pyCub

Lukáš Rustler

Get Started
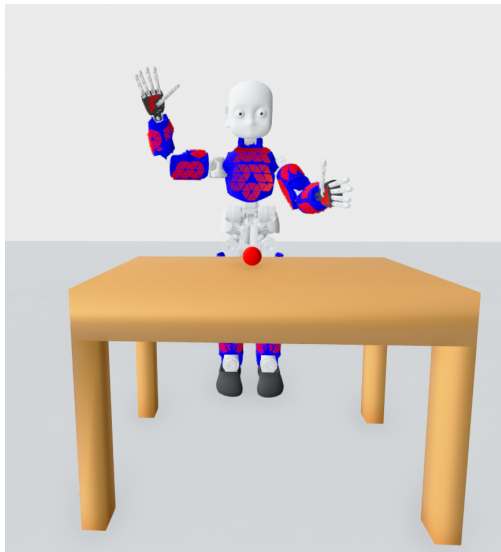
Introduction

- iCub Humanoid Robot
- pure Python3
  - 3.8 - 3.11
- physics from PyBullet[a]
- visualization in Open3D[b]

---

[a]https://pybullet.org
[b]https://www.open3d.org

Installation

Python only

- install Python3.10-3.11
- install with `python3 -m pip install icub_pybullet`
  - or, clone `https://github.com/rustlluk/pyCub`, go to `icub_pybullet` directory and install the with `python3 -m pip install .`

Docker

- Native (GNU/Linux only) Native Version
- VNC (All systems) VNC Version

GitPod

- Open `https://gitpod.io/#github.com/rustlluk/pycub`

For more information see GitHub README or documentation at
`https://lukasrustler.cz/pycub_documentation`.

Components

Config
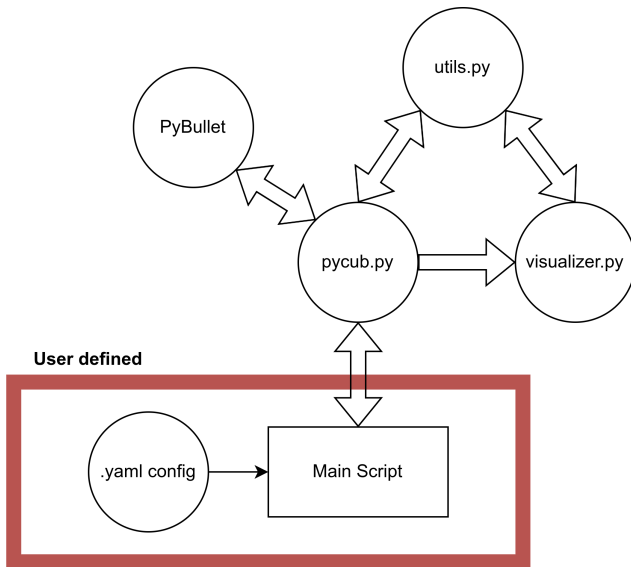
Important options:

- `gui`: GUI options
    - `standard`: default OpenGL visualization; True/False; default: True
    - `web`: web visualization visualization; True/False; default: False
- `end_effector`: which link is used as EE in Cartesian control; string; default: "r_hand"
- `initial_joint_angles`: dictionary with initial angles (in degrees) for joints. Can be empty.
- `log`: logging settings
    - `log`: whether to log; True/False; default: True
    - `period`: period of logging; float; default: 0.01; 0 for logging at each step
- `simulation_step`: the simulation advances for 1/simulation_step; float; default: 240; low value can break the simulation
- `self_collisions`: whether to detect self-collisions of robot links; True/False; default: True

Config - Objects

- way how to load other objects then the robot
- can load *.urdf* or *.obj* files
  - URDF from *.obj* file is created automatically with: `mass = 0.2 kg`; `lateral_friction = 1`; `rolling_friction = 0`

Structure:

- `urdfs`:
  - paths: list of paths to the files; relative to *other_meshes* directory
  - positions: list of 1x3 lists of positions of the files in world frame
  - fixed: list of bools; True when the object is not movable, i.e., it is not influenced by gravity
  - color: list of 1x3 lists of 0-1 float to specify RGB color; can be an empty list when URDF is used
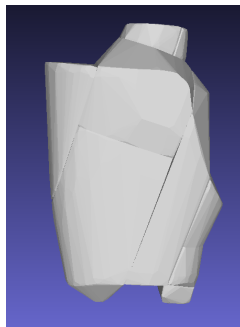
Example:

```
urdfs:
  paths: [plane/plane.urdf, ball/ball.obj, table/table.obj]
  positions: [[0, 0, 0]], [−0.35, 0, −0.1], [−0.6, −0.4, −0.225]]
  fixed: [True, False, True]
  color: [[], [1, 0, 0], [0.825, 0.41, 0.12]]
```

Collision Meshes

To simplify collision detection, PyBullet uses convex hulls of collision geometries. That means that collisions for concave meshes are not precise. From that reason, V-HACD is used to decompose collision geometries to convex parts. Everything is done automatically inside `pyCub`[1].

(a) Original forearm mesh.

(b) Decomposed forearm mesh.

---

[1]First run with new meshes takes more time as the meshes are decomposed.

User Scripts

The most simple example that loads the world and waits until the GUI is closed is shown below.

```python
from icub_pybullet.pycub import pyCub

if __name__ == "__main__":
    # load the robot with correct world/config
    client = pyCub(config="with_ball.yaml")

    # wait until the gui is closed
    while client.is_alive():
        client.update_simulation()
```

Simulation Control

- The simulation is not updating by itself, i.e., users have to call `pyCub.update_simulation()` to do one step.
- By default, a simulation step with GUI is performed only if the last step was done more than 0.01 second ago, no matter how often you call `pyCub.update_simulation()`. Without GUI the simulation is running as fast as possible.
  - This is usefull mainly to make visualization slower
  - you can control it with parameter in `pyCub.update_simulation()`. For example, to make the visualization run as fast as possible use `pyCub.update_simulation(None)`
- Some function (e.g., moving) can update the simulation automatically

Joints and Links

- There are two list for joints and links, `pyCub.joints` and `pyCub.links`. The lists include instances of Joint and Link classes.
- The lists include only joints that are not fixed and links that contain collision geometry.

Important Joint variables:

- `name`: string name of the joint; can be used for Joint Space Movement
- `robot_joint_id`: index of the joint in URDF; used by PyBullet
- `joints_id`: index of the joint in `pyCub.joints` list; can be used for Joint Space Movement

The reason to have two sets of indexes is that iCub URDF contains a lot of fixed joints, and it is easier for users to care only about the moveable ones.

To find a joint index by joint name or vice versa, there is a function `pyCub.find_joint_id()`.

Joint Space Movement

Movement in joint space can be achieved with function
`pyCub.move_position(self, joints, positions, wait=True, velocity=1, set_col_state=True, check_collision=True, timeout=None)`.

- `joints` can be an integer (index of the joint), string (name of the joint) or list of integers or strings
- `positions` can be a float or list of floats with the same size as `joints`
- if `wait` is set to True then the command is blocking, i.e., the main script will wait until the motion is done (all joints are the desired position or collision occured)
    - if `wait` is False, then you can check for the end of the movement in the main script with `pyCub.wait_motion_done()` or `pyCub.motion_done()`
- `velocity` sets the maximum joint velocity. **The robot may still go slower if the trajectory does not allow for higher velocity.**
- if `check_collision` is True and `wait` is also True, then the robot stops even if a collision occurs.
- `timeout` can be float in seconds after which the movement is stopped

Cartesian Movement

Movement in Cartesian space can be achieved with
`pyCub.move_cartesian(self, pose, wait=True, velocity=1,`
`check_collision=True, timeout=None)`.

- `pose` as 6D end-effector pose of type `utils.Pose`; it contains two attributes `pos` and `ori` that lists of position (1x3) and orientation (1x4, x, y, z, w quaternion)
- other arguments are the same as for Joint Space movement

End-effector of the robot can changed by changing `pyCub.end_effector` of your `pyCub` instance with a different instance of `pyCub.EndEffector` class.

The movement itself is achieved by computing the inverse kinematics of the input pose and running the joint space movement.
There is no planner included. The resulting trajectories will be mostly random, and collisions are checked only during movement.

Joint Velocity Movement

Movement in velocity space can be achieved with function
`pyCub.move_velocity(self, joints, velocities)`.

- `joints` can be an integer (index of the joint), string (name of the joint) or list of integers or strings
- `velocities` can be a float or list of floats with the same size as `joints`

There are no wait or collision parameters for this kind of movements, as it usually considered low-level.

Waiting for Motion

There are three main ways to wait for motion completion.

1. setting `wait` parameter of `move_position()` or `move_cartesian()` to True
2. using `pyCub.wait_motion_done(sleep_duration=0.01, check_collision=True)`
   - this way you can change visualization speed
   - the function will return in moment when all joints are at the desired position, when collision occurs, or when the movement is longer than timeout
     - in case of collision, `pyCub.collision_during_motion` is set to True
3. use `pyCub.motion_done(joints=None, check_collision=True)`
   - this way you can do other things while waiting

```python
while not client.motion_done(): # while motion
    # DO SOMETHING
    client.update_simulation(0.1) # update simulation
```

Logging

- things in the terminal can be logged using `pyCub.logger`
  - it uses python `logging` library
  - there is 5 levels (debug, info, warning, error, critical); debug is not showing by default
  - e.g., `pyCub.logging.info("Information message")`
- you can also log 6D Pose of the end-effector. If you set `log_pose` in a config file to True, the poses of the robot will be stored at `pyCub.pose_logger`.
- if you set `log.log` in .yaml config to True, then the state of the robot is also saved to .csv file
  - it can be used to "replay" the simulation later
  - the structure is `timestamp;steps_done;joint_0;...;joint_n`
    - in case of skin, there is also comma-separated output of each enabled skin part

	timestamp ⇕	steps_done ⇕	r_hip_pitch ⇕	r_hip_roll ⇕
1	1704980437.011004	1	-2.347182728778396e-07	2.5472441931889862e-11
2	1704980437.0968451	2	-4.4835364622873956e-07	4.865775742303313e-11
3	1704980437.1096358	3	-6.406416272993617e-07	6.952707909501663e-11
4	1704980437.1224136	4	-8.136935392390321e-07	8.830938650966017e-11
5	1704980437.1350477	5	-9.694334936051998e-07	1.0521336509366838e-10