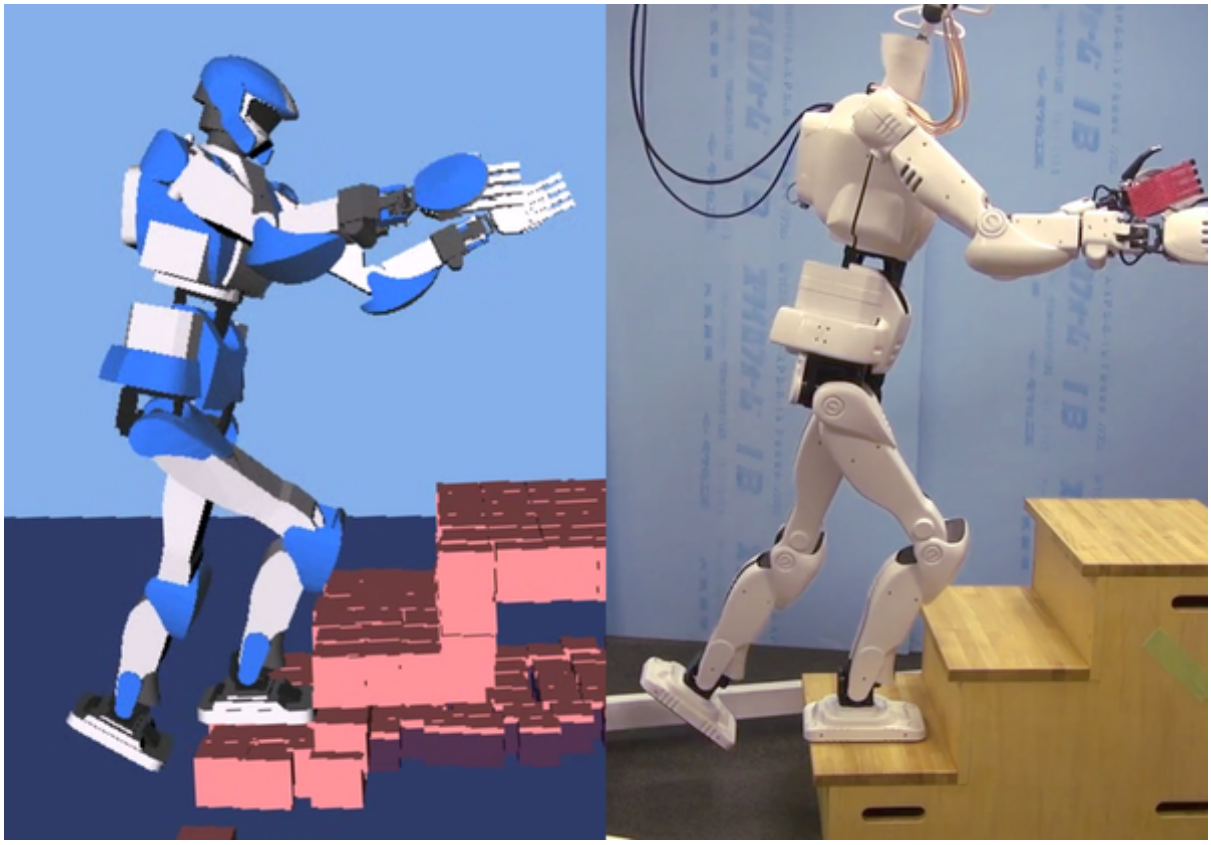
pymanoid Documentation

Release 1.2.0

Stéphane Caron

Jun 16, 2022

Contents



1 Examples

1.1 Linear inverted pendulum stabilization

This example implements a basic stabilizer for the linear inverted pendulum model based on proportional feedback of the divergent component of motion.

class lip_stabilization.**Pusher**(*pendulum*, *gain=0.1*)

Send impulses to the inverted pendulum every once in a while.

Parameters

- **pendulum** (*pymanoid.models.InvertedPendulum*) – Inverted pendulum to de-stabilize.
- **gain** (*scalar*) – Magnitude of velocity jumps.

Notes

You know, I've seen a lot of people walkin' 'round // With tombstones in their eyes // But the pusher don't care // Ah, if you live or if you die

on_tick(*sim*)

Apply regular impulses to the inverted pendulum.

Parameters **sim** (*pymanoid.Simulation*) – Simulation instance.

class lip_stabilization.**Stabilizer**(*pendulum*, *gain=2.0*)

Inverted pendulum stabilizer based on proportional DCM feedback.

Parameters

- **pendulum** (*pymanoid.models.InvertedPendulum*) – Inverted pendulum to stabilizer.
- **gain** (*scalar*) – DCM feedback gain.

draw(*dcm*, *cop*)

Draw extra points to illustrate stabilizer behavior.

Parameters

- **dcm** (*(3,) array*) – Divergent component of motion.
- **cop** (*(3,) array*) – Center of pressure.

on_tick(*sim*)

Set inverted pendulum CoP and stiffness inputs.

Parameters **sim** (*pymanoid.Simulation*) – Simulation instance.

Notes

See [Englsberger15] for details on the definition of the virtual repellent point (VRP). Here we differentiate between the constants lambda and omega: lambda corresponds to the "CoP-based inverted pendulum" and omega to the "floating-base inverted pendulum" models described in Section II.B of [Caron17w].

Overall, we can interpret `omega` as a normalized stiffness between the CoM and VRP, while `lambda` corresponds to the virtual leg stiffness between the CoM and ZMP. The reason why the mapping is nonlinear is that the ZMP is constrained to lie on the contact surface, while the CoM can move in 3D.

If we study further the relationship between lambda and omega, we find that they are actually related by a Riccati equation [Caron19].

1.2 Inverse kinematics

This example uses inverse kinematics (IK) to achieve a set of whole-body tasks. It contains two equivalent implementations of the IK solver setup. The former is best for beginners as it uses the simpler `Stance` interface. The latter is for more advanced users and shows how to add individual tasks one by one.

The example loads the JVRC-1 humanoid model, then generates a posture where the robot has both feet on pre-defined contact locations. The robot tracks a reference COM position given by the red box, which you can move around directly by using the interaction mode of the OpenRAVE GUI.

`inverse_kinematics.`**`setup_ik_from_stance`**`()`
Setup inverse kinematics from the simpler stance interface.

Notes

This function is equivalent to *`setup_ik_from_tasks()`* below.

`inverse_kinematics.`**`setup_ik_from_tasks`**`()`
Setup the inverse kinematics task by task.

Note: This function is equivalent to *`setup_ik_from_stance()`* above. Beginners should take a look at that one first.

Notes

See this tutorial on inverse kinematics for details.

1.3 Contact stability

Wrench friction cone

This example shows how to compute the contact wrench cone (CWC), a generalized multi-contact friction cone. See [Caron15] for details.

CoM static-equilibrium polygon

This example computes the static-equilibrium CoM polygon, i.e. the set of CoM positions that can be sustained usingf a given set of contacts. See [Caron16] for details.

class com_static_polygon.**COMSync**(*stance*, *com_above*)

Update stance CoM from the GUI handle in polygon above the robot.

Parameters

- **stance** (*pymanoid.Stance*) – Contacts and COM position of the robot.
- **com_above** (*pymanoid.Cube*) – CoM handle in static-equilibrium polygon.

on_tick(*sim*)

Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

class com_static_polygon.**SupportPolygonDrawer**(*stance*, *method*, *z_polygon*, *color='g'*)

Draw the static-equilibrium polygon of a contact set.

Parameters

- **stance** (*pymanoid.Stance*) – Contacts and COM position of the robot.
- **method** (*string*) – Method to compute the static equilibrium polygon. Choices are: 'bretl', 'cdd' and 'hull'.
- **z_polygon** (*scalar*) – Height where to draw the CoM static-equilibrium polygon.
- **color** (*tuple or string, optional*) – Area color.

on_tick(*sim*)

Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

Multi-contact ZMP support areas

This example computes the multi-contact ZMP support area for a given robot stance (contacts and CoM position). See [Caron16] for details.

class zmp_support_area.**COMSync**(*stance*, *com_above*)

on_tick(*sim*)
Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

class zmp_support_area.**StaticWrenchDrawer**(*stance*)
Draw contact wrenches applied to a robot in static-equilibrium.

Parameters **stance** (Stance) – Contacts and COM position of the robot.

class zmp_support_area.**SupportAreaDrawer**(*stance*, *height=0.0*, *color=None*)
Draw the pendular ZMP area of a contact set.

Parameters

- **stance** (Stance) – Contacts and COM position of the robot.
- **height** (*scalar, optional*) – Height of the ZMP support area in the world frame.
- **color** (*tuple or string, optional*) – Area color.

on_tick(*sim*)
Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

CoM acceleration cone

This example shows the set of 3D CoM accelerations that a given set of contacts can realize in the pendulum mode of motion (i.e. no angular momentum). See [Caron16] for details.

class com_accel_cone.**AccelConeDrawer**(*stance*, *scale=0.1*, *color=None*)
Draw the COM acceleration cone of a contact set.

Parameters

- **stance** (Stance) – Contacts and COM position of the robot.
- **scale** (*scalar, optional*) – Acceleration to distance conversion ratio, in [s]^2.
- **color** (*tuple or string, optional*) – Area color.

on_tick(*sim*)
Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

class com_accel_cone.**COMSync**(*stance*, *com_above*)

on_tick(*sim*)
Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

class com_accel_cone.**StaticWrenchDrawer**(*stance*)
Draw contact wrenches applied to a robot in static-equilibrium.

Parameters **stance** (Stance) – Contacts and COM position of the robot.

CoM robust static-equilibrium polytope

This example computes the robust static-equilibrium CoM polyhedron. See [Audren18] for details. Running this example requires the StabiliPy library.

class com_robust_static_polytope.**StaticWrenchDrawer**(*stance*)
Draw contact wrenches applied to a robot in static-equilibrium.

Parameters **stance** (Stance) – Contacts and COM position of the robot.

class com_robust_static_polytope.**SupportPolyhedronDrawer**(*stance*, *z=0.0*, *color=None*, *method='qhull'*)
Draw the robust static-equilibrium polyhedron of a contact set.

Parameters

- **stance** (Stance) – Contacts and COM position of the robot.
- **color** (*tuple or string, optional*) – Area color.
- **method** (*string, optional*) – Method to compute the static equilibrium polygon. Choices are cdd, qhull (default) and parma

on_tick(*sim*)
Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

1.4 Horizontal walking

This example implements a walking pattern generator for horizontal walking based on linear model predictive control <https://hal.inria.fr/file/index/docid/390462/filename/Preview.pdf>.

class horizontal_walking.**HorizontalWalkingFSM**(*ssp_duration*, *dsp_duration*)

Finite State Machine for biped walking.

Parameters

- **ssp_duration** (*scalar*) – Duration of single-support phases, in [s].
- **dsp_duration** (*scalar*) – Duration of double-support phases, in [s].

on_tick(*sim*)

Update function run at every simulation tick.

Parameters **sim** (Simulation) – Instance of the current simulation.

run_com_mpc()

Run CoM predictive control for single-support state.

run_double_support()

Run double-support state.

run_single_support()

Run single-support state.

run_standing()

Run standing state.

run_swing_foot()

Run swing foot interpolator for single-support state.

start_double_support()

Switch to double-support state.

start_single_support()

Switch to single-support state.

start_standing()

Switch to standing state.

start_swing_foot()

Initialize swing foot interpolator for single-support state.

horizontal_walking.**generate_footsteps**(*distance*, *step_length*, *foot_spread*, *friction*)

Generate a set of footsteps for walking forward.

Parameters

- **distance** (*scalar*) – Total distance to walk forward in [m].

- **step_length** (*scalar*) – Distance between right and left heel in double support.
- **foot_spread** (*scalar*) – Lateral distance between left and right foot centers.
- **friction** (*scalar*) – Friction coefficient between a robot foot and a step.

1.5 Multi-contact walking

This example implements a walking pattern generator for multi-contact locomotion based on predictive control of the 3D acceleration of the center of mass [Caron16].

class multi_contact_walking.**COMTube**(*start_com*, *target_com*, *start_stance*, *next_stance*, *radius*, *margin=0.01*)

Primal tube of COM locations, computed with its dual acceleration cone.

Parameters

- **start_com** (*array*) – Start position of the COM.
- **target_com** (*array*) – End position of the COM.
- **start_stance** (Stance) – Stance used to compute the contact wrench cone.
- **radius** (*scalar*) – Side of the cross-section square (for shape > 2).
- **margin** (*scalar*) – Safety margin (in [m]) around boundary COM positions.

Notes

When there is an SS-to-DS contact switch, this strategy computes one primal tube and two dual intersection cones. The primal tube is a parallelepiped containing both the COM current and target locations. Its dual cone is used during the DS phase. The dual cone for the SS phase is calculated by intersecting the latter with the dual cone of the current COM position in single-contact.

compute_dual_hrep()
Compute halfspaces of the dual cones.

compute_dual_vrep()
Compute vertices of the dual cones.

compute_primal_hrep()
Compute halfspaces of the primal tube.

compute_primal_vrep()
Compute vertices of the primal tube.

class multi_contact_walking.**COMTubePredictiveControl**(*com*, *fsm*, *preview_buffer*, *nb_mpc_steps*, *tube_radius*)

Feedback controller that continuously runs the preview controller and sends outputs to a COMAccelBuffer.

Parameters

- **com** (PointMass) – Current state (position and velocity) of the COM.
- **fsm** (MultiContactWalkingFSM) – Instance of finite state machine.
- **preview_buffer** (PreviewBuffer) – MPC outputs are sent to this buffer.
- **nb_mpc_steps** (*int*) – Discretization step of the preview window.
- **tube_radius** (*scalar*) – Tube radius in [m] for the L1 norm.

compute_preview_control(*switch_time*, *horizon*, *state_constraints=False*)

Compute controller and store it in `self.preview_control`.

compute_preview_tube()

Compute preview tube and store it in `self.tube`.

on_tick(*sim*)

Entry point called at each simulation tick.

Parameters **sim** (Simulation) – Instance of the current simulation.

class multi_contact_walking.**MultiContactWalkingFSM**(*stances*, *robot*, *swing_height*, *cycle=False*)

Finite State Machine for biped walking.

Parameters

- **stances** (*list of Stances*) – Consecutives stances traversed by the FSM.
- **robot** (Robot) – Controller robot.
- **swing_height** (*scalar*) – Relative height in [m] for the apex of swing foot trajectories.
- **cycle** (*bool, optional*) – If `True`, the first stance will succeed the last one.

on_tick(*sim*)
Update the FSM after a tick of the control loop.

Parameters **sim** (Simulation) – Instance of the current simulation.

class multi_contact_walking.**PointMassWrenchDrawer**(*point_mass*, *contact_set*)

on_tick(*sim*)
Find supporting contact forces at each COM acceleration update.

Parameters **sim** (*pymanoid.Simulation*) – Simulation instance.

class multi_contact_walking.**PreviewBuffer**(*u_dim*, *callback=None*)
Buffer used to store controls on a preview window.

Parameters

- **u_dim** (*int*) – Dimension of preview control vectors.
- **callback** (*function, optional*) – Function to call with each new control *(u, dT)*.

get_next_control()
Get the next pair (*u*, *dT*) in the preview window.

Returns **(u, dT)** – Next control in the preview window.

Return type array, scalar

on_tick(*sim*)
Entry point called at each simulation tick.

Parameters **sim** (Simulation) – Current simulation instance.

reset()
Reset preview buffer to its empty state.

update_preview(*U*, *dT*, *switch_step=None*)
Update preview with a filled PreviewControl object.

Parameters

- **U** (*array, shape=(N * d,)*) – Vector of stacked preview controls, each of dimension *d*.
- **dT** (*array, shape=(N,)*) – Sequence of durations, one for each preview control.
- **switch_step** (*int, optional*) – Optional index of a contact-switch step in the sequence.

class multi_contact_walking.**PreviewDrawer**
Draw preview trajectory, in blue and yellow for the SS and DS parts respectively.

on_tick(*sim*)
Entry point called at each simulation tick.

Parameters **sim** (Simulation) – Instance of the current simulation.

class multi_contact_walking.**SwingFoot**(*swing_height*, *color='c'*)

Invisible body used for swing foot interpolation.

Parameters

- **swing_height** (*double*) – Height in [m] for the apex of the foot trajectory.
- **color** (*char, optional*) – Color applied to all links of the KinBody.

reset(*start_pose*, *end_pose*)

Reset both end poses of the interpolation.

Parameters

- **start_pose** (*array*) – New start pose.
- **end_pose** (*array*) – New end pose.

update_pose(*s*)

Update pose to a given index `s` in the swing-foot motion.

Parameters **s** (*scalar*) – Index between 0 and 1 in the swing-foot motion.

class multi_contact_walking.**TubeDrawer**

Draw preview COM tube along with its dual acceleration cone.

on_tick(*sim*)

Entry point called at each simulation tick.

Parameters **sim** (Simulation) – Instance of the current simulation.

class multi_contact_walking.**UpdateCOMTargetAccel**(*com_target*, *preview_buffer*)

on_tick(*sim*)

Entry point called at each simulation tick.

Parameters **sim** (Simulation) – Instance of the current simulation.

multi_contact_walking.**generate_staircase**(*radius*, *angular_step*, *height*, *roughness*, *friction*, *ds_duration*, *ss_duration*, *init_com_offset=None*)

Generate a new slanted staircase with tilted steps.

Parameters

- **radius** (*scalar*) – Staircase radius in [m].
- **angular_step** (*scalar*) – Angular step between contacts in [rad].
- **height** (*scalar*) – Altitude variation in [m].

- **roughness** (*scalar*) – Amplitude of contact roll, pitch and yaw in [rad].
- **friction** (*scalar*) – Friction coefficient between a robot foot and a step.
- **ds_duration** (*scalar*) – Duration of double-support phases in [s].
- **ss_duration** (*scalar*) – Duration of single-support phases in [s].
- **init_com_offset** (*array, optional*) – Initial offset applied to first stance.

multi_contact_walking.**random**(*size=None*)
Return random floats in the half-open interval [0.0, 1.0). Alias for *random_sample* to ease forward-porting to the new random API.

multi_contact_walking.**seed**(*self*, *seed=None*)
Reseed a legacy MT19937 BitGenerator

Notes

This is a convenience, legacy function.

The best practice is to **not** reseed a BitGenerator, rather to recreate a new one. This method is here for legacy reasons. This example demonstrates best practice.

```
>>> from numpy.random import MT19937
>>> from numpy.random import RandomState, SeedSequence
>>> rs = RandomState(MT19937(SeedSequence(123456789)))
# Later, you want to restart the stream
>>> rs = RandomState(MT19937(SeedSequence(987654321)))
```

1.6 VHIP stabilization

This example compares two stabilizers for the inverted pendulum model. The first one (baseline) is based on proportional feedback of the 3D DCM [Englsberger15]. The second one (proposed) performs proportional feedback of a 4D DCM of the same model [Caron20].

class vhip_stabilization.**BonusPolePlacementStabilizer**(*pendulum*, *k_z*)
This is a "bonus" stabilizer, not reported in the paper, that was an intermediate step in our derivation of the VHIPQPStabilizer.

Parameters

- **pendulum** (*pymanoid.models.InvertedPendulum*) – Inverted pendulum to stabilize.

- **k_z** (*scalar*) – Feedback gain between DCM altitude and normalized leg stiffness input.

Notes

This stabilizer also performs pole placement on a 4D DCM (using a velocity rather than position DCM though), but contrary to VHIPQPStabilizer it doesn't force the closed-loop matrix to be diagonal. We started out exploring this stabilizer first.

The first thing to observe by direct pole placement is that the gain matrix has essentially four non-zero gains in general. You can try out the *set_poles()* function to verify this.

The closed-loop system with four gains has structure: in the horizontal plane it is equivalent to the VRPStabilizer, and the normalized leg stiffness lambda depends on both the vertical DCM and the natural frequency omega. We observed that this system performs identically to the previous one in the horizontal plane, and always worse than the previous one vertically.

However, raising the k_z (vertical DCM to lambda) gain to large values, we noticed that the vertical tracking of this stabilizer converged to that of the VRPStabilizer. In the limit where k_z goes to infinity, the system slides on the constraint given by Equation (21) in the paper. This is how we came to the derivation of the VHIPQPStabilizer.

compute_compensation()
Compute CoP and normalized leg stiffness compensation.

set_critical_gains(*k_z*)
Set critical gain `k_omega` for a desired vertical DCM gain `k_z`.

Parameters **k_z** (*scalar*) – Desired vertical DCM to normalized leg stiffness gain.

set_gains(*gains*)
Set gains from 4D DCM error to 3D input `[zmp_x, zmp_y, lambda]`.

Parameters **gains** (*(4,) array*) – List of gains `[k_x, k_y, k_z, k_omega]`.

set_poles(*poles*)
Place poles using SciPy's implementation of Kautsky et al.'s algorithm.

Parameters **poles** (*(4,) array*) – Desired poles of the closed-loop system.

class vhip_stabilization.**DCMPlotter**(*stabilizers*)

on_tick(*sim*)
Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

class vhip_stabilization.**Plotter**(*stabilizers*)

on_tick(*sim*)
Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

class vhip_stabilization.**Pusher**(*pendulums*, *gain=0.1*)
Send impulses to the inverted pendulum every once in a while.

Parameters

- **pendulums** (*list of pymanoid.models.InvertedPendulum*) – Inverted pendulums to de-stabilize.
- **gain** (*scalar*) – Magnitude of velocity jumps.

Notes

You know, I've seen a lot of people walkin' 'round // With tombstones in their eyes // But the pusher don't care // Ah, if you live or if you die

on_tick(*sim*)
Apply regular impulses to the inverted pendulum.

Parameters **sim** (*pymanoid.Simulation*) – Simulation instance.

class vhip_stabilization.**Stabilizer**(*pendulum*)
Base class for stabilizer processes.

Parameters **pendulum** (*pymanoid.models.InvertedPendulum*) – Inverted pendulum to stabilize.

contact
Contact frame and area dimensions.

Type pymanoid.Contact

dcm
Position of the DCM in the world frame.

Type (3,) array

omega
Instantaneous natural frequency of the pendulum.

Type scalar

pendulum
Measured state of the reduced model.

Type pymanoid.InvertedPendulum

ref_com
Desired center of mass (CoM) position.

Type (3,) array

ref_comd
Desired CoM velocity.

Type (3,) array

ref_cop
Desired center of pressure (CoP).

Type (3,) array

ref_lambda
Desired normalized leg stiffness.

Type scalar

ref_omega
Desired natural frequency.

Type scalar

on_tick(*sim*)
Set inverted pendulum CoP and stiffness inputs.

Parameters **sim** (*pymanoid.Simulation*) – Simulation instance.

reset_pendulum()
Reset inverted pendulum to its reference state.

class vhip_stabilization.**VHIPQPStabilizer**(*pendulum*)
Stabilizer based on proportional feedback of the 4D divergent component of motion of the variable-height inverted pendulum (VHIP).

Parameters **pendulum** (*pymanoid.models.InvertedPendulum*) – Inverted pendulum to stabilize.

Notes

This implementation transcripts QP matrices from *VHIPStabilizer*. We checked that the two produce the same outputs before switching to C++ in <https://github.com/stephane-caron/vhip_walking_controller/>. (This step would not have been necessary if we had a modeling language for convex optimization directly in C++.)

compute_compensation()
Compute CoP and normalized leg stiffness compensation.

class vhip_stabilization.**VHIPStabilizer**(*pendulum*)
Stabilizer based on proportional feedback of the 4D divergent component of motion of the variable-height inverted pendulum (VHIP).

Parameters **pendulum** (*pymanoid.models.InvertedPendulum*) – Inverted pendulum to stabilize.

Notes

This implementation uses CVXPY <https://www.cvxpy.org/>. Using this modeling language here allowed us to try various formulations of the controller before converging on this one. We can only praise the agility of this approach, as opposed to e.g. writing QP matrices directly.

See "Biped Stabilization by Linear Feedback of the Variable-Height Inverted Pendulum Model" (Caron, 2019) for detail on the controller itself.

compute_compensation()
Compute CoP and normalized leg stiffness compensation.

class vhip_stabilization.**VRPStabilizer**(*pendulum*)
Inverted pendulum stabilizer based on proportional feedback of the 3D divergent component of motion (DCM) applied to the virtual repellent point (VRP).

Parameters **pendulum** (*pymanoid.models.InvertedPendulum*) – Inverted pendulum to stabilize.

ref_dcm
Desired (3D) divergent component of motion.

Type (3,) array

ref_vrp
Desired virtual repellent point (VRP).

Type (3,) array

Notes

See "Three-Dimensional Bipedal Walking Control Based on Divergent Component of Motion" (Englsberger et al., IEEE Transactions on Robotics) for details.

compute_compensation()
Compute CoP and normalized leg stiffness compensation.

vhip_stabilization.**push_three_times**()
Apply three pushes of increasing magnitude to the CoM.

Note: This is the function used to generate Fig. 1 in the manuscript <https://hal.archives-ouvertes.fr/hal-02289919v1/document>.

---

vhip_stabilization.**record_video**()
Record accompanying video of the paper.

2 Interpolation

Interpolation is the operation of computing trajectories that connect an initial state (position and its derivatives such as velocity and acceleration) to a desired one.

2.1 Position

In the absence of inequality constraints, interpolation for positions can be solved by polynomial interpolation. The following functions and classes provide simple interfaces for this common operation.

pymanoid.interp.**interpolate_cubic_bezier**(*p0*, *p1*, *p2*, *p3*)

Compute the cubic Bezier curve corresponding to four control points.

Parameters

- **p0** (*(3,) array*) – First control point.
- **p1** (*(3,) array*) – Second control point.
- **p2** (*(3,) array*) – Third control point.
- **p3** (*(3,) array*) – Fourth control point.

Returns **P** – Polynomial function of the Bezier curve.

Return type NDPolynomial

pymanoid.interp.**interpolate_cubic_hermite**(*p0*, *v0*, *p1*, *v1*)

Compute the third-order polynomial of the Hermite curve connecting (p_0, v_0) to (p_1, v_1).

Parameters

- **p0** (*(3,) array*) – Start point.
- **v0** (*(3,) array*) – Start velocity.
- **p1** (*(3,) array*) – End point.
- **v1** (*(3,) array*) – End velocity.

Returns **P** – Polynomial function of the Hermite curve.

Return type NDPolynomial

class pymanoid.interp.**LinearPosInterpolator**(*start_pos*, *end_pos*, *duration*, *body=None*)

Linear interpolation between two positions.

Parameters

- **start_pos** (*(3,) array*) – Initial position to interpolate from.
- **end_pos** (*(3,) array*) – End position to interpolate to.
- **body** (*Body, optional*) – Body affected by the update function.

class pymanoid.interp.**CubicPosInterpolator**(*start_pos*, *end_pos*, *duration*, *body=None*)

Cubic interpolation between two positions with zero-velocity boundary conditions.

Parameters

- **start_pos** (*(3,) array*) – Initial position to interpolate from.
- **end_pos** (*(3,) array*) – End position to interpolate to.
- **body** (Body, *optional*) – Body affected by the update function.

class pymanoid.interp.**QuinticPosInterpolator**(*start_pos*, *end_pos*, *duration*, *body=None*)

Quintic interpolation between two positions with zero-velocity and zero-acceleration boundary conditions.

Parameters

- **start_pos** (*(3,) array*) – Initial position to interpolate from.
- **end_pos** (*(3,) array*) – End position to interpolate to.
- **body** (Body, *optional*) – Body affected by the update function.

2.2 Orientation

In the absence of inequality constraints, interpolation for orientations can be solved by spherical linear interpolation. The following functions and classes provide simple interfaces for this common operation. They compute *poses*: following OpenRAVE terminology, the pose of a rigid body denotes the 7D vector of its 4D orientation quaternion followed by its 3D position coordinates.

pymanoid.interp.**interpolate_pose_linear**(*pose0*, *pose1*, *x*)

Standalone function for linear pose interpolation.

Parameters

- **pose0** (*(7,) array*) – First pose.
- **pose1** (*(7,) array*) – Second pose.
- **x** (*scalar*) – Number between 0 and 1.

Returns **pose** – Linear interpolation between the first two arguments.

Return type (7,) array

pymanoid.interp.**interpolate_pose_quadratic**(*pose0*, *pose1*, *x*)

Pose interpolation that is linear in orientation (SLERP) and quadratic (Bezier) in position.

Parameters

- **pose0** (*(7,) array*) – First pose.
- **pose1** (*(7,) array*) – Second pose.

- **x** (*scalar*) – Number between 0 and 1.

Returns **pose** – Linear interpolation between the first two arguments.

Return type (7,) array

Note: Initial and final linear velocities on the trajectory are zero.

---

class pymanoid.interp.**LinearPoseInterpolator**(*start_pose*, *end_pose*, *duration*, *body=None*)

Interpolate a body trajectory between two poses. Intermediate positions are interpolated linearly, while orientations are computed by spherical linear interpolation.

Parameters

- **start_pose** (*(7,) array*) – Initial pose to interpolate from.
- **end_pose** (*(7,) array*) – End pose to interpolate to.
- **body** (Body, *optional*) – Body affected by the update function.

eval_pos(*s*)

Evaluate position at index s between 0 and 1.

Parameters **s** (*scalar*) – Normalized trajectory index between 0 and 1.

class pymanoid.interp.**CubicPoseInterpolator**(*start_pose*, *end_pose*, *duration*, *body=None*)

Interpolate a body trajectory between two poses. Intermediate positions are interpolated cubically with zero boundary velocities, while orientations are computed by spherical linear interpolation.

Parameters

- **start_pose** (*(7,) array*) – Initial pose to interpolate from.
- **end_pose** (*(7,) array*) – End pose to interpolate to.
- **body** (Body, *optional*) – Body affected by the update function.

eval_pos(*s*)

Evaluate position at index s between 0 and 1.

Parameters **s** (*scalar*) – Normalized trajectory index between 0 and 1.

class pymanoid.interp.**PoseInterpolator**(*start_pose*, *end_pose*, *duration*, *body=None*)

Interpolate a body trajectory between two poses. Orientations are computed by spherical linear interpolation.

Parameters

- **start_pose** (*(7,) array*) – Initial pose to interpolate from.

- **end_pose** (*(7,) array*) – End pose to interpolate to.
- **body** (*Body, optional*) – Body affected by the update function.

draw(*color='b'*)
Draw positions of the interpolated trajectory.

Parameters **color** (*char or triplet, optional*) – Color letter or RGB values, default is 'b' for blue.

Returns **handle** – OpenRAVE graphical handle. Must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type openravepy.GraphHandle

eval_pos(*s*)
Evaluate position at index s between 0 and 1.

Parameters **s** (*scalar*) – Normalized trajectory index between 0 and 1.

class pymanoid.interp.**QuinticPoseInterpolator**(*start_pose*, *end_pose*, *duration*, *body=None*)
Interpolate a body trajectory between two poses. Intermediate positions are interpolated quintically with zero-velocity and zero-acceleration boundary conditions, while orientations are computed by spherical linear interpolation.

Parameters

- **start_pose** (*(7,) array*) – Initial pose to interpolate from.
- **end_pose** (*(7,) array*) – End pose to interpolate to.
- **body** (*Body, optional*) – Body affected by the update function.

eval_pos(*s*)
Evaluate position at index s between 0 and 1.

Parameters **s** (*scalar*) – Normalized trajectory index between 0 and 1.

3 Forward kinematics

3.1 Rigid transformations

Rotations and rigid-body transformations can be represented in many ways. For rotations, the three main formats used in pymanoid are:

- **Roll-pitch-yaw angles:** that is to say Euler angles corresponding to the sequence (1, 2, 3).
- **Quaternions:** 4D vectors *[w x y z]*, with the scalar term *w* coming first following the OpenRAVE convention.

- **Rotation matrices:** 3×3 matrices R whose inverse is equal to their transpose.

Rigid-body transformations can be represented by:

- **Poses:** 7D vectors consisting of the quaternion of the orientation followed by its position.
- **Transformation matrices:** 4×4 matrices T.

Functions are provided to convert between all these representations. Most of them are adapted from the comprehensive guide [Diebel06].

pymanoid.transformations.**apply_transform**(*T*, *p*)

Apply a transformation matrix *T* to point coordinates *p*.

Parameters

- **T** (*(4, 4) array*) – Homogeneous transformation matrix.
- **p** (*(7,) or (3,) array*) – Pose or point coordinates.

Returns **Tp** – Result after applying the transformation.

Return type (7,) or (3,) array

Notes

For a single point, it is faster to apply the matrix multiplication directly rather than calling the OpenRAVE function:

```
In [33]: %timeit dot(T, hstack([p, 1]))[:3]
100000 loops, best of 3: 3.82 µs per loop

In [34]: %timeit list(transformPoints(T, [p]))
100000 loops, best of 3: 6.4 µs per loop
```

pymanoid.transformations.**crossmat**(*x*)

Cross-product matrix of a 3D vector.

Parameters **x** (*(3,) array*) – Vector on the left-hand side of the cross product.

Returns **C** – Cross-product matrix of *x*.

Return type (3, 3) array

pymanoid.transformations.**integrate_angular_acceleration**(*R*, *omega*, *omegad*, *dt*)

Integrate constant angular acceleration $\dot{\omega}$ for a duration *dt* starting from the orientation R and angular velocity ω.

Parameters

- **R** (*(3, 3) array*) – Rotation matrix.
- **omega** (*(3,) array*) – Initial angular velocity.

- **omegad** (*(3,) array*) – Constant angular acceleration.
- **dt** (*scalar*) – Duration.

Returns **Ri** – Rotation matrix after integration.

Return type (3, 3) array

Notes

This function applies Rodrigues' rotation formula. See also the following nice post on integrating rotations.

pymanoid.transformations.**integrate_body_acceleration**(*T*, *v*, *a*, *dt*)

Integrate constant body acceleration a for a duration *dt* starting from the affine transform T and body velocity v.

Parameters

- **T** (*(4, 4) array*) – Affine transform of the rigid body.
- **v** (*(3,) array*) – Initial body velocity of the rigid body.
- **a** (*(3,) array*) – Constant body acceleration.
- **dt** (*scalar*) – Duration.

Returns **T_f** – Affine transform after integration.

Return type (3, 3) array

Notes

With full frame notations, the affine transform $T = {}^{W}T_{B}$ maps vectors from the rigid body frame B to the inertial frame W. Its body velocity $v = \begin{bmatrix} v_B & \omega \end{bmatrix}$ is a twist in the inertial frame, with coordinates taken at the origin of B. The same goes for its body acceleration $a = \begin{bmatrix} a_B & \dot{\omega} \end{bmatrix}$.

pymanoid.transformations.**magnus_expansion**(*omega*, *omegad*, *dt*)

Compute the integral Ω obtained by integrating $\dot{\omega}$ for a duration *dt* starting from ω.

Parameters

- **omega** (*(3,) array*) – Initial angular velocity.
- **omegad** (*(3,) array*) – Constant angular acceleration.
- **dt** (*scalar*) – Duration.

Returns **Omega** – Integral of `omegad` for `dt` starting from `omega`.

Return type (3,) array

Notes

This function only computes the first three terms of the Magnus expansion. See <https://github.com/jrl-umi3218/RBDyn/pull/66/files> if you need a more advanced version.

pymanoid.transformations.**pose_from_transform**(*T*)
Pose vector from a homogeneous transformation matrix.

Parameters **T** (*(4, 4) array*) – Homogeneous transformation matrix.

Returns **pose** – Pose vector *[qw qx qy qz x y z]* of the transformation matrix.

Return type (7,) array

pymanoid.transformations.**quat_from_rotation_matrix**(*R*)
Quaternion from rotation matrix.

Parameters **R** (*(3, 3) array*) – Rotation matrix.

Returns **quat** – Quaternion in *[w x y z]* format.

Return type (4,) array

pymanoid.transformations.**quat_from_rpy**(*rpy*)
Quaternion frmo roll-pitch-yaw angles.

Parameters **rpy** (*(3,) array*) – Vector of roll-pitch-yaw angles in [rad].

Returns **quat** – Quaternion in *[w x y z]* format.

Return type (4,) array

Notes

Roll-pitch-yaw are Euler angles corresponding to the sequence (1, 2, 3).

pymanoid.transformations.**rotation_matrix_from_quat**(*quat*)
Rotation matrix from quaternion.

Parameters **quat** (*(4,) array*) – Quaternion in *[w x y z]* format.

Returns **R** – Rotation matrix.

Return type (3, 3) array

pymanoid.transformations.**rotation_matrix_from_rpy**(*rpy*)
Rotation matrix from roll-pitch-yaw angles.

Parameters **rpy** (*(3,) array*) – Vector of roll-pitch-yaw angles in [rad].

Returns **R** – Rotation matrix.

Return type (3, 3) array

pymanoid.transformations.**rpy_from_quat**(*quat*)
Roll-pitch-yaw angles of a quaternion.

Parameters **quat** (*(4,) array*) – Quaternion in *[w x y z]* format.

Returns **rpy** – Array of roll-pitch-yaw angles, in [rad].

Return type (3,) array

Notes

Roll-pitch-yaw are Euler angles corresponding to the sequence (1, 2, 3).

pymanoid.transformations.**rpy_from_rotation_matrix**(*R*)
Roll-pitch-yaw angles of rotation matrix.

Parameters **R** (*array*) – Rotation matrix.

Returns **rpy** – Array of roll-pitch-yaw angles, in [rad].

Return type (3,) array

Notes

Roll-pitch-yaw are Euler angles corresponding to the sequence (1, 2, 3).

pymanoid.transformations.**transform_from_pose**(*pose*)
Transformation matrix from a pose vector.

Parameters **pose** (*(7,) array*) – Pose vector *[qw qx qy qz x y z]*.

Returns **T** – Homogeneous transformation matrix of the pose vector.

Return type (4, 4) array

pymanoid.transformations.**transform_inverse**(*T*)
Inverse of a transformation matrix. Yields the same result but faster than `numpy.linalg.inv()` on such matrices.

Parameters **T** (*(4, 4) array*) – Homogeneous transformation matrix.

Returns **T_inv** – Inverse of *T*.

Return type (4, 4) array

3.2 Rigid body model

class pymanoid.body.**Body**(*rave_body*, *pos=None*, *rpy=None*, *pose=None*, *color=None*, *visible=True*)
Base class for rigid bodies. Wraps OpenRAVE's KinBody type.

Parameters

- **rave_body** (*openravepy.KinBody*) – OpenRAVE body to wrap.
- **pos** (*array, shape=(3,), optional*) – Initial position in inertial frame.

- **rpy** (*array, shape=(3,), optional*) – Initial orientation in inertial frame.
- **pose** (*array, shape=(7,), optional*) – Initial pose. Supersedes `pos` and `rpy` if they are provided at the same time.
- **color** (*char, optional*) – Color code in matplotlib convention ('b' for blue, 'g' for green, ...).
- **visible** (*bool, optional*) – Visibility in the GUI.

property R

Rotation matrix *R* from local to world coordinates.

property T

Homogeneous coordinates of the rigid body.

These coordinates describe the orientation and position of the rigid body by the 4 x 4 transformation matrix

$$T = \begin{bmatrix} R & p \\ 0_{1\times 3} & 1 \end{bmatrix}$$

where *R* is a *3 x 3* rotation matrix and *p* is the vector of position coordinates.

Notes

More precisely, *T* is the transformation matrix *from* the body frame *to* the world frame: if $\tilde{p}_{\text{body}} = [x\ y\ z\ 1]$ denotes the homogeneous coordinates of a point in the body frame, then the homogeneous coordinates of this point in the world frame are $\tilde{p}_{\text{world}} = T\tilde{p}_{\text{body}}$.

property adjoint_matrix

Adjoint matrix converting wrenches in the local frame $\mathcal{L}$ to the inertial frame $\mathcal{W}$, that is the matrix ${}^{\mathcal{W}}A_{\mathcal{L}}$ such that:

$${}^{\mathcal{W}}w = {}^{\mathcal{W}}A_{\mathcal{L}}{}^{\mathcal{L}}w$$

Returns **A** – Adjoint matrix.

Return type array

apply_twist(*v*, *omega*, *dt*)

Apply a twist $[v\ \omega]$ defined in the local coordinate frame.

Parameters

- **v** (*(3,) array*) – Linear velocity in local frame.
- **omega** (*(3,) array*) – Angular velocity in local frame.
- **dt** (*scalar*) – Duration of twist application in [s].

property b

Binormal vector directing the *y*-axis of the body frame.

dist (*point*)
Distance from the body frame origin to another point.

Parameters **point** (*array or Point*) – Point to compute the distance to.

hide ()
Make the body invisible.

property index
OpenRAVE index of the body.

Notes

This index is notably used to compute jacobians and hessians.

property n
Normal vector directing the z-axis of the body frame.

property name
Body name.

property normal
Normal vector directing the z-axis of the body frame.

property p
Position coordinates *[x y z]* in the world frame.

property pitch
Pitch angle of the body orientation.

property pos
Position coordinates *[x y z]* in the world frame.

property pose
Body pose as a 7D quaternion + position vector.

The pose vector $[q_w\, q_x\, q_y\, q_z\, x\, y\, z]$ consists of a quaternion $q = [q_w\, q_x\, q_y\, q_z]$ (with the real term q_w coming first) for the body orientation, followed by the coordinates $p = [x\, y\, z]$ in the world frame.

property quat
Quaternion of the rigid body orientation.

remove ()
Remove body from OpenRAVE environment.

property roll
Roll angle of the body orientation.

property rotation_matrix
Rotation matrix *R* from local to world coordinates.

property rpy
Roll-pitch-yaw angles.

They correspond to Euleur angles for the sequence (1, 2, 3). See [Diebel06] for details.

set_color(*color*)
Set the color of the rigid body.

Parameters **color** (*tuple or string*) – RGB tuple, or color code in matplotlib convention.

set_name(*name*)
Set body name in OpenRAVE scope.

name [string] Body name.

set_pitch(*pitch*)
Set the pitch angle of the body orientation.

Parameters **pitch** (*scalar*) – Pitch angle in [rad].

set_pos(*pos*)
Set the position of the body in the world frame.

Parameters **pos** (*array, shape=(3,)*) – 3D vector of position coordinates.

set_pose(*pose*)
Set the 7D pose of the body orientation.

Parameters **pose** (*(7,) array*) – Pose of the body, i.e. quaternion + position in world frame.

set_quat(*quat*)
Set the quaternion of the body orientation.

Parameters **quat** (*(4,) array*) – Quaternion in (w, x, y, z) format.

set_roll(*roll*)
Set the roll angle of the body orientation.

Parameters **roll** (*scalar*) – Roll angle in [rad].

set_rotation_matrix(*R*)
Set the orientation of the rigid body.

Recall that this orientation is described by the rotation matrix *R from* the body frame *to* the world frame.

Parameters **R** (*(3, 3) array*) – Rotation matrix.

set_rpy(*rpy*)
Set the roll-pitch-yaw angles of the body orientation.

Parameters **rpy** (*scalar triplet*) – Triplet *(r, p, y)* of roll-pitch-yaw angles.

set_transform(*T*)
Set homogeneous coordinates of the rigid body.

Parameters **T** (*array, shape=(4, 4)*) – Transform matrix.

set_transparency (*transparency*)
Set the transparency of the rigid body.

Parameters **transparency** (*double, optional*) – Transparency value from 0 (opaque) to 1 (invisible).

set_x (*x*)
Set the *x*-coordinate of the body in the world frame.

Parameters **x** (*scalar*) – New *x*-coordinate.

set_y (*y*)
Set the *y*-coordinate of the body in the world frame.

Parameters **y** (*scalar*) – New *y*-coordinate.

set_yaw (*yaw*)
Set the yaw angle of the body orientation.

Parameters **yaw** (*scalar*) – Yaw angle in [rad].

set_z (*z*)
Set the *z*-coordinate of the body in the world frame.

Parameters **z** (*scalar*) – New *z*-coordinate.

show ()
Make the body visible.

property t
Tangent vector directing the *x*-axis of the body frame.

property transform
Homogeneous coordinates of the rigid body.

These coordinates describe the orientation and position of the rigid body by the 4 x 4 transformation matrix

$$T = \begin{bmatrix} R & p \\ 0_{1\times 3} & 1 \end{bmatrix}$$

where *R* is a *3 x 3* rotation matrix and *p* is the vector of position coordinates.

Notes

More precisely, *T* is the transformation matrix *from* the body frame *to* the world frame: if $\tilde{p}_{\text{body}} = [x\ y\ z\ 1]$ denotes the homogeneous coordinates of a point in the body frame, then the homogeneous coordinates of this point in the world frame are $\tilde{p}_{\text{world}} = T\tilde{p}_{\text{body}}$.

translate (*translation*)
Apply a translation to the body.

Parameters **translation** (*(3,) array*) – Offset to apply to the position (world coordinates) of the body.

property x
x-coordinate in the world frame.

property y
y-coordinate in the world frame.

property yaw
Yaw angle of the body orientation.

property z
z-coordinate in the world frame.

class pymanoid.body.**Box**(*X*, *Y*, *Z*, *pos=None*, *rpy=None*, *pose=None*, *color='r'*, *visible=True*, *dZ=0.0*)
Rectangular box.

Parameters

- **X** (*scalar*) – Box half-length in [m].
- **Y** (*scalar*) – Box half-width in [m].
- **Z** (*scalar*) – Box half-height in [m].
- **pos** (*array, shape=(3,)*) – Initial position in the world frame.
- **rpy** (*array, shape=(3,)*) – Initial orientation in the world frame.
- **pose** (*array, shape=(7,)*) – Initial pose in the world frame.
- **color** (*char*) – Color letter in [‘r’, ‘g’, ‘b’].
- **visible** (*bool, optional*) – Visibility in the GUI.
- **dZ** (*scalar, optional*) – Shift in box normal coordinates used to make Contact slabs.

class pymanoid.body.**Cube**(*size*, *pos=None*, *rpy=None*, *pose=None*, *color='r'*, *visible=True*)
Cube.

Parameters

- **size** (*scalar*) – Half-length of a side of the cube in [m].
- **pos** (*array, shape=(3,)*) – Initial position in the world frame.
- **rpy** (*array, shape=(3,)*) – Initial orientation in the world frame.
- **pose** (*array, shape=(7,)*) – Initial pose in the world frame.
- **color** (*char*) – Color letter in [‘r’, ‘g’, ‘b’].
- **visible** (*bool, optional*) – Visibility in the GUI.

class pymanoid.body.**Manipulator**(*manipulator*, *pos=None*, *rpy=None*, *pose=None*, *color=None*, *visible=True*, *shape=None*, *friction=None*)

Manipulators are special bodies with an end-effector property.

Parameters

- **manipulator** (*openravepy.KinBody*) – OpenRAVE manipulator object.
- **pos** (*array, shape=(3,), optional*) – Initial position in inertial frame.
- **rpy** (*array, shape=(3,), optional*) – Initial orientation in inertial frame.
- **pose** (*array, shape=(7,), optional*) – Initial pose. Supersedes `pos` and `rpy` if they are provided at the same time.
- **color** (*char, optional*) – Color code in matplotlib convention ('r' for red, 'b' for blue, etc.).
- **visible** (*bool, optional*) – Visibility in the GUI.
- **shape** (*(scalar, scalar), optional*) – Dimensions (half-length, half-width) of a contact patch in [m].
- **friction** (*scalar, optional*) – Static friction coefficient for potential contacts.

property force

Resultant of contact forces applied on the effector (if defined). Coordinates are given in the end-effector frame.

get_contact(*pos=None*, *shape=None*)

Get contact located at the current manipulator pose.

Parameters

- **pos** (*(3,) array, optional*) – Override manipulator position with this one.
- **shape** (*(scalar, scalar), optional*) – Dimensions (half-length, half-width) of contact patch in [m].

Returns **contact** – Contact located at manipulator pose.

Return type *Contact*

property index

Index used in Jacobian and Hessian computations.

property moment

Moment of contact forces applied on the effector (if defined). Coordinates are given in the end-effector frame.

class pymanoid.body.**Point**(*pos*, *vel=None*, *accel=None*, *size=0.01*, *color='r'*, *visible=True*)

Points represented by cubes with a default size.

Parameters

- **pos** (*array, shape=(3,)*) – Initial position in the world frame.
- **vel** (*array, shape=(3,), optional*) – Initial velocity in the world frame.
- **accel** (*array, shape=(3,), optional*) – Initial acceleration in the world frame.
- **size** (*scalar, optional*) – Half-length of a side of the cube in [m].
- **color** (*char*) – Color letter in [‘r’, ‘g’, ‘b’].
- **visible** (*bool, optional*) – Visibility in the GUI.

copy(*color='r'*, *visible=True*)

Copy constructor.

Parameters

- **color** (*char, optional*) – Color of the copy, in [‘r’, ‘g’, ‘b’].
- **visible** (*bool, optional*) – Should the copy be visible?

integrate_constant_accel(*pdd*, *dt*)

Apply Euler integration for a constant acceleration.

Parameters

- **pdd** (*array, shape=(3,)*) – Point acceleration in the world frame.
- **dt** (*scalar*) – Duration in [s].

integrate_constant_jerk(*pddd*, *dt*)

Apply Euler integration for a constant jerk.

Parameters

- **pddd** (*array, shape=(3,)*) – Point jerk in the world frame.
- **dt** (*scalar*) – Duration in [s].

property pd

Point velocity.

property pdd

Point acceleration.

set_accel(*pdd*)

Update the point acceleration.

Parameters **pdd** (*array, shape=(3,)*) – Acceleration coordinates in the world frame.

set_vel(*pd*)
Update the point velocity.

Parameters **pd** (*array, shape=(3,)*) – Velocity coordinates in the world frame.

property xd
Point velocity along x-axis.

property xdd
Point acceleration along x-axis.

property yd
Point velocity along y-axis.

property ydd
Point acceleration along y-axis.

property zd
Point velocity along z-axis.

property zdd
Point acceleration along z-axis.

class pymanoid.body.**PointMass**(*pos*, *mass*, *vel=None*, *color='r'*, *visible=True*, *size=None*)
Point with a mass property and a size proportional to it.

Parameters

- **pos** (*(3,) array*) – Initial position in the world frame.
- **mass** (*scalar*) – Total mass in [kg].
- **vel** (*(3,) array, optional*) – Initial velocity in the world frame.
- **color** (*char, optional*) – Color letter in [‘r’, ‘g’, ‘b’].
- **visible** (*bool, optional*) – Visibility in the GUI.
- **size** (*scalar, optional*) – Half-length of a side of the CoM cube handle, in [m].

copy(*color='r'*, *visible=True*)
Copy constructor.

Parameters

- **color** (*char, optional*) – Color of the copy, in [‘r’, ‘g’, ‘b’].
- **visible** (*bool, optional*) – Should the copy be visible?

property momentum
Linear momentum in the world frame.

3.3 Robot model

class pymanoid.robot.**Humanoid**(*path*, *root_body*)

Humanoid robots add a free base and centroidal computations.

property R

Rotation matrix *R* of from free-flyer to the world frame.

Returns **R** – Rotation matrix.

Return type array, shape=(3, 3)

property T

Rotation matrix *R* of from free-flyer to the world frame.

Returns **R** – Rotation matrix.

Return type array, shape=(3, 3)

property b

Binormal vector directing the *y*-axis of the free-flyer frame.

property cam

Centroidal angular momentum.

property com

Position of the center of mass in the world frame.

property comd

Velocity of the center of mass in the world frame.

compute_angular_momentum(*p*)

Compute the angular momentum with respect to point *p*.

Parameters **p** (*array, shape=(3,)*) – Application point *p* in world coordinates.

Returns **am** – Angular momentum of the robot at *p*.

Return type array, shape=(3,)

compute_angular_momentum_hessian(*p*)

Returns the Hessian tensor $H(q)$ such that the rate of change of the angular momentum with respect to point *P* is

$$\dot{L}_P(q, \dot{q}) = J(q)\ddot{q} + \dot{q}^T H(q)\dot{q}$$

where $J(q)$ is the angular-momentum jacobian.

Parameters **p** (*array, shape=(3,)*) – Application point in world coordinates.

Returns **H** – Hessian tensor of the angular momentum at the application point.

Return type array, shape=(3, 3, 3)

compute_angular_momentum_jacobian(*p*)

Compute the Jacobian matrix J(q) such that the angular momentum of the robot at *P* is given by:

$$L_P(q, \dot{q}) = J(q)\dot{q}$$

Parameters **p** (*array, shape=(3,)*) – Application point *P* in world coordinates.

Returns **J_am** – Jacobian giving the angular momentum of the robot at *P*.

Return type array, shape=(3,)

compute_cam()

Compute the centroidal angular momentum.

compute_cam_hessian(*q*)

Compute the matrix H(q) such that the rate of change of the CAM is

$$\dot{L}_G(q, \dot{q}) = J(q)\ddot{q} + \dot{q}^T H(q)\dot{q}$$

Parameters **q** (*array*) – Vector of joint angles.

Returns **H** – Hessian matrix for the centroidal angular-momentum.

Return type array

compute_cam_jacobian()

Compute the jacobian matrix *J(q)* such that the CAM is given by:

$$L_G(q, \dot{q}) = J(q)\dot{q}$$

Returns **J_cam** – Jacobian matrix mapping to the centroidal angular momentum.

Return type array

compute_cam_rate(*qdd*)

Compute the time-derivative of the CAM.

Parameters **qdd** (*array*) – Vector of joint accelerations.

Returns **cam** – Coordinates of the angular momentum at the COM.

Return type array

Note: This function is not optimized.

---

compute_com_acceleration(*qdd*)

Compute COM acceleration from joint accelerations $\ddot{q}$.

Parameters **qdd** (*array*) – Vector of DOF accelerations.

Returns **comdd** – COM acceleration.

Return type (3,) array

compute_com_hessian()

Compute the Hessian matrix *H(q)* of the center of mass *G*, such that the acceleration of *G* in the world frame is:

$$\ddot{p}_G = J(q)\ddot{q} + \dot{q}^T H(q)\dot{q}$$

compute_com_jacobian()

Compute the Jacobian matrix $J_{\mathrm{com}}(q)$ of the center of mass *G*, such that the velocity of *G* in the world frame is:

$$\dot{p}_G = J_{\mathrm{com}}(q)\dot{q}$$

Returns **J_com** – Jacobian of the center of mass.

Return type array

compute_gravito_inertial_wrench(*qdd*, *p*)

Compute the gravito-inertial wrench at point *P*:

$$w_P = \begin{bmatrix} f^{gi} \\ \tau_P^{gi} \end{bmatrix} = \begin{bmatrix} m(g - \ddot{p}_G) \\ (p_G - p_P) \times m(g - \ddot{p}_G) - \dot{L}_G \end{bmatrix}$$

with *m* the robot mass, *g* the gravity vector, *G* the center of mass, $\ddot{p}_G$ the COM acceleration, and $\dot{L}_G$ the rate of change of the centroidal angular momentum.

Parameters

- **qdd** (*array*) – Vector of DOF accelerations.
- **p** (*array, shape=(3,)*) – Application point of the gravito-inertial wrench.

Returns **w_P** – Coordinates of the gravito-inertial wrench expressed at point P in the world frame.

Return type array, shape=(6,)

compute_net_contact_wrench(*qdd*, *p*)

Compute the gravito-inertial wrench at point *P*:

$$w_P = \begin{bmatrix} f^c \\ \tau_P^c \end{bmatrix} = \sum_i \begin{bmatrix} f_i \\ (p_i - p_P) \times f_i + \tau_i \end{bmatrix}$$

with the contact wrench (f_i, τ_i) is applied at the i^{th} contact point located at p_i.

Parameters

- **qdd** (*array*) – Vector of DOF accelerations.
- **p** (*array, shape=(3,)*) – Application point of the gravito-inertial wrench.

Returns **w_P** – Coordinates of the gravito-inertial wrench expressed at point P in the world frame.

Return type array, shape=(6,)

Notes

From the Newton-Euler equations of the system, the net contact wrench is opposite to the gravito-inertial wrench computed by *pymanoid.robot.Humanoid.compute_gravito_inertial_wrench()*.

compute_zmp(*qdd*, *origin=None*, *normal=None*)
Compute the Zero-tilting Moment Point (ZMP).

Parameters

- **qdd** (*array*) – Vector of joint accelerations.
- **origin** (*array, shape=(3,), optional*) – Origin O of the ZMP plane. Defaults to the origin of the world frame.
- **normal** (*array, shape=(3,), optional*) – Normal n of the ZMP plane. Defaults to the vertical.

Returns **zmp** – ZMP of the gravito-inertial wrench in the plane (O, n).

Return type array, shape=(3,)

Notes

See [Sardain04] for an excellent introduction to the concepts of ZMP and center of pressure. See [Caron17z] for the more general definition of ZMP support areas in arbitrary planes.

get_com_point_mass()
Get the center of mass as a PointMass instance.

Returns **com** – Center of mass of the robot.

Return type *PointMass*

get_dof_name_from_index(*index*)
Get DOF name from its index in the kinematic chain.

Parameters **index** (*integer*) – DOF index.

Returns **name** – DOF name.

Return type string

hide_com()
Hide center of mass.

property n
Normal vector directing the z-axis of the free-flyer frame.

property p
Position of the free-flyer in the world frame.

Returns **p** – Position coordinates in the world frame.

Return type array, shape=(3,)

property pose
Pose of the free-flyer in the world frame.

Returns pose – Pose in OpenRAVE format (*qw*, *qx*, *qy*, *qz*, *x*, *y*, *z*).

Return type array, shape=(7,)

property quat
Quaternion of the free-flyer orientation in the world frame.

Returns quat – Quaternion vector (*w*, *x*, *y*, *z*).

Return type array, shape=(4,)

property rpy
Orientation of the free-flyer in the world frame.

Returns rpy – Roll-pitch-yaw angles, corresponding to Euler sequence (1, 2, 3).

Return type array, shape=(3,)

set_dof_values(*q*, *dof_indices=None*, *clamp=False*)
Set the joint values of the robot.

Parameters

- **q** (*array*) – Vector of joint angle values (ordered by DOF indices).
- **dof_indices** (*list of integers, optional*) – List of DOF indices to update.
- **clamp** (*bool, optional*) – Correct *q* if it exceeds joint limits (not done by default).

set_dof_velocities(*qd*, *dof_indices=None*)
Set the joint velocities of the robot.

Parameters

- **q** (*array*) – Vector of joint angular velocities (ordered by DOF indices).
- **dof_indices** (*list of integers, optional*) – List of DOF indices to update.

set_pos(*pos*)
Set the position of the free-flyer, a.k.a. free-floating or base frame of the robot.

Parameters pos (*array, shape=(3,)*) – Position coordinates in the world frame.

set_pose(*pose*)
Set the pose of the free-flyer, a.k.a. free-floating or base frame of the robot.

Parameters pose (*array, shape=(7,)*) – Pose in OpenRAVE format (*qw*, *qx*, *qy*, *qz*, *x*, *y*, *z*).

set_quat(*quat*)
Set the orientation of the free-flyer, a.k.a. free-floating or base frame of the robot.

Parameters **quat** (*array, shape=(4,)*) – Quaternion vector (*w*, *x*, *y*, *z*).

set_rpy(*rpy*)
Set the orientation of the free-flyer, a.k.a. free-floating or base frame of the robot.

Parameters **rpy** (*array, shape=(3,)*) – Roll-pitch-yaw angles, corresponding to Euler sequence (1, 2, 3).

show_com()
Show a red ball at the location of the center of mass.

property t
Tangent vector directing the *x*-axis of the free-flyer frame.

class pymanoid.robot.**Robot**(*path=None*, *xml=None*)
Robot with a fixed base. This class wraps OpenRAVE's Robot type.

Parameters

- **path** (*string*) – Path to the COLLADA model of the robot.
- **xml** (*string, optional*) – Environment description in OpenRAVE XML format.

compute_contact_hessian(*contacts*)
Compute the contact Hessian.

Parameters **contacts** (*pymanoid.ContactSet*) – Contacts between the robot and its environment.

Returns **H_contact** – Contact Hessian.

Return type array

compute_contact_jacobian(*contacts*)
Compute the contact Jacobian.

Parameters **contacts** (*pymanoid.ContactSet*) – Contacts between the robot and its environment.

Returns **J_contact** – Contact Jacobian.

Return type array

compute_inertia_matrix()
Compute the inertia matrix $M(q)$ of the robot, that is, the matrix such that the equations of motion can be written:

$$M(q)\ddot{q} + \dot{q}^T C(q)\dot{q} + g(q) = S^T\tau + J_c(q)^T F$$

with:

- q – vector of joint angles (DOF values)

- $\dot{q}$ – vector of joint velocities
- $\ddot{q}$ – vector of joint accelerations
- $C(q)$ – Coriolis tensor (derivative of M(q) w.r.t. q)
- $g(q)$ – gravity vector
- S – selection matrix on actuated joints
- τ – vector of actuated joint torques
- $J_c(q)$ – matrix of stacked contact Jacobians
- F – vector of stacked contact wrenches

Returns **M** – Inertia matrix for the robot's current joint angles.

Return type array

Notes

This function applies the unit-vector method described by Walker & Orin in [Walker82]. It is not efficient, so if you are looking for performance, you should consider more recent libraries such as pinocchio or RBDyn.

compute_inverse_dynamics (*qdd=None*, *external_torque=None*)

Wrapper around OpenRAVE's inverse dynamics function, which implements the Recursive Newton-Euler algorithm by Walker & Orin [Walker82]. It returns the three terms t_m, t_c and t_g such that:

$$t_m = M(q)\ddot{q}$$
$$t_c = \dot{q}^T C(q)\dot{q}$$
$$t_g = g(q)$$

where the equations of motion are written:

$$t_m + t_c + t_g = S^T\tau + J_c(q)^T F$$

Parameters

- **qdd** (*array, optional*) – Vector $\ddot{q}$ of joint accelerations.
- **external_torque** (*array, optional*) – Additional vector of external joint torques for the right-hand side of the equations of motion.

Returns

- **tm** (*array*) – Torques due to inertial accelerations. Will be `None` if `qdd` is not provided.
- **tc** (*array*) – Torques due to nonlinear (centrifugal and Coriolis) effects.

- **tg** (*array*) – Torques due to gravity.

compute_link_hessian (*link*, *p=None*)

Compute the Hessian *H(q)* of a frame attached to a robot link, the acceleration of which is given by:

$$\begin{bmatrix} a_p \\ \dot{\omega} \end{bmatrix} = J(q)\ddot{q} + \dot{q}^T H(q)\dot{q}$$

where a_p is the linear acceleration of the point *p* (default is the origin of the link frame) and $\dot{\omega}$ is the angular accelerations of the frame.

Parameters

- **link** (*integer or pymanoid.Link*) – Link identifier: either a link index, or the Link object directly.
- **p** (*array*) – Point coordinates in the world frame.

compute_link_jacobian (*link*, *p=None*)

Compute the Jacobian *J(q)* of a frame attached to a given link, the velocity of this frame being given by:

$$\begin{bmatrix} v_p \\ \omega \end{bmatrix} = J(q)\dot{q}$$

where v_p is the linear velocity of the link at point *p* (default is the origin of the link frame) and ω is the angular velocity of the link.

Parameters

- **link** (*integer or pymanoid.Link*) – Link identifier: either a link index, or the Link object directly.
- **p** (*array*) – Point coordinates in the world frame.

compute_link_pos_hessian (*link*, *p=None*)

Compute the translation Hessian H(q) of a point *p* on `link`, i.e. the matrix such that the acceleration of *p* is given by:

$$a_p = J(q)\ddot{q} + \dot{q}^T H(q)\dot{q}.$$

Parameters

- **link** (*integer or pymanoid.Link*) – Link identifier: either a link index, or the Link object directly.
- **p** (*array*) – Point coordinates in the world frame.

compute_link_pos_jacobian (*link*, *p=None*)

Compute the position Jacobian of a point *p* on a given robot link.

Parameters

- **link** (*integer or pymanoid.Link*) – Link identifier: either a link index, or the Link object directly.

- **p** (*array*) – Point coordinates in the world frame.

compute_link_pose_jacobian (*link*)

Compute the pose Jacobian of a given link, i.e. the matrix *J(q)* such that:

$$\begin{bmatrix} \dot{\xi} \\ v_L \end{bmatrix} = J(q)\dot{q},$$

with ξ a quaternion for the link orientation and $v_L = \dot{p}_L$ the velocity of the origin L of the link frame, so that the link pose is $[\xi p_L]$ and the left-hand side of the equation above is its time-derivative.

Parameters **link** (*integer or pymanoid.Link*) – Link identifier: either a link index, or the Link object directly.

compute_static_gravity_torques (*external_torque=None*)

Compute static-equilibrium torques for the manipulator.

Parameters **external_torque** (*array*) – Vector of external joint torques.

Returns **tg** – Vector of static joint torques compensating gravity, and `external_torque` if applicable.

Return type array

get_dof_limits (*dof_indices=None*)

Get the pair $(q_{\min}, q_{\max})$ of DOF limits.

Parameters **dof_indices** (*list of DOF indexes, optional*) – Only compute limits for these indices.

Returns

- **q_min** (*array*) – Vector of minimal DOF values.
- **q_max** (*array*) – Vector of maximal DOF values.

Notes

This OpenRAVE function is wrapped because it is too slow in practice. On my machine:

In [1]: %timeit robot.get_dof_limits() 1000000 loops, best of 3: 237 ns per loop

In [2]: %timeit robot.rave.GetDOFLimits() 100000 loops, best of 3: 9.24 µs per loop

get_dof_values (*dof_indices=None*)

Get DOF values for a set of DOF indices.

Parameters **dof_indices** (*list of integers*) – List of DOF indices.

Returns **q_indices** – Vector of DOF values for these indices.

Return type array

get_dof_velocities(*dof_indices=None*)
Get DOF velocities for a set of DOF indices.

Parameters **dof_indices** (*list of integers*) – List of DOF indices.

Returns **qd_indices** – Vector of DOF velocities for these indices.

Return type array

get_link(*name*)
Get robot link.

Parameters **name** (*string*) – Link name in the robot model.

Returns **link** – Link handle.

Return type pymanoid.Body

hide()
Make the robot invisible.

property q
Vector of DOF values.

property qd
Vector of DOF velocities.

set_color(*color*)
Set the color of the robot.

Parameters **color** (*tuple or string*) – RGB tuple, or color code in matplotlib convention.

set_dof_limits(*q_min*, *q_max*, *dof_indices=None*)
Set DOF limits of the robot.

Parameters

- **q_min** (*list or array*) – Lower-bound on joint angle values, ordered by DOF indices.
- **q_max** (*list or array*) – Upper-bound on joint angle values, ordered by DOF indices.
- **dof_indices** (*list, optional*) – List of DOF indices to update.

set_dof_values(*q*, *dof_indices=None*, *clamp=False*)
Set DOF values of the robot.

Parameters

- **q** (*list or array*) – Joint angle values, ordered by DOF indices.

- **dof_indices** (*list, optional*) – List of DOF indices to update.
- **clamp** (*bool*) – Correct joint angles when exceeding joint limits.

set_dof_velocities(*qd*, *dof_indices=None*)
Set the joint velocities of the robot.

Parameters

- **qd** (*list or array*) – Joint angle velocities, ordered by DOF indices.
- **dof_indices** (*list, optional*) – List of DOF indices to update.

set_transparency(*transparency*)
Update robot transparency.

Parameters **transparency** (*scalar*) – Transparency value from 0 (opaque) to 1 (invisible).

show()
Make the robot visible.

4 Inverse kinematics

Inverse kinematics (IK) is the problem of computing *motions* (velocities, accelerations) that make the robot achieve a given set of *tasks*, such as putting a foot on a surface, moving the center of mass (COM) to a target location, etc. If you are not familiar with these concepts, check out this introduction to inverse kinematics.

4.1 Tasks

Tasks are the way to specify objectives to the robot model in a human-readable way.

class pymanoid.tasks.**AxisAngleContactTask**(*robot*, *link*, *target*, *weight=None*, *gain=None*, *exclude_dofs=None*)
Contact task using angle-axis rather than quaternion computations. The benefit of this task is that some degrees of constraint (DOC) of the contact constraint can be (fully or partially) disabled via the *doc_mask* attribute.

Parameters

- **robot** (Robot) – Target robot.
- **link** (*Link or string*) – Robot Link, or name of the link field in the robot argument.

- **target** (*list or array (shape=(7,)) or pymanoid.Body*) – Pose coordinates of the link's target.
- **weight** (*scalar, optional*) – Task weight used in IK cost function. If None, needs to be set later.
- **gain** (*scalar, optional*) – Proportional gain of the task.
- **exclude_dofs** (*list of integers, optional*) – DOF indices not used by task.

doc_mask
Array of dimension six whose values lie between 0 (degree of constraint fully disabled) and 1 (fully enabled). The first three values correspond to translation (Tx, Ty, Tz) while the last three values correspond to rotation (Rx, Ry, Rz).

Type array

update_target(*target*)
Update the task residual with a new target.

Parameters **target** (*Point or array or list*) – New link position target.

class pymanoid.tasks.**COMAccelTask**(*robot*, *weight=None*, *gain=None*, *exclude_dofs=None*)
COM acceleration task.

Parameters

- **robot** (Humanoid) – Target robot.
- **target** (*list or array or Point*) – Coordinates or the target COM position.
- **weight** (*scalar, optional*) – Task weight used in IK cost function. If None, needs to be set later.
- **gain** (*scalar, optional*) – Proportional gain of the task.
- **exclude_dofs** (*list of integers, optional*) – DOF indices not used by task.

Notes

Expanding $\ddot{x}_G = u$ in terms of COM Jacobian and Hessian, the equation of the task is:

$$J_{\mathrm{COM}}\dot{q}_{\mathrm{next}} = \frac{1}{2}u + J_{\mathrm{COM}}\dot{q} - \frac{1}{2}\delta t \dot{q}^T H_{\mathrm{COM}}\dot{q}$$

See the documentation of the PendulumModeTask for a detailed derivation.

class pymanoid.tasks.**COMTask**(*robot*, *target*, *weight=None*, *gain=None*, *exclude_dofs=None*)
COM tracking task.

Parameters

- **robot** (Humanoid) – Target robot.
- **target** (*list or array or* Point) – Coordinates or the target COM position.
- **weight** (*scalar, optional*) – Task weight used in IK cost function. If None, needs to be set later.
- **gain** (*scalar, optional*) – Proportional gain of the task.
- **exclude_dofs** (*list of integers, optional*) – DOF indices not used by task.

update_target(*target*)

Update the task residual with a new target.

Parameters **target** (Point *or array or list*) – New COM position target.

class pymanoid.tasks.**ContactTask**(*robot*, *link*, *target*, *weight=None*, *gain=None*, *exclude_dofs=None*)

In essence, a contact task is a *pymanoid.tasks.PoseTask* with a much (much) larger weight. For plausible motions, no task should have a weight higher than (or even comparable to) that of contact tasks.

class pymanoid.tasks.**DOFTask**(*robot*, *index*, *target*, *weight=None*, *gain=None*, *exclude_dofs=None*)

Track a reference DOF value.

Parameters

- **robot** (Robot) – Target robot.
- **index** (*string or integer*) – DOF index or string of DOF identifier.
- **target** (*scalar*) – Target DOF value.
- **weight** (*scalar, optional*) – Task weight used in IK cost function. If None, needs to be set later.
- **gain** (*scalar, optional*) – Proportional gain of the task.
- **exclude_dofs** (*list of integers, optional*) – DOF indices not used by task.

class pymanoid.tasks.**MinAccelTask**(*robot*, *weight=None*, *gain=None*, *exclude_dofs=None*)

Task to minimize joint accelerations.

Parameters

- **robot** (Robot) – Target robot.
- **weight** (*scalar, optional*) – Task weight used in IK cost function. If None, needs to be set later.
- **gain** (*scalar, optional*) – Gain of the task.

- **exclude_dofs** (*list of integers, optional*) – DOF indices not used by task.

Note: As the differential IK returns velocities, we approximate the minimization over $\ddot{q}$ by that over $(\dot{q}_{\text{next}} - \dot{q})$. See the documentation of the PendulumModeTask for details on the discrete approximation of $\ddot{q}$.

---

class pymanoid.tasks.**MinCAMTask**(*robot*, *weight=None*, *gain=None*, *exclude_dofs=None*)

Minimize the centroidal angular momentum (not its derivative).

Parameters

- **robot** (Humanoid) – Target robot.
- **weight** (*scalar, optional*) – Task weight used in IK cost function. If None, needs to be set later.
- **gain** (*scalar, optional*) – Proportional gain of the task.
- **exclude_dofs** (*list of integers, optional*) – DOF indices not used by task.

class pymanoid.tasks.**MinVelTask**(*robot*, *weight=None*, *gain=None*, *exclude_dofs=None*)

Task to minimize joint velocities

Parameters

- **robot** (Robot) – Target robot.
- **weight** (*scalar, optional*) – Task weight used in IK cost function. If None, needs to be set later.
- **gain** (*scalar, optional*) – Proportional gain of the task.
- **exclude_dofs** (*list of integers, optional*) – DOF indices not used by task.

class pymanoid.tasks.**PendulumModeTask**(*robot*, *weight=None*, *gain=None*, *exclude_dofs=None*)

Task to minimize the rate of change of the centroidal angular momentum.

Parameters

- **robot** (Humanoid) – Target robot.
- **weight** (*scalar, optional*) – Task weight used in IK cost function. If None, needs to be set later.
- **gain** (*scalar, optional*) – Proportional gain of the task.
- **exclude_dofs** (*list of integers, optional*) – DOF indices not used by task.

Notes

The way this task is implemented may be surprising. Basically, keeping a constant CAM L_G means that $\dot{L}_G = 0$, that is,

$$\frac{\mathrm{d}(J_{\mathrm{CAM}}\dot{q})}{\mathrm{d}t} = 0 \;\Leftrightarrow\; J_{\mathrm{CAM}}\ddot{q} + \dot{q}^T H_{\mathrm{CAM}}\dot{q} = 0$$

Because the IK works at the velocity level, we approximate $\ddot{q}$ by finite difference from the previous robot velocity. Assuming that the velocity $\dot{q}_{\mathrm{next}}$ output by the IK is applied immediately, joint angles become:

$$q' = q + \dot{q}_{\mathrm{next}}\delta t$$

Meanwhile, the Taylor expansion of q is

$$q' = q + \dot{q}\delta t + \frac{1}{2}\ddot{q}\delta t^2,$$

so that applying $\dot{q}_{\mathrm{next}}$ is equivalent to having the following constant acceleration over δt:

$$\ddot{q} = 2\frac{\dot{q}_{\mathrm{next}} - \dot{q}}{\delta t}.$$

Replacing in the Jacobian/Hessian expansion yields:

$$2J_{\mathrm{CAM}}\frac{\dot{q}_{\mathrm{next}} - \dot{q}}{\delta t} + +\dot{q}^T H_{\mathrm{CAM}}\dot{q} = 0.$$

Finally, the task in $\dot{q}_{\mathrm{next}}$ is:

$$J_{\mathrm{CAM}}\dot{q}_{\mathrm{next}} = J_{\mathrm{CAM}}\dot{q} - \frac{1}{2}\delta t\dot{q}^T H_{\mathrm{CAM}}\dot{q}$$

Note how there are two occurrences of J_{CAM}: one in the task residual, and the second in the task Jacobian.

class pymanoid.tasks.**PosTask**(*robot*, *link*, *target*, *weight=None*, *gain=None*, *exclude_dofs=None*)

Position task for a given link.

Parameters

- **robot** (Robot) – Target robot.
- **link** (*Link*) – One of the Link objects in the kinematic chain of the robot.
- **target** (*list or array (shape=(3,)) or pymanoid.Body*) – Coordinates of the link's target.
- **weight** (*scalar, optional*) – Task weight used in IK cost function. If None, needs to be set later.
- **gain** (*scalar, optional*) – Proportional gain of the task.
- **exclude_dofs** (*list of integers, optional*) – DOF indices not used by task.

update_target(*target*)
Update the task residual with a new target.

Parameters **target** (*Point or array or list*) – New link position target.

class pymanoid.tasks.**PoseTask**(*robot*, *link*, *target*, *weight=None*, *gain=None*, *exclude_dofs=None*)
Pose task for a given link.

Parameters

- **robot** (*Robot*) – Target robot.
- **link** (*Link or string*) – Robot Link, or name of the link field in the `robot` argument.
- **target** (*list or array (shape=(7,)) or pymanoid.Body*) – Pose coordinates of the link's target.
- **weight** (*scalar, optional*) – Task weight used in IK cost function. If None, needs to be set later.
- **gain** (*scalar, optional*) – Proportional gain of the task.
- **exclude_dofs** (*list of integers, optional*) – DOF indices not used by task.

update_target(*target*)
Update the task residual with a new target.

Parameters **target** (*Point or array or list*) – New link position target.

class pymanoid.tasks.**PostureTask**(*robot*, *q_ref*, *weight=None*, *gain=None*, *exclude_dofs=None*)
Track a set of reference joint angles, a common choice to regularize the weighted IK problem.

Parameters

- **robot** (*Robot*) – Target robot.
- **q_ref** (*array*) – Vector of reference joint angles.
- **weight** (*scalar, optional*) – Task weight used in IK cost function. If None, needs to be set later.
- **gain** (*scalar, optional*) – Proportional gain of the task.
- **exclude_dofs** (*list of integers, optional*) – DOF indices not used by task.

class pymanoid.tasks.**Task**(*weight=None*, *gain=None*, *exclude_dofs=None*)
Generic IK task.

Parameters

- **weight** (*scalar, optional*) – Task weight used in IK cost function. If None, needs to be set later.
- **gain** (*scalar, optional*) – Proportional gain of the task.
- **exclude_dofs** (*list of integers, optional*) – DOF indices not used by task.

Note: See this tutorial on inverse kinematics for an introduction to the concepts used here.

---

cost(*dt*)
Compute the weighted norm of the task residual.

Parameters **dt** (*scalar*) – Time step in [s].

Returns **cost** – Current cost value.

Return type scalar

exclude_dofs(*dofs*)
Exclude additional DOFs from being used by the task.

jacobian()
Compute the Jacobian matrix of the task.

Returns **J** – Jacobian matrix of the task.

Return type array

residual(*dt*)
Compute the residual of the task.

Returns **r** – Residual vector of the task.

Return type array

Notes

Residuals returned by the `residual` function must have the unit of a velocity. For instance, $\dot{q}$ and $(q_1 - q_2)/dt$ are valid residuals, but $\frac{1}{2}q$ is not.

4.2 Solver

The IK solver is the numerical optimization program that converts task targets and the current robot state to joint motions. In pymanoid, joint motions are computed as velocities that are integrated forward during each simulation cycle (other IK solvers may compute acceleration or jerk values, which are then integrated twice or thrice respectively).

class pymanoid.ik.**IKSolver**(*robot*, *active_dofs=None*, *doflim_gain=0.5*)
Compute velocities bringing the system closer to fulfilling a set of tasks.

Parameters

- **robot** (*Robot*) – Robot to be updated.
- **active_dofs** (*list of integers, optional*) – List of DOFs updated by the IK solver.
- **doflim_gain** (*scalar, optional*) – DOF-limit gain as described in [Kanoun12]. In this implementation, it should be between zero and one.

doflim_gain
DOF-limit gain as described in [Kanoun12]. In this implementation, it should be between zero and one.

Type scalar, optional

lm_damping
Add Levenberg-Marquardt damping as described in [Sugihara11]. This damping significantly improves numerical stability, but convergence gets slower when its value is too high.

Type scalar

slack_dof_limits
Add slack variables to maximize DOF range? This method is used in [Nozawa16] to keep joint angles as far away from their limits as possible. It slows down computations as there are twice as many optimization variables, but is more numerically stable and won't produce inconsistent constraints. Defaults to False.

Type bool

slack_maximize
Linear cost weight applied when `slack_dof_limits` is True.

Type scalar

slack_regularize
Regularization weight applied when `slack_dof_limits` is True.

Type scalar

qd
Velocity returned by last solver call.

Type array

robot
Robot model.

Type pymanoid.Robot

tasks
Dictionary of active IK tasks, indexed by task name.

Type dict

Notes

One unsatisfactory aspect of the DOF-limit gain is that it slows down the robot when approaching DOF limits. For instance, it may slow down a foot motion when approaching the knee singularity, despite the robot being able to move faster with a fully extended knee.

add(*task*)
Add a new task to the IK solver.

Parameters **task** (Task) – New task to add to the list.

build_qp_matrices(*dt*)
Build matrices of the quatratic program.

Parameters **dt** (*scalar*) – Time step in [s].

Returns

- **P** (*(n, n) array*) – Positive semi-definite cost matrix.
- **q** (*array*) – Cost vector.
- **qd_max** (*array*) – Maximum joint velocity vector.
- **qd_min** (*array*) – Minimum joint velocity vector.

Notes

When the robot model has joint acceleration limits, special care should be taken when computing the corresponding velocity bounds for the IK. In short, the robot now needs to avoid the velocity range where it (1) is not going to collide with a DOF limit in one iteration but (2) cannot brake fast enough to avoid a collision in the future due to acceleration limits. This function implements the solution to this problem described in Equation (14) of [Flacco15].

clear()
Clear all tasks in the IK solver.

compute_cost(*dt*)
Compute the IK cost of the present system state for a time step of dt.

Parameters **dt** (*scalar*) – Time step in [s].

compute_velocity(*dt*)
Compute a new velocity satisfying all tasks at best.

Parameters **dt** (*scalar*) – Time step in [s].

Returns **qd** – Vector of active joint velocities.

Return type array

Note: This QP formulation is the default for *pymanoid.ik.IKSolver.solve()* (posture generation) as it converges faster.

Notes

The method implemented in this function is reasonably fast but may become unstable when some tasks are widely infeasible. In such situations, you can either increase the Levenberg-Marquardt bias `self.lm_damping` or set `slack_dof_limits=True` which will call *pymanoid.ik.IKSolver.compute_velocity_with_slack()*.

The returned velocity minimizes squared residuals as in the weighted cost function, which corresponds to the Gauss-Newton algorithm. Indeed, expanding the square expression in `cost(task, qd)` yields

$$\text{minimize } \dot{q} J^T J \dot{q} - 2 r^T J \dot{q}$$

Differentiating with respect to $\dot{q}$ shows that the minimum is attained for $J^T J \dot{q} = r$, where we recognize the Gauss-Newton update rule.

compute_velocity_with_slack(*dt*)

Compute a new velocity satisfying all tasks at best, while trying to stay away from kinematic constraints.

Parameters **dt** (*scalar*) – Time step in [s].

Returns **qd** – Vector of active joint velocities.

Return type array

Note: This QP formulation is the default for *pymanoid.ik.IKSolver.step()* as it has a more numerically-stable behavior.

Notes

Check out the discussion of this method around Equation (10) of [Nozawa16]. DOF limits are better taken care of by slack variables, but the variable count doubles and the QP takes roughly 50% more time to solve.

on_tick(*sim*)

Step the IK at each simulation tick.

Parameters **sim** (Simulation) – Simulation instance.

print_costs(*qd*, *dt*)

Print task costs for the current IK step.

Parameters

- **qd** (*array*) – Robot DOF velocities.
- **dt** (*scalar*) – Timestep for the IK.

remove (*ident*)
Remove a task.

Parameters **ident** (*string or object*) – Name or object with a `name` field identifying the task.

set_active_dofs (*active_dofs*)
Set DOF indices modified by the IK.

Parameters **active_dofs** (*list of integers*) – List of DOF indices.

set_gains (*gains*)
Set task gains from a dictionary.

Parameters **gains** (*string -> double dictionary*) – Dictionary mapping task labels to default gain values.

set_weights (*weights*)
Set task weights from a dictionary.

Parameters **weights** (*string -> double dictionary*) – Dictionary mapping task labels to default weight values.

solve (*max_it=1000*, *cost_stop=1e-10*, *impr_stop=1e-05*, *dt=0.01*, *warm_start=False*, *debug=False*)
Compute joint-angles that satisfy all kinematic constraints at best.

Parameters

- **max_it** (*integer*) – Maximum number of solver iterations.
- **cost_stop** (*scalar*) – Stop when cost value is below this threshold.
- **impr_stop** (*scalar, optional*) – Stop when cost improvement (relative variation from one iteration to the next) is less than this threshold.
- **dt** (*scalar, optional*) – Time step in [s].
- **warm_start** (*bool, optional*) – Set to True if the current robot posture is a good guess for IK. Otherwise, the solver will start by an exploration phase with DOF velocity limits relaxed and no Levenberg-Marquardt damping.
- **debug** (*bool, optional*) – Set to True for additional debug messages.

Returns

- **nb_it** (*int*) – Number of solver iterations.
- **cost** (*scalar*) – Final value of the cost function.

Notes

Good values of *dt* depend on the weights of the IK tasks. Small values make convergence slower, while big values make the optimization unstable (in which case there may be no convergence at all).

step(*dt*)

Apply velocities computed by inverse kinematics.

Parameters **dt** (*scalar*) – Time step in [s].

Acceleration limits

When the robot model has joint acceleration limits, *i.e.* when a vector of size `robot.nb_dofs` is specified in `robot.qdd_lim` at initialization, the inverse kinematics will include them in its optimization problem. The formulation is more complex than a finite-difference approximation: joint velocities will be selected so that (1) the joint does not collide with its position limit in one iteration, but also (2) despite its acceleration limit, it can still brake fast enough to avoid colliding with its position limit in the future. The inverse kinematics in pymanoid implements the solution to this problem described in Equation (14) of [Flacco15].

4.3 Stance

The most common IK tasks for humanoid locomotion are the *COM task* and *end-effector pose task*. The `Stance` class avoids the hassle of creating and adding these tasks one by one. To use it, first create your targets:

```
com_target = robot.get_com_point_mass()
lf_target = robot.left_foot.get_contact(pos=[0, 0.3, 0])
rf_target = robot.right_foot.get_contact(pos=[0, -0.3, 0])
```

Then create the stance and bind it to the robot:

```
stance = Stance(com=com_target, left_foot=lf_target, right_foot=rf_
↪target)
stance.bind(robot)
```

Calling the *bind* function will automatically add the corresponding tasks to the robot IK solver. See the inverse_kinematics.py example for more details.

class pymanoid.stance.**Stance**(*com*, *left_foot=None*, *right_foot=None*, *left_hand=None*, *right_hand=None*)

Set of contact and center of mass (COM) targets for the humanoid.

Parameters

- **com** (`PointMass`) – Center of mass target.
- **left_foot** (`Contact, optional`) – Left-foot contact target.

- **right_foot** (*Contact, optional*) – Right-foot contact target.
- **left_hand** (*Contact, optional*) – Left-hand contact target.
- **right_hand** (*Contact, optional*) – Right-hand contact target.

bind(*robot*, *reg='posture'*)

Bind stance as robot IK targets.

Parameters

- **robot** (*pymanoid.Robot*) – Target robot.
- **reg** (*string, optional*) – Regularization task, either "posture" or "min_vel".

property bodies

List of end-effector and COM bodies.

compute_pendular_accel_cone(*com_vertices=None*, *zdd_max=None*, *reduced=False*)

Compute the pendular COM acceleration cone of the stance.

The pendular cone is the reduction of the Contact Wrench Cone when the angular momentum at the COM is zero.

Parameters

- **com_vertices** (*list of (3,) arrays, optional*) – Vertices of a COM bounding polytope.
- **zdd_max** (*scalar, optional*) – Maximum vertical acceleration in the output cone.
- **reduced** (*bool, optional*) – If `True`, returns the reduced 2D form rather than a 3D cone.

Returns **vertices** – List of 3D vertices of the (truncated) COM acceleration cone, or of the 2D vertices of the reduced form if `reduced` is `True`.

Return type list of (3,) arrays

Notes

The method is based on a rewriting of the CWC formula, followed by a 2D convex hull on dual vertices. The algorithm is described in [Caron16].

When `com` is a list of vertices, the returned cone corresponds to COM accelerations that are feasible from *all* COM located inside the polytope. See [Caron16] for details on this conservative criterion.

compute_static_equilibrium_polygon (*method='hull'*)
Compute the halfspace and vertex representations of the static-equilibrium polygon (SEP) of the stance.

Parameters **method** (*string, optional*) – Which method to use to perform the projection. Choices are 'bretl', 'cdd' and 'hull' (default).

compute_zmp_support_area (*height*, *method='bretl'*)
Compute an extension of the (pendular) multi-contact ZMP support area with optional pressure limits on each contact.

Parameters

- **height** (*array, shape=(3,)*) – Height at which the ZMP support area is computed.
- **method** (*string, default='bretl'*) – Polytope projection algorithm, can be `"bretl"` or `"cdd"`.

Returns **vertices** – Vertices of the ZMP support area.

Return type list of arrays

Notes

There are two polytope projection algorithms: 'bretl' is adapted from in [Bretl08] while 'cdd' corresponds to the double-description formulation from [Caron17z]. See the Appendix from [Caron16] for a performance comparison.

property contacts
List of active contacts.

dist_to_sep_edge (*com*)
Algebraic distance of a COM position to the edge of the static-equilibrium polygon.

Parameters **com** (*array, shape=(3,)*) – COM position to evaluate the distance from.

Returns **dist** – Algebraic distance to the edge of the polygon. Inner points get a positive value, outer points a negative one.

Return type scalar

find_static_supporting_wrenches ()
Find supporting contact wrenches in static-equilibrium.

Returns **support** – Mapping between each contact *i* in the contact set and a supporting contact wrench $w^i_{C_i}$.

Return type list of (*Contact*, array) couples

free_contact (*effector_name*)
Free a contact from the stance and get the corresponding end-effector target. Its

task stays in the IK, but the effector is not considered in contact any more (e.g. for contact wrench computations).

Parameters **effector_name** (*string*) – Name of end-effector target, for example "left_foot".

Returns **effector** – IK target contact.

Return type pymanoid.Contact

static from_json(*path*)
Create a new stance from a JSON file.

Parameters **path** (*string*) – Path to the JSON file.

hide()
Hide end-effector and COM body handles.

load(*path*)
Load stance from JSON file.

Parameters **path** (*string*) – Path to the JSON file.

property nb_contacts
Number of active contacts.

save(*path*)
Save stance into JSON file.

Parameters **path** (*string*) – Path to JSON file.

set_contact(*effector*)
Set contact on a effector.

Parameters **effector** (*pymanoid.Contact*) – IK target contact.

show()
Show end-effector and COM body handles.

4.4 Example

In this example, we will see how to put the humanoid robot model in a desired configuration. Let us initialize a simulation with a 30 ms timestep and load the JVRC-1 humanoid:

```
sim = pymanoid.Simulation(dt=0.03)
robot = JVRC1('JVRC-1.dae', download_if_needed=True)
sim.set_viewer()  # open GUI window
```

We define targets for foot contacts;

```
lf_target = Contact(robot.sole_shape, pos=[0, 0.3, 0], visible=True)
rf_target = Contact(robot.sole_shape, pos=[0, -0.3, 0],␣
↪visible=True)
```

Next, let us set the altitude of the robot's free-flyer (attached to the waist link) 80 cm above contacts. This is useful to give the IK solver a good initial guess and avoid coming out with a valid but weird solution in the end.

```
robot.set_dof_values([0.8], dof_indices=[robot.TRANS_Z])
```

This being done, we initialize a point-mass that will serve as center-of-mass (COM) target for the IK. Its initial position is set to `robot.com`, which will be roughly 80 cm above contacts as it is close to the waist link:

```
com_target = PointMass(pos=robot.com, mass=robot.mass)
```

All our targets being defined, we initialize IK tasks for the feet and COM position, as well as a posture task for the (necessary) regularization of the underlying QP problem:

```
lf_task = ContactTask(robot, robot.left_foot, lf_target, weight=1.)
rf_task = ContactTask(robot, robot.right_foot, rf_target, weight=1.)
com_task = COMTask(robot, com_target, weight=1e-2)
reg_task = PostureTask(robot, robot.q, weight=1e-6)
```

We initialize the robot's IK using all DOFs, and insert our tasks:

```
robot.init_ik(active_dofs=robot.whole_body)
robot.ik.add_task(lf_task)
robot.ik.add_task(rf_task)
robot.ik.add_task(com_task)
robot.ik.add_task(reg_task)
```

We can also throw in some extra DOF tasks for a nicer posture:

```
robot.ik.add_task(DOFTask(robot, robot.R_SHOULDER_R, -0.5, gain=0.5,
↪ weight=1e-5))
robot.ik.add_task(DOFTask(robot, robot.L_SHOULDER_R, +0.5, gain=0.5,
↪ weight=1e-5))
```

Finally, call the IK solver to update the robot posture:

```
robot.ik.solve(max_it=100, impr_stop=1e-4)
```

The resulting posture looks like this:

Alternatively, rather than creating and adding all tasks one by one, we could have used the *Stance* interface:

```
stance = Stance(com=com_target, left_foot=lf_target, right_foot=rf_
↪target)
stance.bind(robot)
robot.ik.add_task(DOFTask(robot, robot.R_SHOULDER_R, -0.5, gain=0.5,
↪ weight=1e-5))
robot.ik.add_task(DOFTask(robot, robot.L_SHOULDER_R, +0.5, gain=0.5,
↪ weight=1e-5))
robot.ik.solve(max_it=100, impr_stop=1e-4)
```

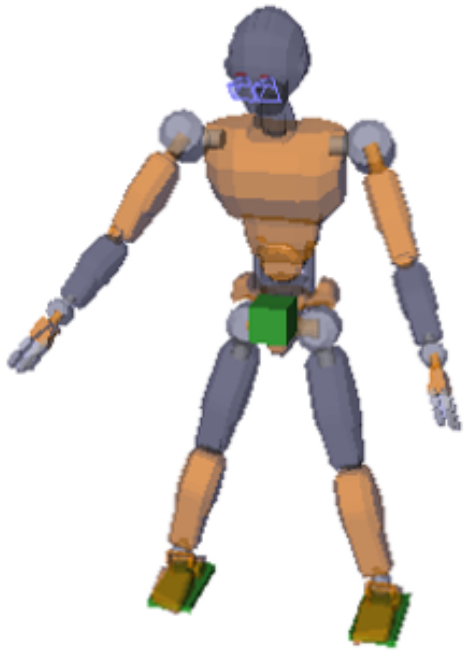


Fig. 1: Robot posture found by inverse kinematics.

This code is more concise and yields the same result.

5 Contact stability

5.1 Contact

class pymanoid.contact.**Contact**(*shape*, *pos=None*, *rpy=None*, *pose=None*, *friction=None*, *link=None*, *slab_thickness=0.01*)

Rectangular contact surface.

Parameters

- **shape** (*(scalar, scalar)*) – Surface dimensions (half-length, half-width) in [m].
- **pos** (*ndarray*) – Contact position in world frame.
- **rpy** (*ndarray*) – Contact orientation in world frame.
- **pose** (*ndarray*) – Initial pose. Supersedes `pos` and `rpy` if they are provided at the same time.
- **friction** (*scalar*) – Static friction coefficient.
- **link** (`body.Manipulator`, *optional*) – Robot link frame in contact.

- **slab_thickness** (*scalar, optional*) – Thickness of the contact slab displayed in the GUI, in [m].
- **max_pressure** (*scalar, optional*) – Maximum pressure on contact.

wrench
Contact wrench coordinates at the contact point in the inertial frame.

Type array, shape=(6,)

compute_grasp_matrix(*p*)
Compute the grasp matrix for a given destination point.

The grasp matrix G_P converts the local contact wrench *w* to the contact wrench w_P at another point *P*:

$$w_P = G_P w$$

All wrench coordinates being taken in the world frame.

Parameters **p** (*array, shape=(3,)*) – Point, in world frame coordinates, where the wrench is taken.

Returns **G** – Grasp matrix G_P.

Return type ndarray

copy(*link=None*, *hide=False*)
Return a copy of the contact.

Parameters

- **link** (body.Manipulator, *optional*) – Robot link frame in contact in the copy.
- **hide** (*bool, optional*) – Hide copy?

property force
Resultant of contact forces in the world frame (if defined).

property force_inequalities
Matrix of force friction cone inequalities in the world frame.

Notes

All linearized friction cones in pymanoid use the inner (conservative) approximation. See for instance this introduction to friction cones for details.

property force_rays
Rays of the force friction cone in the world frame.

Notes

All linearized friction cones in pymanoid use the inner (conservative) approximation. See for instance this introduction to friction cones for details.

property force_span
Span matrix of the force friction cone in world frame.

This matrix S is such that all valid contact forces can be written:

$$f = S\lambda, \quad \lambda \geq 0$$

Notes

All linearized friction cones in pymanoid use the inner (conservative) approximation. See for instance this introduction to friction cones for details.

get_scaled_contact_area(*scale*)
Get the vertices of the scaled contact area.

Parameters **scale** (*scalar*) – Contact area is scaled by this ratio.

Returns **vertices** – List of vertex coordinates in the world frame.

Return type list of arrays

property moment
Moment of contact forces in the world frame (if defined).

set_wrench(*wrench*)
Set contact wrench directly.

Parameters **wrench** (*array, shape=(6,)*) – Wrench coordinates given in the contact frame.

Notes

This function switches the contact to "managed" mode, as opposed to the default "supporting" mode where the wrench distributor finds contact wrenches by numerical optimization.

unset_wrench()
Return contact to supporting mode.

property vertices
Vertices of the contact area.

wrench_at(*point*)
Get contact wrench at a given point in the world frame.

Parameters **point** (*array, shape=(3,)*) – Point P where the wrench is expressed.

Returns **wrench** – Contact wrench w_P at P in the world frame.

Return type array, shape=(6,)

property wrench_hrep

H-representation of friction inequalities (and optional pressure limits) in world frame.

This matrix-vector pair describes the linearized Coulomb friction model (in the fixed contact mode) and pressure limits by:

$$Fw \leq g$$

where w is the contact wrench at the contact point (`self.p`) in the world frame. See [Caron15] for the derivation of the formula for F.

property wrench_inequalities

Matrix F of friction inequalities in world frame.

This matrix describes the linearized Coulomb friction model (in the fixed contact mode) by:

$$Fw \leq 0$$

where w is the contact wrench at the contact point (`self.p`) in the world frame. See [Caron15] for the derivation of the formula for F.

property wrench_rays

Rays (V-rep) of the contact wrench cone in world frame.

property wrench_span

Span matrix of the contact wrench cone in world frame.

This matrix is such that all valid contact wrenches can be written as:

$$w_P = S\lambda, \quad \lambda \geq 0$$

where S is the friction span and λ is a vector with positive coordinates.

Returns **S** – Span matrix of the contact wrench cone.

Return type array, shape=(6, 16)

Notes

Note that the contact wrench coordinates w_P ("output" of S) are taken at the contact point P (`self.p`) and in the world frame. Meanwhile, the number of columns of S results from our choice of 4 contact points (one for each vertex of the rectangular area) with 4-sided friction pyramids at each.

5.2 Multiple contacts

class pymanoid.contact.**ContactSet**(*contacts=None*)

compute_grasp_matrix(*p*)

Compute the grasp matrix of all contact wrenches at point p.

Parameters **p** (*array, shape=(3,)*) – Point where the resultant wrench is taken at.

Returns **G** – Grasp matrix giving the resultant contact wrench w_P of all contact wrenches as $w_P = G w_{all}$, with w_{all} the stacked vector of contact wrenches (each wrench being taken at its respective contact point and in the world frame).

Return type array, shape=(6, m)

compute_static_equilibrium_polygon(*method='hull'*)

Compute the static-equilibrium polygon of the center of mass.

Parameters **method** (*string, optional*) – Choice between 'bretl', 'cdd' or 'hull'.

Returns **vertices** – 2D vertices of the static-equilibrium polygon.

Return type list of arrays

Notes

The method 'bretl' is adapted from in [Bretl08] where the static-equilibrium polygon was introduced. The method 'cdd' corresponds to the double-description approach described in [Caron17z]. See the Appendix from [Caron16] for a performance comparison.

compute_wrench_inequalities(*p*)

Compute the matrix of wrench cone inequalities in the world frame.

Parameters **p** (*array, shape=(3,)*) – Point where the resultant wrench is taken at.

Returns **F** – Friction matrix such that all valid contact wrenches satisfy $Fw \leq 0$, where w is the resultant contact wrench at p.

Return type array, shape=(m, 6)

compute_wrench_span(*p*)

Compute the span matrix of the contact wrench cone in world frame.

Parameters **p** (*array, shape=(3,)*) – Point where the resultant-wrench coordinates are taken.

Returns **S** – Span matrix of the net contact wrench cone.

Return type array, shape=(6, m)

Notes

The span matrix S_P such that all valid contact wrenches can be written as:

$$w_P = S_P \lambda, \quad \lambda \geq 0$$

where w_P denotes the contact-wrench coordinates at point *P*.

find_supporting_wrenches(*wrench*, *point*, *friction_weight=0.01*, *cop_weight=1.0*, *yaw_weight=0.0001*, *solver='quadprog'*)

Find supporting contact wrenches for a given net contact wrench.

Parameters

- **wrench** (*array, shape=(6,)*) – Resultant contact wrench w_P to be realized.
- **point** (*array, shape=(3,)*) – Point *P* where the wrench is expressed.
- **friction_weight** (*scalar, optional*) – Weight on friction minimization.
- **cop_weight** (*scalar, optional*) – Weight on COP deviations from the center of the contact patch.
- **solver** (*string, optional*) – Name of the QP solver to use. Options are 'quadprog' (default) or 'cvxopt'. The latter is slower but more numerically stable if your resulting wrenches are extremal.

Returns support – Mapping between each contact *i* and a supporting contact wrench $w^i_{C_i}$. Contact wrenches satisfy friction constraints and sum up to the net wrench: $\sum_c w^i_P = w_P$.

Return type list of (*Contact*, array) pairs

Notes

Wrench coordinates are returned in their respective contact frames ($w^i_{C_i}$), not at the point *P* where the net wrench w_P is given.

property supporting_contacts

Set of supporting contacts, i.e. excluding managed contacts where the user provides the external wrench.

5.3 Computing contact forces

Contact wrenches exerted on the robot while it moves can be computed by quadratic programming in the *wrench distributor* of a stance. This process is automatically created when binding a `Stance` to the robot model. It will only be executed if you schedule it to your simulation. Here is a small example:

```
from pymanoid import robots, Simulation, Stance

sim = Simulation(dt=0.03)
robot = robots.JVRC1(download_if_needed=True)
stance = Stance(
    com=robot.get_com_point_mass(),
    left_foot=robot.left_foot.get_contact(pos=[0, 0.3, 0]),
    right_foot=robot.right_foot.get_contact(pos=[0, -0.3, 0]))
stance.com.set_z(0.8)
stance.bind(robot)
sim.schedule(robot.ik)
sim.schedule(robot.wrench_distributor)
sim.start()
```

You can see the computed wrenches in the GUI by scheduling the corresponding drawer process:

```
from pymanoid.gui import RobotWrenchDrawer

sim.set_viewer()
sim.schedule_extra(RobotWrenchDrawer(robot))
```

Once the wrench distributor is scheduled, it will store its outputs in the contacts of the stance as well as in the robot's manipulators. Therefore, you can access `robot.left_foot.wrench` or `robot.stance.left_foot.wrench` equivalently. Note that wrenches are given in the *world frame* rooted at their respective contact points.

class pymanoid.stance.**StanceWrenchDistributor**(*stance*)

Wrench distribution process.

Parameters **stance** (Stance) – Stance to distribute wrenches from.

Notes

This process computes wrenches for supporting contacts and stores them in each contact, as well as in the robot's manipulators. For instance, you will be able to access `robot.left_foot.wrench` or `robot.stance.left_foot.wrench` equivalently. Note that supporting wrenches are given in the world frame rooted at their respective contacts.

on_tick(*sim*)

Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

6 Walking pattern generation

Walking pattern generation converts a high-level objective such as "go through this sequence of footsteps" to a time-continuous joint trajectory. For position-controlled robots such as those simulated in pymanoid, joint trajectories are computed by inverse kinematics from intermediate task targets. The two main targets for walking are the swing foot and center of mass.

6.1 Swing foot trajectory

The foot in the air during a single-support phase is called the *swing foot*. Its trajectory can be implemented by spherical linear interpolation for the orientation and polynomial interpolation for the position of the foot in the air.

class pymanoid.swing_foot.**SwingFoot**(*start_contact*, *end_contact*, *duration*, *takeoff_clearance=0.05*, *landing_clearance=0.05*)

Polynomial swing foot interpolator.

Parameters

- **start_contact** (*pymanoid.Contact*) – Initial contact.
- **end_contact** (*pymanoid.Contact*) – Target contact.
- **duration** (*scalar*) – Swing duration in [s].
- **takeoff_clearance** (*scalar, optional*) – Takeoff clearance height at 1/4th of the interpolated trajectory.
- **landing_clearance** (*scalar, optional*) – Landing clearance height at 3/4th of the interpolated trajectory.

draw(*color='r'*)

Draw swing foot trajectory.

Parameters **color** (*char or triplet, optional*) – Color letter or RGB values, default is 'b' for blue.

integrate(*dt*)

Integrate swing foot motion forward by a given amount of time.

Parameters **dt** (*scalar*) – Duration of forward integration, in [s].

interpolate()

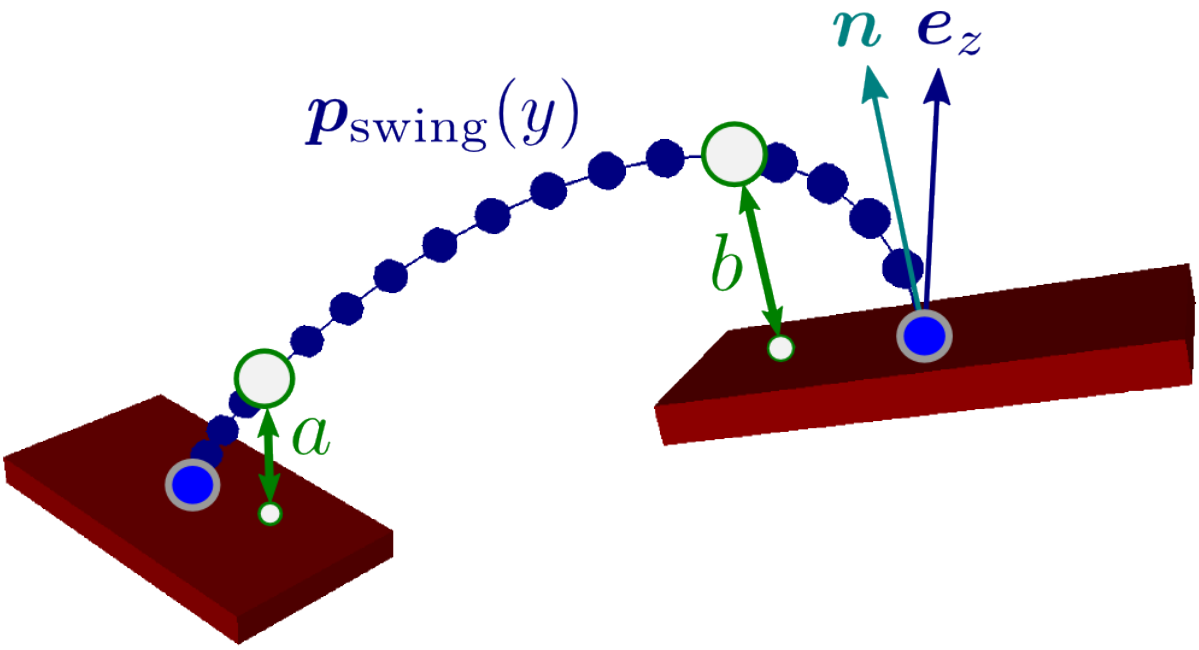
Interpolate swing foot path.

Returns **path** – Polynomial path with index between 0 and 1.

Return type pymanoid.NDPolynomial

Notes

A swing foot trajectory comes under two conflicting imperatives: stay close to the ground while avoiding collisions. We assume here that the terrain is uneven but free from obstacles. To avoid robot-ground collisions, we enforce two clearance distances: one at toe level for the takeoff contact and the other at heel level for the landing contact.



We interpolate swing foot trajectories as cubic Hermite curves with four boundary constraints: initial and final positions corresponding to contact centers, and tangent directions parallel to contact normals. This leaves two free parameters, corresponding to the norms of the initial and final tangents, that can be optimized upon. It can be shown that minimizing these two parameters subject to the clearance conditions `a > a_min` and `b > b_min` is a small constrained least squares problem.

6.2 Linear model predictive control

Linear model predictive control [Wieber06] generates a dynamically-consistent trajectory for the center of mass (COM) while walking. It applies to walking over a flat floor, where the assumptions of the linear inverted pendulum mode (LIPM) [Kajita01] can be applied.

class pymanoid.mpc.**LinearPredictiveControl**(*A*, *B*, *C*, *D*, *e*, *x_init*, *x_goal*, *nb_steps*, *wxt=None*, *wxc=None*, *wu=0.001*)

Predictive control for a system with linear dynamics and linear constraints.

The discretized dynamics of a linear system are described by:

$$x_{k+1} = Ax_k + Bu_k$$

where x is assumed to be the first-order state of a configuration variable p, i.e., it stacks both the position p and its time-derivative $\dot{p}$. Meanwhile, the system is linearly constrained by:

$$x_0 = x_{\text{init}}$$
$$\forall k,\ C_k x_k + D_k u_k \leq e_k$$

The output control law minimizes a weighted combination of two types of costs:

- **Terminal state error** $\|x_{\text{nb\_steps}} - x_{\text{goal}}\|^2$ with weight w_{xt}.
- **Cumulated state error:** $\sum_k \|x_k - x_{\text{goal}}\|^2$ with weight w_{xc}.
- **Cumulated control costs:** $\sum_k \|u_k\|^2$ with weight w_u.

Parameters

- **A** (*array, shape=(n, n)*) – State linear dynamics matrix.
- **B** (*array, shape=(n, dim(u))*) – Control linear dynamics matrix.
- **x_init** (*array, shape=(n,)*) – Initial state as stacked position and velocity.
- **x_goal** (*array, shape=(n,)*) – Goal state as stacked position and velocity.
- **nb_steps** (*int*) – Number of discretization steps in the preview window.
- **C** (*array, shape=(m, dim(u)), list of arrays, or None*) – Constraint matrix on state variables. When this argument is an array, the same matrix *C* is applied at each step *k*. When it is `None`, the null matrix is applied.
- **D** (*array, shape=(l, n), or list of arrays, or None*) – Constraint matrix on control variables. When this argument is an array, the same matrix *D* is applied at each step *k*. When it is `None`, the null matrix is applied.
- **e** (*array, shape=(m,), list of arrays*) – Constraint vector. When this argument is an array, the same vector *e* is applied at each step *k*.
- **wxt** (*scalar, optional*) – Weight on terminal state cost, or `None` to disable.
- **wxc** (*scalar, optional*) – Weight on cumulated state costs, or `None` to disable (default).
- **wu** (*scalar, optional*) – Weight on cumulated control costs.

Notes

In numerical analysis, there are three classes of methods to solve boundary value problems: single shooting, multiple shooting and collocation. The solver implemented in this class follows the single shooting method.

property X
Series of system states over the preview window.

Note: This property is only available after `solve()` has been called.

---

build()
Compute internal matrices defining the preview QP.

Notes

See [Audren14] for details on the matrices Φ and Ψ, as we use similar notations below.

solve(***kwargs*)
Compute the series of controls that minimizes the preview QP.

Parameters

- **solver** (*string, optional*) – Name of the QP solver in `qpsolvers.available_solvers`.
- **initvals** (*array, optional*) – Vector of initial U values used to warm-start the QP solver.

property solve_and_build_time
Total computation time taken by MPC computations.

6.3 Nonlinear predictive control

The assumptions of the LIPM are usually too restrictive to walk over uneven terrains. In this case, one can turn to more general (nonlinear) model predictive control. A building block for nonlinear predictive control of the center of mass is implemented in the *pymanoid.centroidal.COMStepTransit* class, where a COM trajectory is constructed from one footstep to the next by solving a nonlinear program (NLP).

class pymanoid.centroidal.**COMStepTransit**(*desired_duration*, *start_com*, *start_comd*, *dcm_target*, *foothold*, *next_foothold*, *omega2*, *nb_steps*, *nlp_options=None*)
Compute a COM trajectory that transits from one footstep to the next. This solution is applicable over arbitrary terrains.

Implements a short-sighted optimization with a DCM boundedness constraint, defined within the floating-base inverted pendulum (FIP) of constant ω. This approach is used in the nonlinear predictive controller of [Caron17w] for dynamic walking over rough terrains.

Parameters

- **desired_duration** (*scalar*) – Desired duration of the transit trajectory.
- **start_com** (*(3,) array*) – Initial COM position.
- **start_comd** (*(3,) array*) – Initial COM velocity.
- **foothold** (Contact) – Location of the stance foot contact.
- **omega** (*scalar*) – Constant of the floating-base inverted pendulum.
- **dcm_target** (*(3,) array*) – Desired terminal divergent-component of COM motion.
- **nb_steps** (*int*) – Number of discretization steps.

Notes

The boundedness constraint in this optimization makes up for the short-sighted preview [Scianca16], as opposed to the long-sighted approach to walking [Wieber06] where COM boundedness results from the preview of several future steps.

The benefit of using ZMP controls rather than COM accelerations lies in the exact integration scheme. This phenomenon appears in flat-floor models as well: in the CART-table model, segments with fixed COM acceleration imply variable ZMPs that may exit the support area (see e.g. Figure 3.17 in [ElKhoury13]).

add_com_height_constraint(*p*, *ref_height*, *max_dev*)

Constraint the COM to deviate by at most *max_dev* from *ref_height*.

Parameters

- **ref_height** (*scalar*) – Reference COM height.
- **max_dev** (*scalar*) – Maximum distance allowed from the reference.

Note: The height is measured along the z-axis of the contact frame here, not of the world frame's.

---

add_friction_constraint(*p*, *z*, *foothold*)

Add a circular friction cone constraint for a COM located at *p* and a (floating-base) ZMP located at *z*.

Parameters

- **p** (*casadi.MX*) – Symbol of COM position variable.
- **z** (*casadi.MX*) – Symbol of ZMP position variable.

add_linear_cop_constraints(*p*, *z*, *foothold*, *scaling=0.95*)
Constraint the COP, located between the COM *p* and the (floating-base) ZMP *z*, to lie inside the contact area.

Parameters

- **p** (*casadi.MX*) – Symbol of COM position variable.
- **z** (*casadi.MX*) – Symbol of ZMP position variable.
- **scaling** (*scalar, optional*) – Scaling factor between 0 and 1 applied to the contact area.

build()
Build the internal nonlinear program (NLP).

draw(*color='b'*)
Draw the COM trajectory.

Parameters **color** (*char or triplet, optional*) – Color letter or RGB values, default is 'b' for green.

Returns **handle** – OpenRAVE graphical handle. Must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type openravepy.GraphHandle

print_results()
Print various statistics on NLP resolution.

solve()
Solve the nonlinear program and store the solution, if found.

7 Numerical optimization

7.1 Quadratic programming

Quadratic programming is a class of numerical optimization problems that can be solved efficiently. It is used for instance by the IK solver to solve whole-body control by inverse kinematics.

pymanoid.qpsolvers.**solve_qp**(*P*, *q*, *G=None*, *h=None*, *A=None*, *b=None*, *solver='quadprog'*, *initvals=None*, *sym_proj=False*)
Solve a Quadratic Program defined as:

minimize (1/2) * x.T * P * x + q.T * x

subject to G * x <= h A * x == b

using one of the available QP solvers.

Parameters

- **P** (*numpy.array, scipy.sparse.csc_matrix or cvxopt.spmatrix*) – Symmetric quadratic-cost matrix (most solvers require it to be definite as well).
- **q** (*numpy.array*) – Quadratic-cost vector.
- **G** (*numpy.array, scipy.sparse.csc_matrix or cvxopt.spmatrix*) – Linear inequality matrix.
- **h** (*numpy.array*) – Linear inequality vector.
- **A** (*numpy.array, scipy.sparse.csc_matrix or cvxopt.spmatrix*) – Linear equality matrix.
- **b** (*numpy.array*) – Linear equality vector.
- **solver** (*string, optional*) – Name of the QP solver, to choose in `qpsolvers.available_solvers`.
- **initvals** (*array, optional*) – Vector of initial *x* values used to warm-start the solver.
- **sym_proj** (*bool, optional*) – Set to *True* when the *P* matrix provided is not symmetric.

Returns **x** – Optimal solution if found, None otherwise.

Return type array or None

Notes

In quadratic programming, the matrix *P* should be symmetric. Many solvers (including CVXOPT, OSQP and quadprog) leverage this property and may return erroneous results when it is not the case. You can set `sym_proj=True` to project *P* on its symmetric part, at the cost of some computation time.

7.2 Nonlinear programming

Nonlinear programming is a catch-all expression for numerical optimization problems that don't have any particular structure, such as *convexity*, allowing them to be solved more efficiently by dedicated methods. It is used for instance in nonlinear model predictive control to compute locomotion trajectories over uneven terrains.

class pymanoid.nlp.**NonlinearProgram**(*solver='ipopt'*, *options=None*)

Wrapper around CasADi to formulate and solve nonlinear optimization problems.

Parameters

- **solver** (*string, optional*) – Solver name. Use 'ipopt' (default) for an interior point method or 'sqpmethod' for sequential quadratic programming.

- **options** (*dict, optional*) – Dictionary of solver options.

add_constraint (*expr*, *lb*, *ub*, *name=None*)
Add a new constraint to the problem.

Parameters

- **expr** (*casadi.MX*) – Python or CasADi symbolic expression.
- **lb** (*array*) – Lower-bound on the expression.
- **ub** (*array*) – Upper-bound on the expression.
- **name** (*string, optional*) – If provided, will stored the expression under this name for future updates.

add_equality_constraint (*expr1*, *expr2*, *name=None*)
Add an equality constraint between two expressions.

Parameters

- **expr1** (*casadi.MX*) – Expression on problem variables.
- **expr2** (*casadi.MX*) – Expression on problem variables.
- **name** (*string, optional*) – If provided, will stored the expression under this name for future updates.

casadi_expand = True
Replace MX with SX expressions in problem formulation.

Note: Setting this option to `True` seems to significantly improve computation times when using IPOPT.

---

create_solver ()
Create a new nonlinear solver.

extend_cost (*expr*)
Add a new expression term to the cost function of the problem.

Parameters **expr** (*casadi.MX*) – Python or CasADi symbolic expression.

has_constraint (*name*)
Check if a given name identifies a problem constraint.

ipopt_fast_step_computation = 'yes'
If set to yes, the algorithm assumes that the linear system that is solved to obtain the search direction, is solved sufficiently well. In that case, no residuals are computed, and the computation of the search direction is a little faster.

ipopt_fixed_variable_treatment = 'relax_bounds'
Default is "make_parameter", but "relax_bounds" seems to make computations faster and numerically stabler.

ipopt_linear_solver = 'ma27'
Linear solver used for step computations.

ipopt_max_cpu_time = 10.0
Maximum number of CPU seconds. Note that this parameter corresponds to processor time, not wall time. For a CPU with N cores, the latter can be as much as N times lower than the former.

ipopt_max_iter = 1000
Maximum number of iterations.

ipopt_mu_strategy = 'adaptive'
"monotone" (default) or "adaptive".

Type Update strategy for barrier parameter

ipopt_nlp_lower_bound_inf = -9000000000.0
Any bound below this value will be considered -inf, i.e. not lower bounded.

ipopt_nlp_upper_bound_inf = 9000000000.0
Any bound above this value will be considered +inf, i.e. not upper bounded.

ipopt_print_level = 0
Output verbosity level between 0 and 12.

ipopt_print_time = False
Print detailed solver computation times.

ipopt_warm_start_init_point = 'yes'
Indicates whether the optimization should use warm start initialization, where values of primal and dual variables are given (e.g. from a previous optimization of a related problem).

property iter_count
Number of LP solver iterations applied by the NLP solver.

new_constant(*name*, *dim*, *value*)
Add a new constant to the problem.

Parameters

- **name** (*string*) – Name of the constant.
- **dim** (*int*) – Number of dimensions.
- **value** (*array, shape=(dim,)*) – Value of the constant.

Note: Constants are implemented as variables with matching lower and upper bounds.

---

new_variable(*name*, *dim*, *init*, *lb*, *ub*)
Add a new variable to the problem.

Parameters

- **name** (*string*) – Variable name.
- **dim** (*int*) – Number of dimensions.
- **init** (*array, shape=(dim,)*) – Initial values.
- **lb** (*array, shape=(dim,)*) – Vector of lower bounds on variable values.
- **ub** (*array, shape=(dim,)*) – Vector of upper bounds on variable values.

property optimal_found
True if and only if the solution is a local optimum.

property return_status
String containing a status message from the NLP solver.

solve()
Call the nonlinear solver.

Returns **x** – Vector of variable coordinates for the best solution found.

Return type array

property solve_time
Time (in seconds) taken to solve the problem.

update_constant(*name*, *value*)
Update the value of an existing constant.

Parameters

- **name** (*string*) – Name of the constant.
- **value** (*array, shape=(dim,)*) – Vector of new values for the constant.

update_constraint_bounds(*name*, *lb*, *ub*)
Update lower- and upper-bounds on an existing constraint.

Parameters

- **name** (*string*) – Identifier of the constraint.
- **lb** (*array*) – New lower-bound of the constraint.
- **ub** (*array*) – New upper-bound of the constraint.

update_variable_bounds(*name*, *lb*, *ub*)
Update the lower- and upper-bounds on an existing variable.

Parameters

- **name** (*string*) – Name of the variable.
- **lb** (*array, shape=(dim,)*) – Vector of lower bounds on variable values.

- **ub** (*array, shape=(dim,)*) – Vector of upper bounds on variable values.

warm_start (*initvals*)
Warm-start the problem with a new vector of initial values.

Parameters **initvals** (*array*) – Vector of initial values for all problem variables.

8 Simulation environment

8.1 Processes

Simulations schedule calls to a number of "processes" into an infinite loop, which represents the control loop of the robot. A process is a simple wrapper around an `on_tick()` function, which is called at each iteration of the simulation loop.

class pymanoid.proc.**Process**
Processes implement the `on_tick` method called by the simulation.

on_tick (*sim*)
Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

pause ()
Stop calling the process at new clock ticks.

resume ()
Resume calling the process at new clock ticks.

8.2 Simulation

The simulation object is both a global environment and a serial process scheduler. As an environment, it is passed as argument when calling the `on_tick()` functions of child processes, and also contains a number of fields, such as `dt` (simulation time step) or `gravity` (gravity vector in the world frame).

class pymanoid.sim.**Simulation** (*dt*, *env_path=None*, *env_xml=None*)
Simulation objects are the entry point of pymanoid scripts (similar to OpenRAVE environments, which they wrap).

Parameters

- **dt** (*real*) – Time interval between two ticks in simulation time.
- **env_path** (*string, optional*) – Load environment from XML/DAE file.
- **env_xml** (*string, optional*) – Load environment from XML string.

dt

Simulation timestep, that is, time interval between two ticks in simulation time. The default value of 30 [ms] is chosen so that stock simulations run real-time on modern computers. Lower simulation steps make forward integration more precise, but will result in loss of real-timeness at some point (a simulation runs real-time when the computer spends less than `dt` to carry out all computations at every tick, so that it takes N seconds of real time to simulate N seconds of simulation time).

Type scalar, optional

property camera_transform

Camera transform in the world frame.

load_mesh(*path*)

Load a pymanoid.Body from a DAE or VRML model.

Parameters **path** (*string*) – Path to DAE or VRML model.

log_comp_time(*label*, *ctime*)

Log computation time for a given process.

Parameters

- **label** (*string*) – Label of the operation.
- **ctime** (*scalar*) – Computation time in [s].

move_camera_to(*T*, *duration=0.0*, *dt=0.03*)

Continuously move the camera frame to a target transform.

Parameters

- **T** (*(4, 4) array*) – Target transform.
- **duration** (*scalar, optional*) – Motion duration.
- **dt** (*scalar, optional*) – Time step between intermeditate camera transforms.

record(*fname=None*)

Record simulation video.

Parameters **fname** (*string, optional*) – Video file name.

run_thread()

Run simulation thread.

schedule(*process*, *paused=False*, *log_comp_times=False*)

Add a process to the schedule list.

Parameters

- **process** (*pymanoid.Process*) – Process that will be called at every simulation tick.
- **paused** (*bool, optional*) – If True, the process will be initially paused.

Note: The order in which processes are scheduled does matter.

schedule_extra(*process*)
Schedule a Process not counted in the computation time budget.

set_background(*color=None*)
Set viewer background color.

Parameters **color** (*string, RGB tuple or None*) – Background color.

set_camera_back(*x=None*, *y=None*, *z=None*)
Align camera axis with the x-axis of the world frame.

Parameters

- **x** (*scalar or None*) – Camera translation along x-axis of the world frame.
- **y** (*scalar or None*) – Camera translation along y-axis of the world frame.
- **z** (*scalar or None*) – Camera translation along z-axis of the world frame.

set_camera_bottom(*x=None*, *y=None*, *z=None*)
Align camera axis with the z-axis of the world frame.

Parameters

- **x** (*scalar or None*) – Camera translation along x-axis of the world frame.
- **y** (*scalar or None*) – Camera translation along y-axis of the world frame.
- **z** (*scalar or None*) – Camera translation along z-axis of the world frame.

set_camera_front(*x=None*, *y=None*, *z=None*)
Align camera axis opposite to the x-axis of the world frame.

Parameters

- **x** (*scalar or None*) – Camera translation along x-axis of the world frame.
- **y** (*scalar or None*) – Camera translation along y-axis of the world frame.
- **z** (*scalar or None*) – Camera translation along z-axis of the world frame.

set_camera_left(*x=None*, *y=None*, *z=None*)
Align camera axis opposite to the y-axis of the world frame.

Parameters

- **x** (*scalar or None*) – Camera translation along x-axis of the world frame.
- **y** (*scalar or None*) – Camera translation along y-axis of the world frame.
- **z** (*scalar or None*) – Camera translation along z-axis of the world frame.

set_camera_right(*x=None*, *y=None*, *z=None*)

Align camera axis with the y-axis of the world frame.

Parameters

- **x** (*scalar or None*) – Camera translation along x-axis of the world frame.
- **y** (*scalar or None*) – Camera translation along y-axis of the world frame.
- **z** (*scalar or None*) – Camera translation along z-axis of the world frame.

set_camera_top(*x=None*, *y=None*, *z=None*)

Align camera axis opposite to the z-axis of the world frame.

Parameters

- **x** (*scalar or None*) – Camera translation along x-axis of the world frame.
- **y** (*scalar or None*) – Camera translation along y-axis of the world frame.
- **z** (*scalar or None*) – Camera translation along z-axis of the world frame.

set_camera_transform(*T*)

Set camera transform.

Parameters **T** (*(4, 4) array*) – Target transform.

Notes

See also *pymanoid.sim.Simulation.move_camera_to()*.

set_viewer(*plugin='qtcoin'*)

Open OpenRAVE viewer.

Parameters **plugin** (*string, optional*) – Viewer plugin name ('qtcoin' or 'qtosg'), defaults to 'qtcoin'.

start()

Start simulation thread.

step (*n=1*)
Perform a given number of simulation steps (default is one).

Parameters **n** (*int, optional*) – Number of simulation steps.

unschedule (*process*)
Remove a process to the schedule list.

Parameters **process** (*pymanoid.Process*) – Process that will be called at every simulation tick.

8.3 Camera recording

To record a video of your simulation, schedule a camera recorder as follows:

```
sim = pymanoid.Simulation(dt=0.03)
camera_recorder = pymanoid.CameraRecorder(sim, fname="my_video.mp4")
sim.schedule_extra(camera_recorder)
```

Upon scheduling the camera recorder, the following message will appear in your Python shell:

```
Please click on the OpenRAVE window.
```

The mouse pointer changes to a cross. While it is like this, click on the OpenRAVE window (so that the recorder knows which window to record from). Then, start or step your simulation as usual.

When your simulation is over, run the video conversion script created in the current directory:

```
./make_pymanoid_video.sh
```

After completion, the file `my_video.mp4` is created in the current directory.

class pymanoid.proc.**CameraRecorder**(*sim*, *fname=None*, *tmp_folder='pymanoid_rec'*)
Video recording process.

When created, this process will ask the user to click on the OpenRAVE GUI to get its window ID. Then, it will save a screenshot in the camera folder at each tick of the simulation (don't expect real-time recording...). When your simulation is over, go to the camera folder and run the script called `make_video.sh`.

Parameters

- **sim** (Simulation) – Simulation instance.
- **fname** (*string, optional*) – Video file name.
- **tmp_folder** (*string, optional*) – Temporary folder where screenshots will be recorded.

Note: Creating videos requires the following dependencies (here listed for Ubuntu 14.04): `sudo apt-get install x11-utils imagemagick`

`libav-tools`.

Note: Don't expect the simulation to run real-time while recording.

Note: The GUI window should stay visible on your screen for the whole duration of the recording. Also, don't resize it, otherwise video conversion will fail later on.

---

on_tick(*sim*)
Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

wait_for(*wait_time*)
Pause the video by repeating the last frame for a certain duration.

Parameters **wait_time** (*scalar*) – Duration in [s].

8.4 Making a new process

Imagine we want to record knee angles while the robot moves in the horizontal walking example. We can create a new process class:

```
class KneeAngleRecorder(pymanoid.Process):

    def __init__(self):
        """
        Initialize process. Don't forget to call parent constructor.
        """
        super(KneeAngleRecorder, self).__init__()
        self.left_knee_angles = []
        self.right_knee_angles = []
        self.times = []

    def on_tick(self, sim):
        """
        Update function run at every simulation tick.

        Parameters
        ----------
        sim : Simulation
            Instance of the current simulation.
        """
        self.left_knee_angles.append(robot.q[robot.left_knee])
        self.right_knee_angles.append(robot.q[robot.right_knee])
        self.times.append(sim.time)
```

To execute it, we instantiate a process of this class and add it to the simulation right before the call to `start_walking()`:

```
recorder = KneeAngleRecorder()
sim.schedule_extra(recorder)
start_walking()  # comment this out to start walking manually
```

We can plot the data recorded in this process at any time by:

```
import pylab
pylab.ion()
pylab.plot(recorder.times, recorder.left_knee_angles)
pylab.plot(recorder.times, recorder.right_knee_angles)
```

Note that we scheduled the recorder as an extra process, using `sim.schedule_extra` rather than `sim.schedule`, so that it does not get counted in the computation time budget of the control loop.

9 Graphical user interface

9.1 Primitive functions

`pymanoid.gui.`**`draw_arrow`**(*origin*, *end*, *color='r'*, *linewidth=0.02*)

Draw an arrow between two points.

Parameters

- **`origin`** (*`array, shape=(3,)`*) – World coordinates of the origin of the arrow.
- **`end`** (*`array, shape=(3,)`*) – World coordinates of the end of the arrow.
- **`color`** (*`char or triplet, optional`*) – Color letter or RGB values, default is 'g' for green.
- **`linewidth`** (*`scalar, optional`*) – Thickness of arrow.

Returns **handle** – OpenRAVE graphical handle. Must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type openravepy.GraphHandle

`pymanoid.gui.`**`draw_force`**(*point*, *force*, *scale=0.0025*, *color='r'*, *linewidth=0.015*)

Draw a force acting at a given point.

Parameters

- **`point`** (*`array, shape=(3,)`*) – Point where the force is acting.
- **`force`** (*`array, shape=(3,)`*) – Force vector, in [N].

- **scale** (*scalar, optional*) – Force-to-distance scaling factor in [N] / [m].
- **linewidth** (*scalar, optional*) – Thickness of force vector.

Returns handles – Must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type list of GUI handles

pymanoid.gui.**draw_line**(*start_point*, *end_point*, *color='g'*, *linewidth=1.0*)

Draw a line between two points.

Parameters

- **start_point** (*array, shape=(3,)*) – One end of the line, in world frame coordinates.
- **end_point** (*array, shape=(3,)*) – Other end of the line, in world frame coordinates.
- **color** (*char or triplet, optional*) – Color letter or RGB values, default is 'g' for green.
- **linewidth** (*scalar*) – Thickness of drawn line.

Returns handle – OpenRAVE graphical handle. Must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type openravepy.GraphHandle

pymanoid.gui.**draw_point**(*point*, *color='g'*, *pointsize=0.01*)

Draw a point.

Parameters

- **point** (*array, shape=(3,)*) – Point coordinates in the world frame.
- **pointsize** (*scalar, optional*) – Radius of the drawn sphere in [m].

Returns handle – OpenRAVE graphical handle. Must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type openravepy.GraphHandle

pymanoid.gui.**draw_points**(*points*, *color='g'*, *pointsize=0.01*)

Draw a list of points.

Parameters

- **point** (*list of arrays*) – List of point coordinates in the world frame.
- **pointsize** (*scalar, optional*) – Radius of the drawn sphere in [m].

Returns handle – OpenRAVE graphical handle. Must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type openravepy.GraphHandle

pymanoid.gui.**draw_trajectory**(*points*, *color='b'*, *linewidth=3*, *pointsize=0.01*)

Draw a trajectory as a set of points connected by line segments.

Parameters

- **= array** (*points*) – List of points or 2D array.
- **shape=(N** – List of points or 2D array.
- **3)** – List of points or 2D array.
- **color** (*char or triplet, optional*) – Color letter or RGB values, default is 'g' for green.
- **linewidth** (*scalar*) – Thickness of drawn line.
- **pointsize** (*scalar*) – Vertex size.

Returns handles – OpenRAVE graphical handles. Must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type list of openravepy.GraphHandle

pymanoid.gui.**draw_wrench**(*surface*, *wrench*, *scale=0.005*, *pointsize=0.02*, *linewidth=0.01*, *yaw_moment=False*)

Draw a 6D wrench as a 3D force applied at the center of pressure of a given surface frame.

Parameters

- **surface** (*surface*) – Surface at which the wrench is acting.
- **wrench** (*ndarray*) – 6D wrench vector in world-frame coordinates.
- **scale** (*scalar*) – Scaling factor between Euclidean and Force spaces.
- **pointsize** (*scalar*) – Point radius in [m].
- **linewidth** (*scalar*) – Thickness of force vector.
- **yaw_moment** (*bool, optional*) – Depict the yaw moment by a blue line.

Returns handles – This list must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type list of OpenRAVE handles

9.2 Convex polyhedra

Polyhedra correspond to the sets of linear inequality constraints applied to the system, for instance the support area in which the zero-tilting moment point (ZMP) of a legged robot should lie to avoid breaking contacts with the ground.

pymanoid.gui.**draw_2d_cone**(*vertices*, *rays*, *normal*, *combined='g-#'*, *color=None*, *faces=None*)

Draw a 2D cone defined from its rays and vertices. The normal vector n of the plane containing the cone must also be supplied.

Parameters

- **vertices** (*list of 3D arrays*) – Vertices of the 2D cone in the world frame.
- **rays** (*list of 3D arrays*) – Rays of the 2D cone in the world frame.
- **normal** (*array, shape=(3,)*) – Unit vector normal to the drawing plane.
- **combined** (*string*) – Drawing spec in matplotlib fashion: color letter, followed by characters representing the faces of the cone to draw ('.' for vertices, '-' for edges, '#' for facets). Default is 'g-#'.
- **color** (*char or triplet, optional*) – Color letter or RGB values, default is 'g' for green.
- **faces** (*string*) – Specifies the faces of the polyhedron to draw. Format is the same as `combined`.

Returns handle – OpenRAVE graphical handle. Must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type openravepy.GraphHandle

pymanoid.gui.**draw_cone**(*apex*, *axis*, *section*, *combined='r-#'*, *color=None*, *linewidth=2.0*, *pointsize=0.05*)

Draw a 3D cone defined from its apex, axis vector and a cross-section polygon (defined in the plane orthogonal to the axis vector).

Parameters

- **apex** (*array*) – Position of the origin of the cone in world coordinates.
- **axis** (*array*) – Unit vector directing the cone axis and lying inside.
- **combined** (*string, optional*) – Drawing spec in matplotlib fashion. Default is 'g-#'.
- **linewidth** (*scalar, optional*) – Thickness of the edges of the cone.
- **pointsize** (*scalar, optional*) – Point size in [m].

Returns handles – Must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type list of GUI handles

pymanoid.gui.**draw_horizontal_polygon**(*points*, *height*, *combined='g-#'*, *color=None*, *faces=None*, *linewidth=1.0*, *pointsize=0.01*)

Draw a horizontal polygon defined as the convex hull of a set of 2D points.

Parameters

- **points** (*list of arrays*) – List of coplanar 2D points.
- **height** (*scalar*) – Height to draw the polygon at.
- **combined** (*string, optional*) – Drawing spec in matplotlib fashion. Default: 'g-#'.
- **color** (*char or triplet, optional*) – Color letter or RGB values, default is 'g' for green.
- **faces** (*string*) – Faces of the polyhedron to draw. Use '.' for vertices, '-' for edges and '#' for facets.
- **linewidth** (*scalar*) – Thickness of drawn line.
- **pointsize** (*scalar*) – Vertex size.

Returns handles – OpenRAVE graphical handles. Must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type list of openravepy.GraphHandle

pymanoid.gui.**draw_polygon**(*points*, *normal*, *combined='g-#'*, *color=None*, *faces=None*, *linewidth=1.0*, *pointsize=0.01*)

Draw a polygon defined as the convex hull of a set of points. The normal vector n of the plane containing the polygon must also be supplied.

Parameters

- **points** (*list of arrays*) – List of coplanar 3D points.
- **normal** (*array, shape=(3,)*) – Unit vector normal to the drawing plane.
- **combined** (*string, optional*) – Drawing spec in matplotlib fashion. Default: 'g-#'.
- **color** (*char or triplet, optional*) – Color letter or RGB values, default is 'g' for green.
- **faces** (*string*) – Faces of the polyhedron to draw. Use '.' for vertices, '-' for edges and '#' for facets.
- **linewidth** (*scalar*) – Thickness of drawn line.
- **pointsize** (*scalar*) – Vertex size.

Returns handles – OpenRAVE graphical handles. Must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type list of openravepy.GraphHandle

pymanoid.gui.**draw_polytope**(*points*, *combined='g-#'*, *color=None*, *faces=None*, *linewidth=1.0*, *pointsize=0.01*, *hull=None*)

Draw a polyhedron defined as the convex hull of a set of points.

Parameters

- **points** (*list of arrays*) – List of 3D points in the world frame.
- **combined** (*string, optional*) – Drawing spec in matplotlib fashion. Default: 'g-#'.
- **color** (*char or triplet, optional*) – Color letter or RGB values, default is 'g' for green.
- **faces** (*string, optional*) – Faces of the polytope to draw. Use '.' for vertices, '-' for edges and '#' for facets.
- **hull** (*scipy.spatial.ConvexHull*) – 2D convex hull provided when drawing polygons, in which case the 3D hull has zero volume.
- **linewidth** (*scalar*) – Thickness of drawn line.
- **pointsize** (*scalar*) – Vertex size.

Returns handles – OpenRAVE graphical handles. Must be stored in some variable, otherwise the drawn object will vanish instantly.

Return type list of openravepy.GraphHandle

Notes

In the `faces` or `combined` strings, use '.' for vertices, '-' for edges and '#' for facets.

9.3 Drawers

class pymanoid.gui.**WrenchDrawer**

Draw contact wrenches applied to the robot.

on_tick(*sim*)

Find supporting contact forces at each COM acceleration update.

Parameters **sim** (*pymanoid.Simulation*) – Simulation instance.

class pymanoid.gui.**PointMassWrenchDrawer**(*point_mass*, *contact_set*)

Draw contact wrenches applied to a point-mass system in multi-contact.

Parameters

- **point_mass** (PointMass) – Point-mass to which forces are applied.
- **contact_set** (ContactSet) – Set of contacts providing interaction forces.

on_tick(*sim*)
Find supporting contact forces at each COM acceleration update.

Parameters **sim** (*pymanoid.Simulation*) – Simulation instance.

class pymanoid.gui.**RobotWrenchDrawer**(*robot*)

on_tick(*sim*)
Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

class pymanoid.gui.**StaticEquilibriumWrenchDrawer**(*stance*)
Draw contact wrenches applied to a robot in static-equilibrium.

Parameters **stance** (*pymanoid.Stance*) – Contacts and COM position of the robot.

class pymanoid.gui.**TrajectoryDrawer**(*body*, *combined='b-'*, *color=None*, *linewidth=3*, *linestyle=None*, *buffer_size=1000*)
Draw the trajectory of a rigid body.

Parameters

- **body** (Body) – Rigid body whose trajectory to draw.
- **combined** (*string, optional*) – Drawing spec of the trajectory in matplotlib fashion.
- **color** (*char or RGBA tuple, optional*) – Drawing color.
- **linewidth** (*scalar, optional*) – Thickness of drawn lines.
- **linestyle** (*char, optional*) – Choix between '-' for continuous and '.' for dotted.
- **buffer_size** (*int, optional*) – Number of trajectory segments to display. Old segments will be replaced by new ones.

on_tick(*sim*)
Main function called by the simulation at each control cycle.

Parameters **sim** (Simulation) – Current simulation instance.

10 References

References

[Caron20] Biped Stabilization by Linear Feedback of the Variable-Height Inverted Pendulum Model, S. Caron, IEEE International Conference on Robotics and Automation, May 2020.

[Caron19] Capturability-based Pattern Generation for Walking with Variable Height, S. Caron, A. Escande, L. Lanari and B. Mallein, IEEE Transactions on Robotics, July 2019.

[Audren18] 3D robust stability polyhedron in multi-contact, H. Audren and A. Kheddar, IEEE Transactions on Robotics, vol., 34, no. 2, February 2018.

[Caron17w] Dynamic Walking over Rough Terrains by Nonlinear Predictive Control of the Floating-base Inverted Pendulum, S. Caron and A. Kheddar, IEEE/RSJ International Conference on Intelligent Robots and Systems, September 2017.

[Caron17z] ZMP support areas for multi-contact mobility under frictional constraints, S. Caron, Q.-C. Pham and Y. Nakamura, IEEE Transactions on Robotics, vol. 33, no. 1, February 2017.

[Scianca16] Intrinsically stable MPC for humanoid gait generation, N. Scianca, M. Cognetti, D. De Simone, L. Lanari and G. Oriolo, IEEE-RAS International Conference on Humanoid Robots, November 2016.

[Caron16] Multi-contact Walking Pattern Generation based on Model Preview Control of 3D COM Accelerations, S. Caron and A. Kheddar, IEEE-RAS International Conference on Humanoid Robots, November 2016.

[Nozawa16] Three-dimensional humanoid motion planning using COM feasible region and its application to ladder climbing tasks, Nozawa, Shunichi, et al., IEEE-RAS International Conference on Humanoid Robots, November 2016.

[Caron15] Stability of surface contacts for humanoid robots: Closed-form formulae of the contact wrench cone for rectangular support areas, S. Caron, Q.-C. Pham and Y. Nakamura, IEEE International Conference on Robotics and Automation, May 2015.

[Englsberger15] Three-Dimensional Bipedal Walking Control Based on Divergent Component of Motion, J. Englsberger, C. Ott and A. Albu-Schäffer, IEEE Transactions on Robotics, vol. 31, no. 2, March 2015.

[Flacco15] Control of Redundant Robots Under Hard Joint Constraints: Saturation in the Null Space, F. Flacco, A. De Luca and O. Khatib, IEEE Transactions on Robotics, vol. 31, no. 3, June 2015.

[Audren14] Model preview control in multi-contact motion-application to a humanoid robot, H. Audren et al., IEEE/RSJ International Conference on Intelligent Robots and Systems, September 2014.

[ElKhoury13] Planning Optimal Motions for Anthropomorphic Systems, A. El Khoury, June 2013.

[Kanoun12] Real-time prioritized kinematic control under inequality constraints for redundant manipulators, O. Kanoun, Robotics: Science and Systems, June 2011.

[Sugihara11] Solvability-Unconcerned Inverse Kinematics by the Levenberg-Marquardt Method, T. Sugihara, IEEE Transactions on Robotics, vol. 27, no. 5, October 2011.

[Bretl08] Testing Static Equilibrium for Legged Robots, T. Bretl and S. Lall, IEEE Transactions on Robotics, vol. 24, no. 4, August 2008.

[Wieber06] Trajectory free linear model predictive control for stable walking in the presence of strong perturbations, P.-B. Wieber, IEEE-RAS International Conference on Humanoid Robots, 2006.

[Diebel06] Representing Attitude: Euler Angles, Unit Quaternions, and Rotation Vectors, J. Diebel, 2006.

[Sardain04] Forces acting on a biped robot. Center of pressure-Zero moment point, P. Sardain and G. Bessonnet, IEEE Transactions on Systems, Man and Cybernetics, vol. 34, no. 5, September 2004.

[Kajita01] The 3D Linear Inverted Pendulum Mode: A simple modeling for a biped walking pattern generation, S. Kajita, F. Kanehiro, K. Kaneko, K. Yokoi and H. Hirukawa, IEEE/RSJ International Conference on Intelligent Robots and Systems, 2001.

[Walker82] Efficient dynamic computer simulation of robotic mechanisms, M. Walker and D. Orin, ASME Trans. J. Dynamics Systems, Measurement and Control, vol. 104, 1982.

Python Module Index

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z