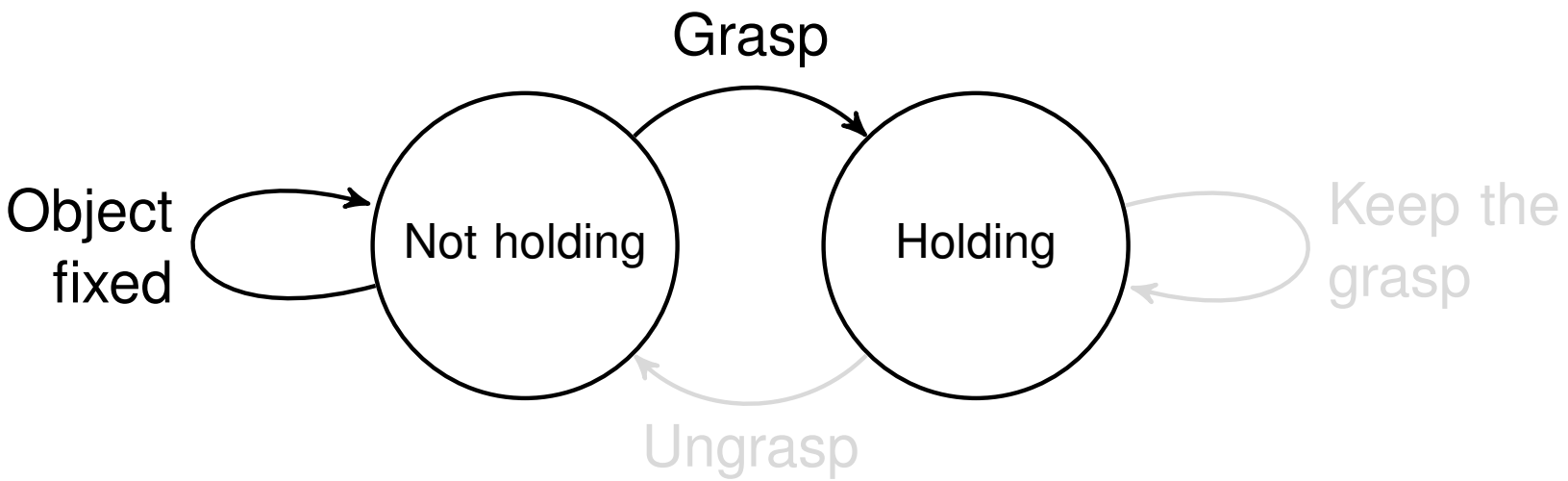
Manipulation

2 states:



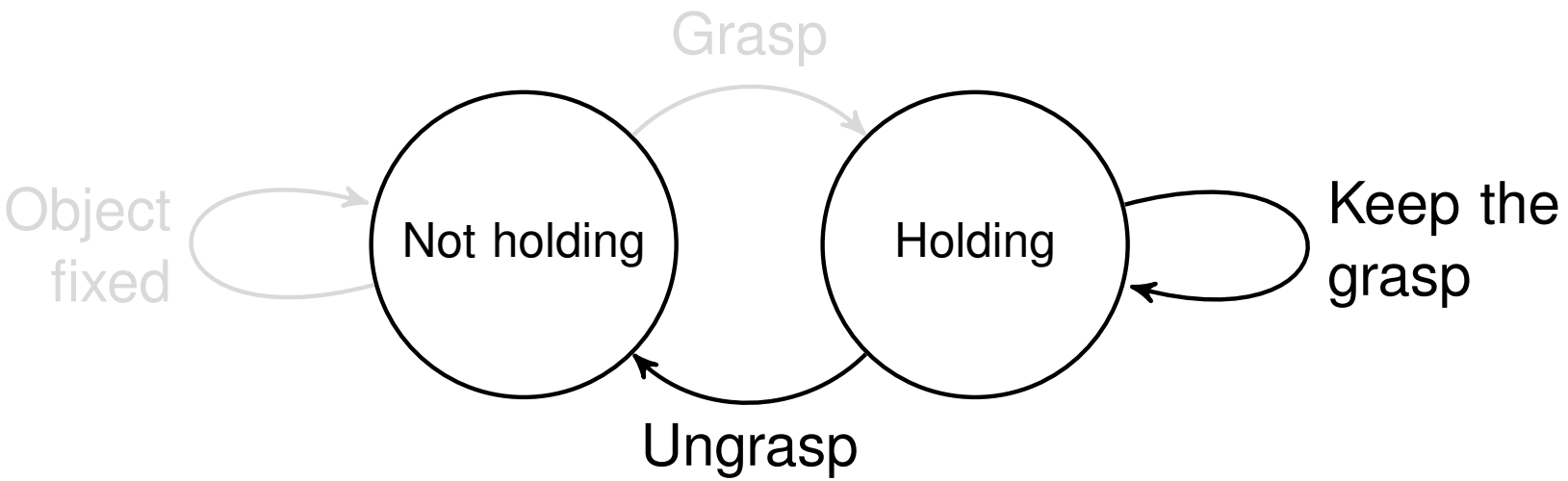
Manipulation

4 transitions:



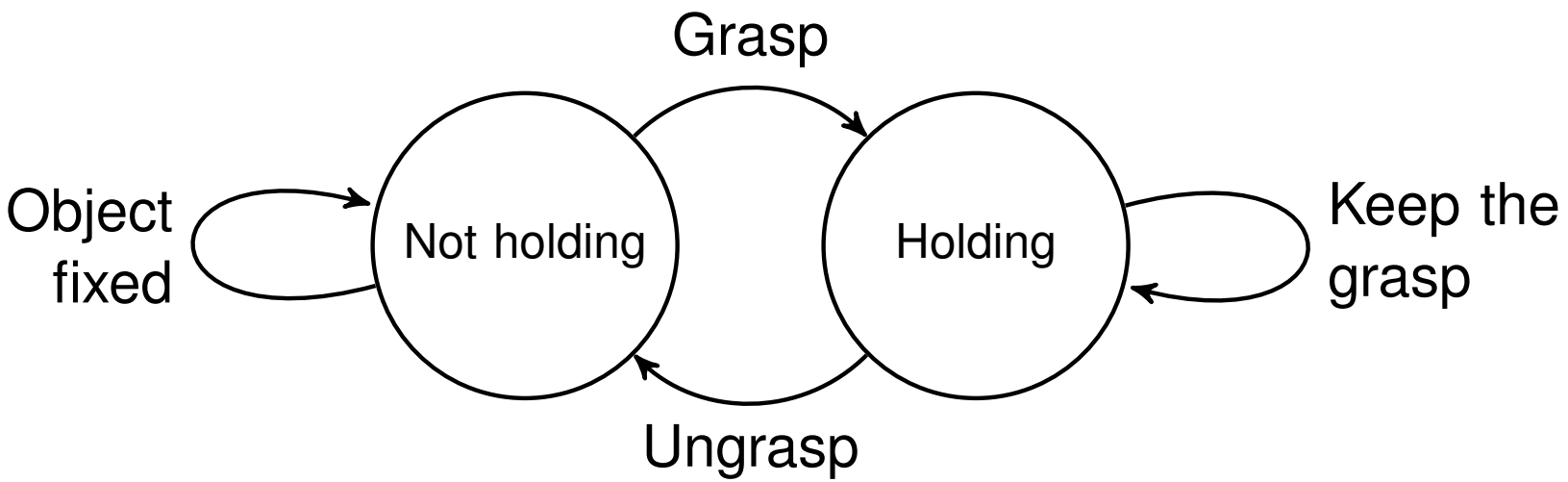
Manipulation

4 transitions:



Manipulation

4 transitions:



Constraint

Definition

A function $f \in D^1(\mathcal{C}, \mathbb{R}^m)$.

Foliation

A leaf of a constraint f is defined by:

$$L_{f_0}(f) = \{\mathbf{q} \in \mathcal{C} | f(\mathbf{q}) = f_0\}$$

where f_0 is called the *right hand side* of the constraint.

Projection

Using a Newton Descent algorithm:

$$\mathbf{q}_{rand} | f(\mathbf{q}_{rand}) \neq f_0 \Rightarrow \mathbf{q}_{proj} | f(\mathbf{q}_{proj}) = f_0$$

Constraint

Definition

A function $f \in D^1(\mathcal{C}, \mathbb{R}^m)$.

Foliation

A leaf of a constraint f is defined by:

$$L_{f_0}(f) = \{\mathbf{q} \in \mathcal{C} | f(\mathbf{q}) = f_0\}$$

where f_0 is called the *right hand side* of the constraint.

Projection

Using a Newton Descent algorithm:

$$\mathbf{q}_{rand} | f(\mathbf{q}_{rand}) \neq f_0 \Rightarrow \mathbf{q}_{proj} | f(\mathbf{q}_{proj}) = f_0$$

Constraint

Definition

A function $f \in D^1(\mathcal{C}, \mathbb{R}^m)$.

Foliation

A leaf of a constraint f is defined by:

$$L_{f_0}(f) = \{\mathbf{q} \in \mathcal{C} | f(\mathbf{q}) = f_0\}$$

where f_0 is called the *right hand side* of the constraint.

Projection

Using a Newton Descent algorithm:

$$\mathbf{q}_{rand} | f(\mathbf{q}_{rand}) \neq f_0 \Rightarrow \mathbf{q}_{proj} | f(\mathbf{q}_{proj}) = f_0$$

Constraint

Two types of constraints:

Configuration

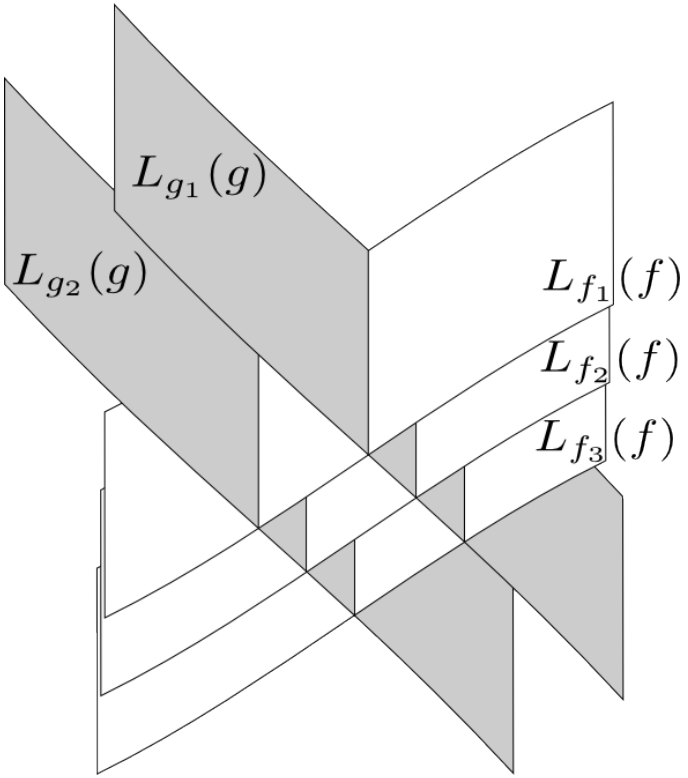
Only one leaf is interesting: $L_0(f)$.

Motion

A leaf also represents reachability space.

Foliation

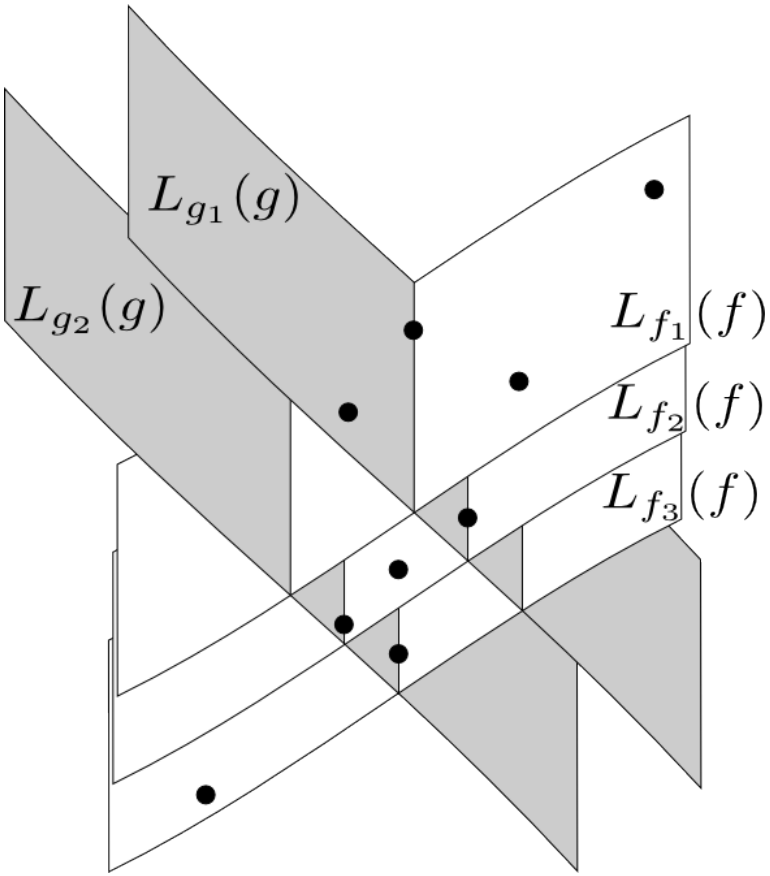
In the configuration space:

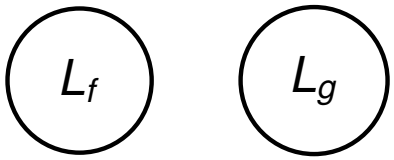


2 constraints on motion

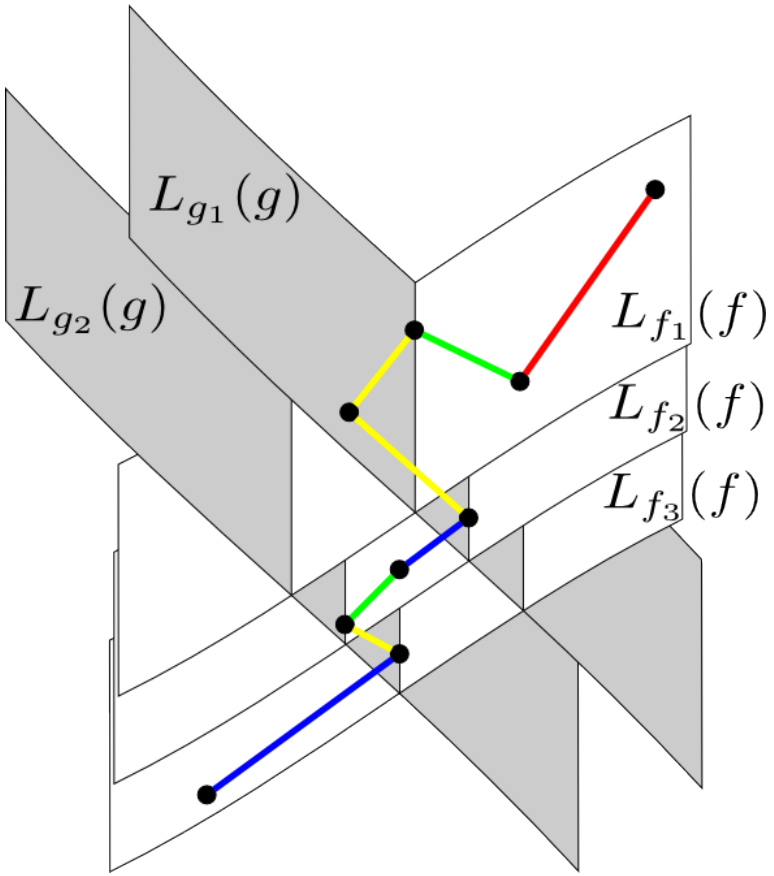
- f: position of the object.
- g: grasp of the object.

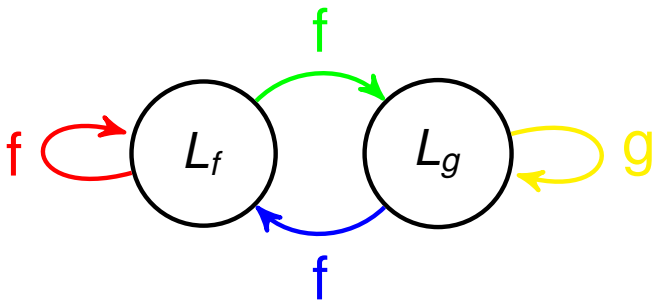
Constraint graph



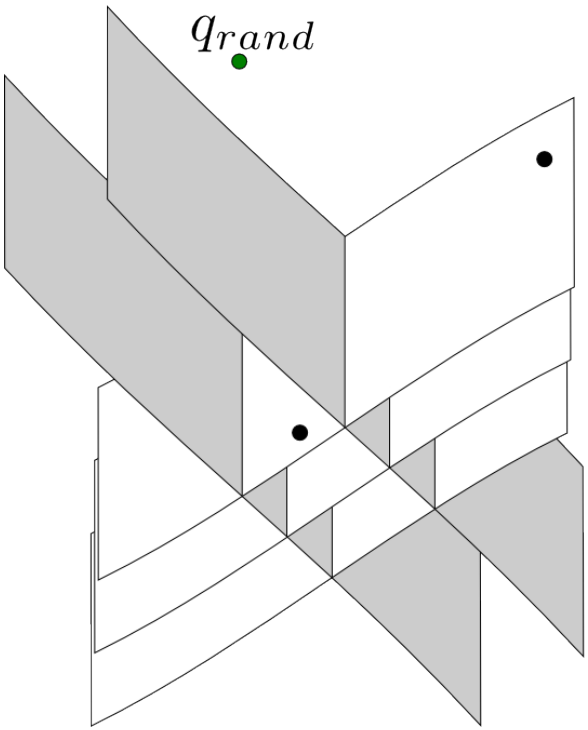


Constraint graph





Rapidly exploring Random Tree



$\mathbf{q}_{rand}$ = shoot_random_config()

$\mathbf{q}_{near}$ = nearest_neighbor($\mathbf{q}_{rand}$, *tree*)

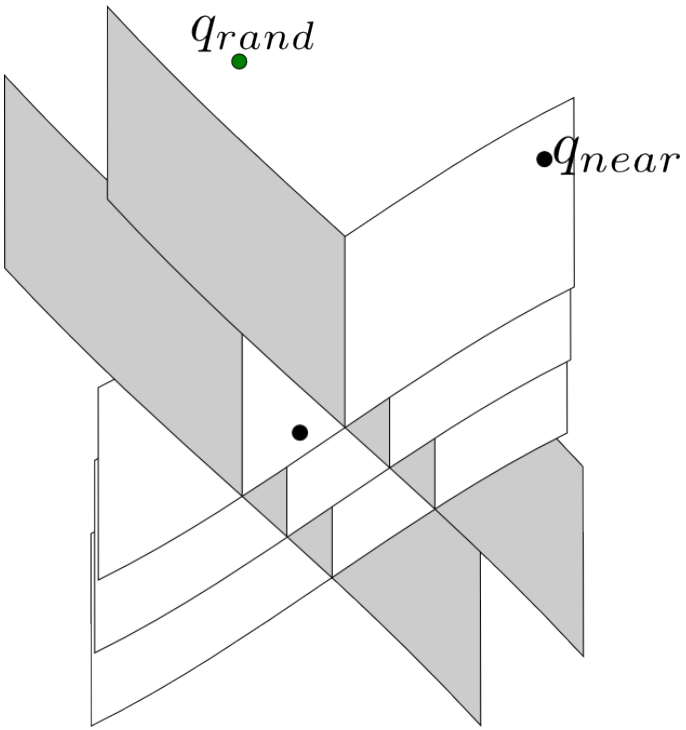
f_e, f_p = select_next_state($\mathbf{q}_{near}$)

$\mathbf{q}_{proj}$ = project($\mathbf{q}_{rand}$, f_e)

$\mathbf{q}_{new}$ = extend($\mathbf{q}_{near}$, $\mathbf{q}_{proj}$, f_p)

tree.insert_node(($\mathbf{q}_{near}$, $\mathbf{q}_{new}$, f_p))

Rapidly exploring Random Tree



$\mathbf{q}_{rand}$ = shoot_random_config()

$\mathbf{q}_{near}$ = nearest_neighbor($\mathbf{q}_{rand}$, *tree*)

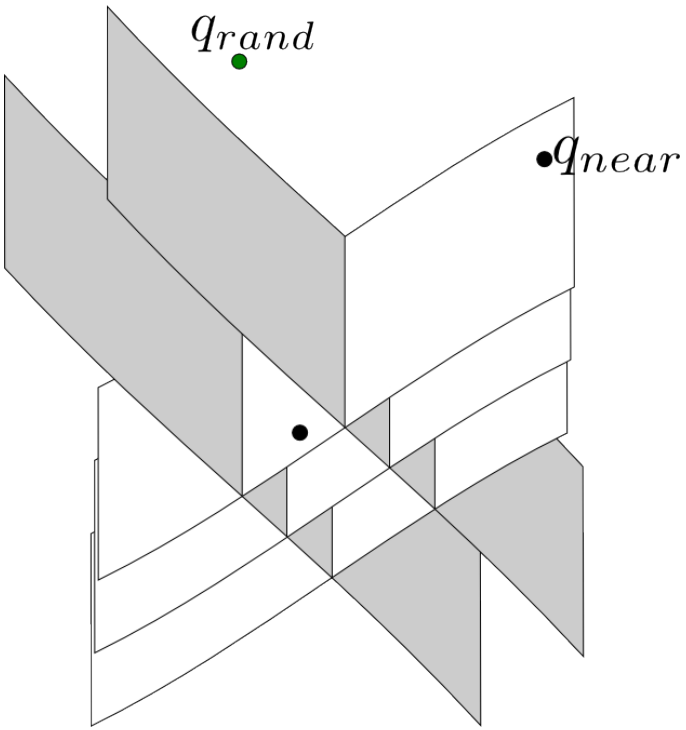
f_e, f_p = select_next_state($\mathbf{q}_{near}$)

$\mathbf{q}_{proj}$ = project($\mathbf{q}_{rand}$, f_e)

$\mathbf{q}_{new}$ = extend($\mathbf{q}_{near}$, $\mathbf{q}_{proj}$, f_p)

tree.insert_node(($\mathbf{q}_{near}$, $\mathbf{q}_{new}$, f_p))

Rapidly exploring Random Tree



$\mathbf{q}_{rand}$ = shoot_random_config()

$\mathbf{q}_{near}$ = nearest_neighbor($\mathbf{q}_{rand}$, *tree*)

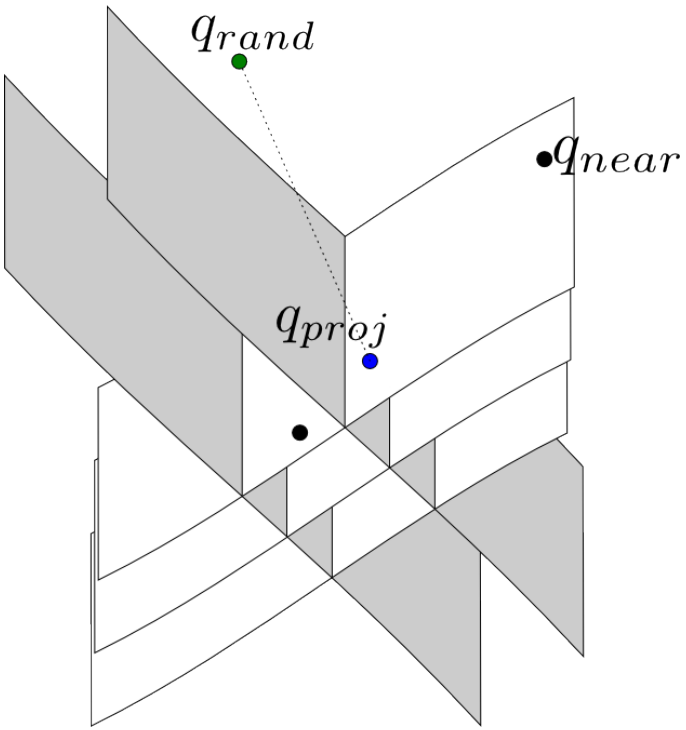
f_e, f_p = select_next_state($\mathbf{q}_{near}$)

$\mathbf{q}_{proj}$ = project($\mathbf{q}_{rand}$, f_e)

$\mathbf{q}_{new}$ = extend($\mathbf{q}_{near}$, $\mathbf{q}_{proj}$, f_p)

tree.insert_node(($\mathbf{q}_{near}$, $\mathbf{q}_{new}$, f_p))

Rapidly exploring Random Tree



$\mathbf{q}_{rand}$ = shoot_random_config()

$\mathbf{q}_{near}$ = nearest_neighbor($\mathbf{q}_{rand}$, *tree*)

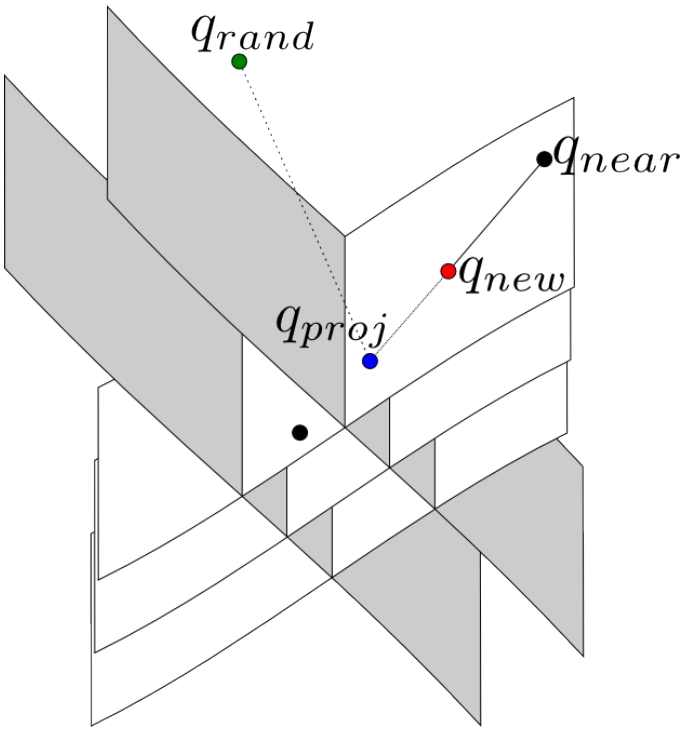
f_e, f_p = select_next_state($\mathbf{q}_{near}$)

$\mathbf{q}_{proj}$ = project($\mathbf{q}_{rand}$, f_e)

$\mathbf{q}_{new}$ = extend($\mathbf{q}_{near}$, $\mathbf{q}_{proj}$, f_p)

tree.insert_node(($\mathbf{q}_{near}$, $\mathbf{q}_{new}$, f_p))

Rapidly exploring Random Tree



$\mathbf{q}_{rand}$ = shoot_random_config()

$\mathbf{q}_{near}$ = nearest_neighbor($\mathbf{q}_{rand}$, *tree*)

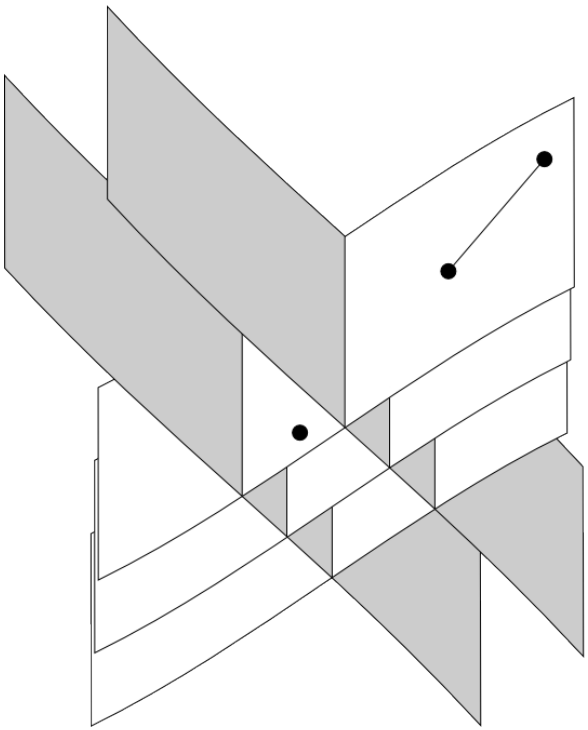
f_e, f_p = select_next_state($\mathbf{q}_{near}$)

$\mathbf{q}_{proj}$ = project($\mathbf{q}_{rand}$, f_e)

$\mathbf{q}_{new}$ = extend($\mathbf{q}_{near}$, $\mathbf{q}_{proj}$, f_p)

tree.insert_node(($\mathbf{q}_{near}$, $\mathbf{q}_{new}$, f_p))

Rapidly exploring Random Tree



$\mathbf{q}_{rand}$ = shoot_random_config()

$\mathbf{q}_{near}$ = nearest_neighbor($\mathbf{q}_{rand}$, *tree*)

f_e, f_p = select_next_state($\mathbf{q}_{near}$)

$\mathbf{q}_{proj}$ = project($\mathbf{q}_{rand}$, f_e)

$\mathbf{q}_{new}$ = extend($\mathbf{q}_{near}$, $\mathbf{q}_{proj}$, f_p)

tree.insert_node(($\mathbf{q}_{near}$, $\mathbf{q}_{new}$, f_p))

hpp-manipulation-corba

Provides tools to:

- read URDF files of robots and objects;
- create grasp contraints between a end-effector (robot) and a handle (object);
- build the graph of constraints;

hpp-manipulation-corba

Provides tools to:

- read URDF files of robots and objects;
- create grasp contraints between a end-effector (robot) and a handle (object);
- build the graph of constraints;

hpp-manipulation-corba

Provides tools to:

- read URDF files of robots and objects;
- create grasp contraints between a end-effector (robot) and a handle (object);
- build the graph of constraints;

Installation and documentation

Everything in `https://humanoid-path-planner.github.io/hpp-doc`

Keep informed

- Mailing list `hpp@laas.fr` to discuss issues related to the software,
- github notifications for issues related to individual packages