capture_walking Manual

Release 0.8

Stéphane Caron

Feb 06, 2018

Contents

An implementation for pymanoid of the walking controller described in *[Caron18]*.

1 Capturability of the inverted pendulum

This framework applies to the *inverted pendulum model* (IPM), a reduced model for 3D walking whose equation of motion is:

$$\ddot{\boldsymbol{c}} = \lambda(\boldsymbol{c} - \boldsymbol{r}) + \boldsymbol{g}$$

with $\boldsymbol{c}$ the position of the center of mass (CoM) and $\boldsymbol{g} = -g\boldsymbol{e}_z$ the gravity vector. The two control inputs of the IPM are the location of its center of pressure (CoP) $\boldsymbol{r}$ and its stiffness λ. Parameters of the IPM are:

- g: the gravitational constant

- λ_{min} and λ_{max}: lower and upper bound on the stiffness λ

This model is implemented in the `pymanoid.InvertedPendulum` class:

```
from pymanoid.sim import gravity_const as g
pendulum = InvertedPendulum(
    com, comd, contact=support_contact,
    lambda_min=0.1 * g, lambda_max=2 * g)
sim.schedule(pendulum)  # integrate IPM dynamics
```

To make the robot's inverse kinematics track the reduce model, call:

```
robot.setup_ik_for_walking(pendulum.com)
sim.schedule(robot.ik)  # enable robot IK
```

where `robot` is a `pymanoid.Humanoid` robot model. IPM states (CoM position `com` and velocity `comd`) will then be sent to the `pymanoid.IKSolver` inverse kinematics of the robot.

1.1 Capture problem

The gist of capturability analysis is to solve *capture problems* that quantify the ability to bring the robot to a stop at a desired 3D target. Mathematically, a capture problem is formalized as:

$$
\begin{aligned}
\text{minimize}_{\boldsymbol{\varphi} \in \mathbb{R}^n} \quad & \sum_{j=1}^{n-1} \left[\frac{\varphi_{j+1} - \varphi_j}{\delta_j} - \frac{\varphi_j - \varphi_{j-1}}{\delta_{j-1}} \right]^2 \\
\text{subject to} \quad & \sum_{j=0}^{n-1} \frac{\delta_j}{\sqrt{\varphi_{j+1}} + \sqrt{\varphi_j}} - \frac{\bar{z}_{\text{i}} \sqrt{\varphi_n} + \dot{\bar{z}}_{\text{i}}}{g} = 0 \\
& \omega_{\text{i,min}}^2 \leq \varphi_n \leq \omega_{\text{i,max}}^2 \\
& \forall j < n, \ \lambda_{\text{min}} \delta_j \leq \varphi_{j+1} - \varphi_j \leq \lambda_{\text{max}} \delta_j \\
& \varphi_1 = \delta_0 g / \bar{z}_{\text{f}}
\end{aligned}
$$

where the following notations are used:

- n is the number of discretization steps
- $\delta_1, \ldots, \delta_n$ are spatial discretization steps

As these quantities don't vary between capture problems during walking, they are set in the constructor of the *capture_walking.CaptureProblem* class:

class `capture_walking.`**`CaptureProblem`**(*lambda_min*, *lambda_max*, *nb_steps*)

Capture optimization problem.

Parameters

- **`lambda_min`** (*scalar*) – Minimum leg stiffness (positive).
- **`lambda_max`** (*scalar*) – Maximum leg stiffness (positive).

- **nb_steps** (*integer*) – Number of segments where $\lambda(t)$ is constant.

The remaining notations in the capture problem above are:

- $\bar{z}_\mathrm{i}$ is the instantaneous CoM height
- $\bar{z}_\mathrm{f}$ is the desired CoM height at the end of the capture trajectory
- ω_min and ω_max are the lower and upper bound on IPM damping (representing notably the limits of the CoP area)

These quantities are state-dependent, and can be set via the following setters:

`CaptureProblem.`**`set_init_omega_lim`**(*init_omega_min*, *init_omega_max*)
Set minimum and maximum values for the initial IPM damping omega.

Parameters

- **init_omega_min** (*scalar*) – Lower bound.
- **init_omega_max** (*scalar*) – Upper bound.

`CaptureProblem.`**`set_init_zbar`**(*init_zbar*)
Set the initial CoM height.

Parameters **init_zbar** (*scalar*) – Initial CoM height.

`CaptureProblem.`**`set_init_zbar_deriv`**(*init_zbar_deriv*)
Set the initial CoM height.

Parameters **init_zbar_deriv** (*scalar*) – Initial derivative of the CoM height.

`CaptureProblem.`**`set_target_height`**(*target_height*)
Set the target CoM height.

Parameters **target_height** (*scalar*) – Target CoM height.

Once a capture problem is fully constructed, you can solve it by calling:

`CaptureProblem.`**`solve`**(*solver=None*)
Solve the capture problem.

Parameters **solver** (*string, optional*) – Solver to use, between `"cps"` (default) and `"ipopt"`. You can also use the internal attribute `self.nlp_solver` to save this setting.

Returns **solution** – Solution to the problem, if any.

Return type *CaptureSolution*

By default, *capture_walking.CaptureProblem* is a thin wrapper used to call CPS, a tailored SQP optimization for this precise problem. You can also call the generic solver IPOPT with the function above (requires CasADi).

1.2 Capture solution

Solutions found by the solver are stored in a:

class capture_walking.**CaptureSolution**(*phi_1_n*, *capture_pb*, *optimal_found=None*)

Solution to a capture optimization problem.

Parameters

- **phi_1_n** (*array*) – Vector of optimization variables returned by call to solver.
- **capture_pb** (CaptureProblem) – Original capture problem.
- **optimal_found** (*bool*) – Did the solver converge to this solution?

Capture solutions are lazily computed: by default, only the instantaneous IPM inputs λ_i, $\boldsymbol{r}_\mathrm{i}$ and ω_i are computed. The complete solution (all values of $\lambda(t)$ as well as its switch times t_j) is completed by calling:

CaptureSolution.**compute_lambda**()

Compute the full vector of stiffness values.

CaptureSolution.**compute_switch_times**()

Compute the times t_j where $s(t_j) = s_j$.

From there, all spatial mappings $\lambda(s), \omega(s), t(s)$ and time mappings $\lambda(t), \omega(t), s(t)$ can be accessed via:

CaptureSolution.**lambda_from_s**(*s*)

Compute the leg stiffness $\lambda(s)$ for a given path index.

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

Returns **lambda_** – Leg stiffness $\lambda(s)$.

Return type scalar

CaptureSolution.**lambda_from_t**(*t*)

Compute the leg stiffness $\lambda(t)$ to apply at time *t*.

Parameters **t** (*scalar*) – Time in [s]. Must be positive.

Returns **lambda_** – Leg stiffness $\lambda(t)$.

Return type scalar

CaptureSolution.**omega_from_s**(*s*)

Compute $\omega(s)$ for a given path index.

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

Returns **omega** – Value of $\omega(s)$.

Return type scalar

`CaptureSolution.`**`omega_from_t`**`(`*t*`)`

Compute the value of $\omega(t)$.

Parameters **t** (*scalar*) – Time in [s]. Must be positive.

Returns **omega** – Value of $\omega(t)$.

Return type scalar

`CaptureSolution.`**`s_from_t`**`(`*t*`)`

Compute the path index corresponding to a given time.

Parameters **t** (*scalar*) – Time in [s]. Must be positive.

Returns **s** – Path index *s(t)*.

Return type scalar

`CaptureSolution.`**`t_from_s`**`(`*s*`)`

Compute the time corresponding to a given path index.

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

Returns **t** – Time *t(s) > 0* in [s].

Return type scalar

Notes

Given the index *j* such that $s_j \leq s < s_{j+1}$, the important formula behind this function is:

$$t(s) = t_{j+1} + \frac{1}{\sqrt{\lambda_j}} \log \left(\frac{\sqrt{\varphi_{i+1}} + \sqrt{\lambda_j} s_{j+1}}{\sqrt{\varphi_{i+1} - \lambda_j (s_{j+1}^2 - s^2)} + \sqrt{\lambda_j} s} \right)$$

See the paper for a derivation of this formula.

2 Walking controller

The ability to solve capture problems is turned into a full-fledged walking controller by the `WalkingController` class:

class `capture_walking.`**`WalkingController`**`(`*robot*, *pendulum*, *contact_feed*, *nb_steps*, *target_height*`)`

Main walking controller.

Parameters

- **robot** (*pymanoid.Robot*) – Robot model.
- **pendulum** (*pymanoid.InvertedPendulum*) – Inverted pendulum model.

- **contact_feed** (*pymanoid.ContactFeed*) – Footstep sequence of the walking scenario.
- **nb_steps** (*int*) – Number of spatial discretization steps in capture problems.
- **target_height** (*scalar*) – CoM height above target contacts in asymptotic static equilibrium.

This class is a pymanoid process that you can readily schedule to your simulation:

```
controller = WalkingController(
    robot, pendulum, contact_feed, nb_steps=10, target_com_height=0.
 ↪8)
sim.schedule(controller)
```

The controller follows a Finite State Machine (FSM) with two states: *Zero-step capture*, where the robot balances on its support leg while swinging for the next footstep, and *One-step capture*, where the robot pushes on its support leg toward the next footstep. See *[Caron18]* for details. When walking is finished, a simple *Double-support capture* strategy is applied to bring the center of mass (CoM) to a mid-foot location.

2.1 Zero-step capture

Zero-step capturability is handled by the *ZeroStepController* class:

class capture_walking.**ZeroStepController**(*pendulum*, *nb_steps*, *target_height*, *cop_gain*)

Balance controller based on predictive control with boundedness condition.

Parameters

- **pendulum** (*pymanoid.InvertedPendulum*) – State estimator of the inverted pendulum.
- **nb_steps** (*integer*) – Number of discretization steps for the preview trajectory.
- **target_height** (*scalar*) – Desired altitude in the stationary regime.
- **cop_gain** (*scalar*) – CoP feedback gain (must be > 1).

Notes

This implementation works in a local frame, as documented in the ICRA 2018 report. Computations are a bit simpler in the world frame as done in *OneStepController*.

The target contact is set independently by calling:

ZeroStepController.**set_contact**(*contact*)

Update the supporting contact.

Parameters **contact** (*pymanoid.Contact*) – New contact to use for stabilization.

The pendulum reference to the inverted pendulum model is used to update the CoM state (position and velocity) when computing control inputs:

ZeroStepController.**compute_controls**()
Compute pendulum controls for the current state.

Returns

- **cop** (*(3,) array*) – CoP coordinates in the world frame.
- **push** (*scalar*) – IPM stiffness $\lambda \geq 0$.

These two inputs can then be sent to the IPM for zero-step capture.

2.2 One-step capture

One-step capturability is handled by the *OneStepController* class:

class capture_walking.**OneStepController**(*pendulum*, *nb_steps*, *target_height*)
Stepping controller based on predictive control with boundedness condition.

Parameters

- **pendulum** (*pymanoid.InvertedPendulum*) – State estimator of the inverted pendulum.
- **nb_steps** (*integer*) – Number of discretization steps for the preview trajectory.
- **target_height** (*scalar*) – Desired altitude at the end of the step.

The support and target contacts are set independently by calling:

OneStepController.**set_contacts**(*support_contact*, *target_contact*)
Update support and target contacts.

Parameters

- **support_contact** (*pymanoid.Contact*) – Contact used during the takeoff phase.
- **target_contact** (*pymanoid.Contact*) – Contact used during the landing phase.

The pendulum reference to the inverted pendulum model is used to update the CoM state (position and velocity) when computing control inputs:

OneStepController.**compute_controls**(*time_to_heel_strike=None*)
Compute pendulum controls for the current state.

Parameters **time_to_heel_strike** (*scalar*) – When set, make sure that the contact switch happens after this time.

Returns

- **cop** (*(3,) array*) – CoP coordinates in the world frame.
- **push** (*scalar*) – Leg stiffness $\lambda \geq 0$.

These two inputs can then be sent to the IPM for one-step capture.

2.3 Double-support capture

At the end of an acyclic contact sequence, a simple double-support strategy is applied by the *DoubleSupportController* class:

class `capture_walking.`**`DoubleSupportController`**(*pendulum*, *stance*, *target_height*, *k=1.0*)

Simple controller used to stop in double support after walking. Implements the control law used to prove small-space controllability of the IPM in an Appendix of the paper.

Parameters

- **`pendulum`** (*`pymanoid.InvertedPendulum`*) – State estimator of the inverted pendulum.
- **`stance`** (*`pymanoid.Stance`*) – Double-support stance.
- **`target_height`** (*`scalar`*) – Desired altitude at the end of the step.
- **`k`** (*`scalar`*) – Stiffness scaling parameter.

Notes

The output CoM acceleration behavior will be that of a spring-damper with critical damping and a variable stiffness of *k * lambda(c)*. See the paper for details.

Like the two preceding classes, the `pendulum` reference to the inverted pendulum model is used to update the CoM state (position and velocity) when computing control inputs:

`OneStepController.`**`compute_controls`**(*time_to_heel_strike=None*)

Compute pendulum controls for the current state.

Parameters **`time_to_heel_strike`** (*`scalar`*) – When set, make sure that the contact switch happens after this time.

Returns

- **cop** (*(3,) array*) – CoP coordinates in the world frame.
- **push** (*scalar*) – Leg stiffness $\lambda \geq 0$.

These two inputs can then be sent to the robot's CoM task directly.

3 References

References

[Caron18] Capturability-based Analysis, Optimization and Control of 3D Bipedal Walking, S. Caron, A. Escande, L. Lanari and B. Mallein, submitted, January 2018.

Index