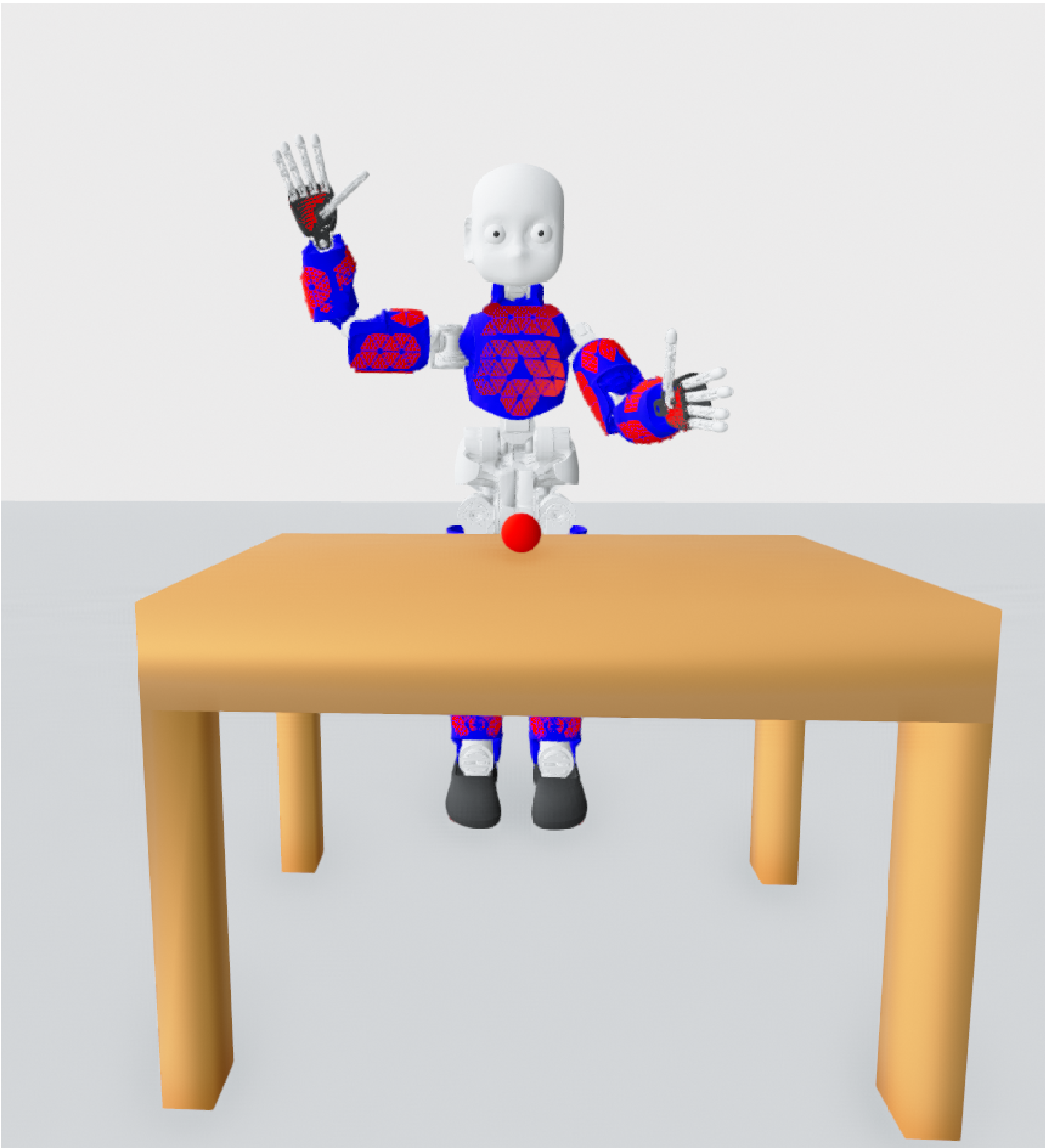
pyCub

Author: Lukas Rustler



CONTENTS:

CHAPTER

ONE

PYCUB DOCUMENTATION

pyCub is iCub humanoid robot simulator written in Python. It uses PyBullet for simulation and Open3D for visualization.

1.1 Installation

- Requires python3.11 to 3.12
  - newer python versions are now not supported due to incompatible with some dependencies
- We recommend using virtual environment when installing from PyPi or from source
  - ```
    python3 -m venv pycub_venv
    source pycub_venv/bin/activate
    OTHER_COMMANDS
    ```

1. (**Recommended**) Install from PyPi
   - `python3 -m pip install icub_pybullet`
2. Install from source
   - Pull this repository
   - ```
     cd PATH_TO_THE_REPOSITORY/icub_pybullet
     python3 -m pip install --upgrade pip
     python3 -m pip install .
     ```
3. Native Docker (GNU/Linux only)
   - see *Docker Native Version* section
4. VNC Docker
   - see *Docker VNC Version* section
5. Gitpod
   - open https://gitpod.io/#github.com/rustlluk/pycub and log in with GitHub account

1.2 Examples

- push_the_ball_pure_joints.py contains an example that shows how to control the robot in joint space
- push_the_ball_cartesian.py contains an example that shows how to control the robot in Cartesian space
- skin_test.py contains an example with balls falling the robot and skin should turn green on the places where contact occurs. You may want to slow the simulation a little bit to see that :)

1.3 Information

- documentation can be found at https://lukasrustler.cz/pyCub/documentation or in pycub.pdf
- presentation with description of functionality can be found at pycub presentation
- simulator code is in pycub.py
  - it uses PyBullet for simulation and provides high-level interface
- visualization code in visualizer.py
  - it uses Open3D for visualization as it is much more customizable than PyBullet default GUI

1.4 FAQ

1. You get some kind of error with the visualization, e.g., segmentation fault.
   1. Try to check graphics mesa/opengl/nvidia drivers as those are the most common source of problems for the openGL visualization
   2. Try Native Docker
   3. In given config your load set in GUI standard=False; web=True to enable web visualization
   4. Try VNC Docker
   5. Try Gitpod Docker
2. You get import errors, e.g. cannot load pycub from icub_pybullet
   1. Install from pip with `python3 -m pip install icub_pybullet`
   2. Install the package correctly with `python3 -m pip install .` from icub_pybullet directory of this repository
   3. put icub_pybullet directory to your PYTHONPATH

1.5 Docker

1.5.1 Native Version

- version with native GUI; useful when you have problems with OpenGL (e.g., usually some driver issues)
- only for GNU/Linux systems (Ubuntu, Mint, Arch, etc.)
  1. install docker-engine (**DO NOT INSTALL DOCKER DESKTOP**)

- **perform** post-installation steps

1. Build/Pull the Docker Image

- clone this repository

```
cd SOME_PATH
git clone https://github.com/rustlluk/pyCub.git
```

- pull the docker image (see *Parameters* for more parameters)

```
cd SPATH_TO_THE_REPOSITORY/Docker
./deploy.py -p PATH_TO_THE_REPOSITORY -c pycub -pu
```

- or, build the docker (see *Parameters* for more parameters)

```
cd SOME_PATH/pycub_ws/Docker
./deploy.py -p PATH_TO_THE_REPOSITORY -c pycub -b
```

- after you pull or build the container, you can run it next time as

```
./deploy.py -c pycub -e
```

- if you want to open new terminal in existing container, run

```
./deploy.py -c pycub -t
```

1.5.2 VNC Version

- this version works also on Windows and MacOS because it uses VNC server to show the GUI, i.e., the output will be shown on http://localhost:6080

  1. Install docker-engine (GNU/Linux only) or docker-desktop (all systems)

  - **perform** post-installation steps

  1. The same as for *Native Version*, but use `-vnc` option, e.g., to pull and run the image

  ```
   cd PATH_TO_THE_REPOSITORY/Docker
  ./deploy.py -p PATH_TO_THE_REPOSITORY -c pycub -pu -vnc
  ```

  2. run `start-vnc-sessions.sh` script in the container

  3. Open http://localhost:6080

1.5.3 Docker + PyCharm

Native Version:

You have two option:

1. Either run pycharm from docker
2. Open your pycharm on your host machine:
   - add ssh interpreter
     - user docker

    - ip can be localhost or ip where you run the docker
    - port 2222
  - uncheck automatic upload to remote folder
  - change remote path to /home/docker/pycub_ws #### VNC/Gitpod version
3. open pycharm from inside the container

1.5.4 Deploy Parameters

- `cd` to folder with Dockerfile
- `./deploy.py`
  - `-b` or `--build` when building
    * default: False
  - `-e` if you just want to run existing docker without building
    * default: False
  - `-p` or `--path` with path to current folder
    * default: “”
  - `-pu` or `--pull` to pull the image from dockerhub
    * default: False
  - `-c` or `--container` with desired name of the new, created container
    * default: my_new_docker
  - `-t` or `--terminal` to run new terminal in running docker session
    * default: False
  - `-pv` or `--python-version` to specify addition python version to install
    * default: 3.11
  - `-pcv` or `--pycharm-version` to specify version of pycharm to use
    * default: 2023.2.3
  - `-bi` or `--base-image` to specify base image that will be used
    * default: ubuntu:20.04
    * other can be found at hub.docker.com

Do this on computer where you will run the code. If you have a server you have to run it on the server over SSH to make things work properly.

1.5.5 Docker FAQ

- **you get error of not being in sudo group when running image**
  - check output of `id -u` command. If the output is not 1000 you have to build the image by yourself and can not pull it
    - this happens when your account is not the first one created on your computer
- **``sudo apt install something`` does not work**
  - you need to run `sudo apt update` first after you run the container for the first time
    - apt things are removed in Dockerfile, so it does not take unnecessary space in the image

1.6 Known bugs

- visualization with skin dies after ~65k steps
  - e.g., https://github.com/isl-org/Open3D/issues/4992

1.7 License

` .. image:: https://img.shields.io/badge/License-CC%20BY%204.0-lightgrey.svg

target
https://img.shields.io/badge/License-CC%20BY%204.0-lightgrey.svg

alt
CC BY 4.0

<http://creativecommons.org/licenses/by/4.0/>`_

This work is licensed under a Creative Commons Attribution 4.0 International License.

` .. image:: https://i.creativecommons.org/l/by/4.0/88x31.png

target
https://i.creativecommons.org/l/by/4.0/88x31.png

alt
CC BY 4.0

<http://creativecommons.org/licenses/by/4.0/>`_

CHAPTER

TWO

EXERCISES

2.1 Push the Ball

The goal is to hit the ball to push it as far as possible from **any part** of the table. The ball will always spawn at the same place. The robot can be moved with position or cartesian control. The trajectories should be collision free and max allowed velocity is 10 (rad/s).

The resulting distance is checked after 2 seconds.

2.1.1 Task

Implement `push_the_ball` function in exercise_1.py that will push the ball as far as possible from the table.

- the function takes one argument:
  - client - instance of pycub class that controls the simulation
    - do not create new one in your code!

2.1.2 Scoring

- points for distance are computed as: min(3, distance*2)
  - i.e., you get max 3 points if you push the ball 1.5m away

2.1.3 Requirements:

Those apply mainly for exercise_1_tester.py **to work correctly**:

- do not create new client instance, use the one that is passed as an argument
- do not rename the function or file
- **Do not** introduce artificial delays in the code, e.g., sleep() or using update_simulation() after the movement is done

Those apply so that you fulfill the exercise as intended:

- Trajectories must be collision free with the table (self-collisions of the robot are allowed)
- Max allowed velocity is 10 (rad/s)
- **Do not** turn of gravity

2.2 Smooth movements

The task is to perform smooth movements with the arms of the robot. The robot should be able to move in a straight line or a circle with a smooth trajectory. In other words, the movement should be continuous and without jerks.

2.2.1 Task

Implement function move() in exercise_2.py that will move the robot's arm in a straight line or a circle.

- the function takes four arguments:
  - client - instance of pycub class that controls the simulation
  - action - string "line" or "circle"
  - axis - list of ints. For "circle" it the length is always 1. For "line" the length can be from 1-3. Individual numbers on the list are axes along which the robot should move. For example, [0, 1] means that the robot should move in x- and y-axis.
  - r - list of floats. The same length as axis. Number in metres. For "circle" it is the radius of the circle. For "line" it is the length of the line in the given axis.
- example arguments:
  - action="circle", axis=[0], r=[0.01] - the end-effector should move in a circle around the x-axis with a radius of 0.01m
  - action="line", axis=[1], r=[-0.05] - the end-effector should follow a line in the y-axis with a distance of -0.05m
  - action="line", axis=[0, 1], r=[0.05, -0.05] - the end-effector should follow a line in the x-axis for 0.05cm and y-axis for -0.05m
    * it should be simultaneous movement in both axes, i.e., it will be one line
- the function **have to** return two arguments: start and end pose of the trajectory
  - both should be obtained with `client.end_effector.get_position()`

2.2.2 Scoring

- you will be given 10 random test cases
  - each will be awarded with 1 points
- both circle and line movements will be evaluated with three sub-criteria:
  - circle:
    * standard deviation of each point of the circle to center; must be < r*0.125
    * difference between expected and real radius; must be < r*0.1
    * mean distance of all points from the expected plane; must be < 0.1
  - line:
    * difference from real length to expected length; must be < 0.0075
    * angle between the expected and real line; must be <0.1

    * mean distance of all points from the expected line; must be < 0.01

- **the smoothness of the movements will is not checked by the automatic evaluation tool!**
- **the code should be as general as possible to work for any allowed input.** Meaning that it should not be, for example, a long if/else (switch) statement for each axis etc.

2.2.3 Requirements:

Those apply mainly for exercise_2_tester.py **to work correctly**:

- do not create new client instance, use the one that is passed as an argument
- do not rename the function or file
- **return two arguments:** start and end pose of the trajectory of type utils.Pose

Those apply so that you fulfill the exercise as intended:

- **Do not** turn of gravity
- Make the movement as smooth as possible

2.3 Gaze control

The goal of this task is to implement gaze controller for the iCub robot that is able to follow a ball moving on a table. The task is simplified to 2D case, i.e., the ball can move in x and y. The user is given the vector from the head to a ball (the where the robot should look) and the vector where the robot is looking right now.

2.3.1 Task

Implement gaze() function in exercise_3.py that will control the gaze of the robot to follow the ball.

- the function takes three arguments:
  - client - instance of pycub class that controls the simulation
  - head_direction - normalized vector representing the view of the robot, i.e., where the robot is looking
  - head_ball_direction - normalized vector representing the direction from the robot to the ball, i.e., where the robot should be looking
- the function should control joints in the necks of the robot to follow the ball
  - the move **must** non-blocking, i.e., parameter wait=False
- you **should not** call `update_simulation()` in this function

2.3.2 Scoring

- the ball will be moving for 10 (default) seconds and each step the error in degrees will be calculated
- maximum number of points is 10 (default; 10 runs of the code) and you can **lose** points based on the following:
  - if the mean absolute error is:
    - less than 0.5 degree - 0% of points
    - more than 0.5 and less than 1 degree - 50% of points
    - more than 1 and less than 5 degrees - 75% of points
    - more than 5 - 100% of points
  - if the max error is:
    - less than 2 degrees - 0% of points
    - more than 2 and less than 5 degrees - 25% of points
    - more than 5 and less than 10 degrees - 50% of points
    - more than 10 - 100% of points

2.3.3 Requirements

Those apply mainly for exercise_3_tester.py **to work correctly**:

- do not create new client instance, use the one that is passed as an argument
- do not rename the function or file
- **use non-blocking movements**, i.e., use parameter wait=False or use velocity control
- **do not call** `update_simulation()` in any of your code

Those apply so that you fulfill the exercise as intended:

- **Do not** turn of gravity

2.4 Resolved-Rate Motion Control

The goal of this task is to implement a Resolved-Rate Motion Control (RRMC) controller that moves the limbs away from collision using feedback from the artificial skin. There will be ball falling on the robot and you should move the correct body part away from it, i.e., move it against the normal of the contact.

2.4.1 Task

Implement process() function in exercise_4.py that will control the robot using RRMC. This time, the function is a method of a class. The reason is the option to store you own variables in the class.

- the class takes one argument:
  - client - instance of pycub class that controls the simulation
- the function should control joints using RRMC -> using velocity control
- you **should not** call `update_simulation()` in any of your code

You will be given four different scenarios with different body_parts being hit by the ball and with different number of balls. You should move every skin part that is in collision with any ball! But always use one contact per skin part, i.e., you should find the biggest cluster of activated taxels (skin points) on each skin part.

The tester script exercise_4_tester.py will run all four tests in sequence from the easier one. You need to manually close the visualization window after each test.

2.4.2 Scoring

There is no automatic evaluation for this task. But basically the task will be correctly fulfilled if the robot moves away from the ball. You can consult the video on top of this README to see possible outcomes. However, keep in mind that those are not the only correct solutions as the movements depends on the parameters selected in RRMC.

2.4.3 Requirements

Those apply mainly for exercise_4_tester.py **to work correctly**:

- do not create new client instance, use the one that is passed as an argument to the RRMC class
- do not rename the function, file or class
- **do not call** `update_simulation()` in any of your code

Those apply so that you fulfill the exercise as intended:

- **Do not** turn of gravity

2.5 Grasp It!

The goal of the task is to grasp a ball from the table. The exercise is divided into two parts:

- find the ball on a table in the image plane. The ball is in the visual field of the robot and is green.
- grasp the ball. Use the position of ball in image plane to compote its 3D position and grasp it.

2.5.1 Task

Implement find_the_ball() and grasp() functions in exercise_5.py that will find and grasp the ball. The function are methods of a class Grasper, that contains several useful methods. After grasp, you should move the ball up such that it is at least 5cm above the table.

The class takes two arguments

- client - instance of pyCub class that controls the simulation
- eye - name of the eye in which the camera is placed; 'l_eye' or 'r_eye' The two function have predefined arguments and returns:
- find_the_ball() - takes no arguments and should return (u, v) - center of the ball in image plane
- grasp() - takes one argument `center` (the 2D center returned by find_the_ball()) and should return 0 in case of successful grasp You are free to add any method or variable to the class, but only find_the_ball() and grasp() will be called.

The class Grasper contains several helpful methods, e.g., to get RGB and Depth image; to deproject the 2D point; to close fingers.

The ball is always green (0, 255, 0) and it's radius is 2.5cm.

2.5.2 Scoring

- If the ball is 5cm above the table and in the hand of the robot after you return from grasp(), you passed the test

2.5.3 Requirements

Those apply mainly for exercise_5_tester.py **to work correctly**:

- do not create new client instance, use the one that is passed as an argument to Grasper class
- do not rename the functions find_the_ball(), grasp() or the file
- The ball must be 5cm above the table and in the hand of the robot after grasp() returns

Those apply so that you fulfill the exercise as intended:

- **Do not** turn of gravity

CHAPTER

THREE

ICUB_PYBULLET

3.1 pyCub

The main class and utils for pyCub simulator

Author
Lukas Rustler

`class icub_pybullet.pycub.`**`EndEffector`**`(`*name: str*, *client: int*`)`

Bases: `object`

Help function for end-effector encapsulaation

Parameters

- **`name`** (*`str`*) – name of the end-effector
- **`client`** (*`pointer to pyCub instance`*) – parent client

`get_position()` → *Pose*

Function to get current position of the end-effector

`class icub_pybullet.pycub.`**`Joint`**`(`*name: str*, *robot_joint_id: int*, *joints_id: int*, *lower_limit: float*, *upper_limit: float*, *max_force: float*, *max_velocity: float*`)`

Bases: `object`

Help class to encapsulate joint information

Parameters

- **`name`** (*`str`*) – name of the joint
- **`robot_joint_id`** (*`int`*) – id of the joint in pybullet
- **`joints_id`** (*`int`*) – id of the joint in pycub.joints
- **`lower_limit`** (*`float`*) – lower limit of the joint
- **`upper_limit`** (*`float`*) – upper limit of the joint
- **`max_force`** (*`float`*) – max force of the joint
- **`max_velocity`** (*`float`*) – max velocity of the joint

`class icub_pybullet.pycub.`**`Link`**`(`*name: str*, *robot_link_id: int*, *urdf_link: int*`)`

Bases: `object`

Help function to encapsulate link information

Parameters

- **name** (*str*) – name of the link
- **robot_link_id** (*int*) – id of the link in pybullet
- **urdf_link** (*int*) – id of the link in pycub.urdfs["robot"].links

class icub_pybullet.pycub.**pyCub**(*config: str | None = 'default.yaml'*)

Bases: BulletClient

Client class which inherits from BulletClient and contains the whole simulation functionality

Parameters
config (*str, optional, default="default.yaml"*) – path to the config file

static **bbox_overlap**(*b1_min: list*, *b1_max: list*, *b2_min: list*, *b2_max: list*) → bool

Function to check whether two bounding boxes overlap

Parameters

- **b1_min** (*list*) – min bbox 1
- **b1_max** (*list*) – max bbox 1
- **b2_min** (*list*) – min bbox 2
- **b2_max** (*list*) – max bbox 2

Returns
whether the boxes overlap

Return type
bool

compute_jacobian(*chain: Literal['left_arm', 'right_arm', 'left_leg', 'right_leg', 'torso', 'head']*, *start: str | None = None*, *end: str | None = None*) → Tuple[array, array]

Help function to compute a Jacobian using RTB of a given chain with optional start and end

Parameters

- **chain** (*str*) – name of the chain
- **start** (*str*) – name of the start link
- **end** (*str*) – name of the end link

Returns
Jacobian and list of joints used

Return type
np.array, np.array

compute_skin() → None

Function to emulate skin activations using ray casting.

contactPoints = {'DISTANCE': 8, 'FLAG': 0, 'FORCE': 9, 'FRICTION1': 10, 'FRICTION2': 12, 'FRICTIONDIR1': 11, 'FRICTIONDIR2': 13, 'IDA': 1, 'IDB': 2, 'INDEXA': 3, 'INDEXB': 4, 'NORMAL': 7, 'POSITIONA': 5, 'POSITIONB': 6}

create_urdf(*object_path: str*, *fixed: bool*, *color: List[float]*, *suffix: str | None = ''*)

Creates a URDF for the given .obj file

Parameters

- **object_path** (*str*) – path to the .obj

- **fixed** (*bool*) – whether the object is fixed in space
- **color** (*list of 3 floats*) – color of the object
- **suffix** (*str, optional, default=""*) – suffix to add to the object name

dynamicsInfo = {'BODYTYPE': 10, 'DAMPING': 8, 'FRICTION': 1, 'INERTIAOR': 4, 'INERTIAPOS': 3, 'INTERTIADIAGONAL': 2, 'MARGIN': 11, 'MASS': 0, 'RESTITUTION': 5, 'ROLLINGFRICTION': 6, 'SPINNINGFRICTION': 7, 'STIFFNESS': 9}

find_joint_id(*joint_name: JOINTS | JOINTS_IDS*) → Tuple[int, int]

Help function to get indexes from joint name of joint index in self.joints list

Parameters
joint_name (*str or int*) – name or index of the link

Returns
joint id in pybullet and pycub space

Return type
int, int

find_link_id(*mesh_name: str*, *robot: int | None = None*, *urdf_name: str | None = 'robot'*) → int

Help function to find link id from mesh name

Parameters

- **mesh_name** (*str*) – name of the mesh (only basename with extension)
- **robot** (*int, optional, default=None*) – robot pybullet id
- **urdf_name** (*str, optional, default="robot"*) – name of the object in pycub.urdfs

Returns
id of the link in pybullet space

Return type
int

find_processes_by_name() → List[int]

Help function to find PIDs of processes with the parent name

Returns
list of matching PIDs

Return type
list of ints

get_camera_depth_images(*eyes: str | List[str] | None = None*) → dict

Gets the images from enabled eye cameras

Parameters
eyes (*str or list of str, optional*) – name of eye/eyes to get images for

Returns
dictionary with eye as keys and np.array of images as values

Return type
dict

get_camera_images(*eyes: str | List[str] | None = None*) → dict

Gets the images from enabled eye cameras

Parameters
`eyes` (*str or list of str, optional*) – name of eye/eyes to get images for

Returns
dictionary with eye as keys and np.array of images as values

Return type
dict

`static get_chains()` → Tuple[dict, dict]

Function to get chains (and corresponding chains of joints) of the robot

Returns
chains of links and chains of joints

Return type
dict, dict

`get_joint_state`(*joints: JOINTS | List[JOINTS] | JOINTS_IDS | List[JOINTS_IDS] | None = None*, *allow_error: bool | None = False*) → List[list]

Get the state of the specified joints

Parameters

- `joints` (*int or list, optional, default=None*) – joint or list of joints to get the state of
- `allow_error` (*bool, optional, default=False*) – whether to allow errors (non-existing joints); useful for Jacobian computation

Returns
list of states of the joints

Return type
list

`init_robot()` → Tuple[int, List[*Joint*], List[*Link*]]

Load the robot URDF and get its joints' information

Returns
robot, its joints and links

Return type
int, list of *Joint*, list of *Link*

`init_urdfs()` → None

Function to load URDFs of other objects

Returns

Return type

`is_alive()` → bool

Checks whether the engine is still running

Returns
True when running

Return type
bool

`jointInfo = {'AXIS': 13, 'DAMPING': 6, 'FLAGS': 5, 'FRICTION': 7, 'INDEX': 0, 'LINKNAME': 12, 'LOWERLIMIT': 8, 'MAXFORCE': 10, 'MAXVELOCITY': 11, 'NAME': 1, 'PARENTINDEX': 16, 'PARENTORN': 15, 'PARENTPOS': 14, 'QINDEX': 3, 'TYPE': 2, 'UINDEX': 4, 'UPPERLIMIT': 9}`

`jointStates = {'FORCES': 2, 'POSITION': 0, 'TORQUE': 3, 'VELOCITY': 1}`

kill_open3d() → None

A bit of a workaround to kill open3d, that seems to hang for some reason.

Returns

Return type

`linkInfo = {'ANGVEL': 7, 'INERTIAORI': 3, 'INERTIAPOS': 2, 'LINVEL': 6, 'URDFORI': 5, 'URDFPOS': 4, 'WORLDORI': 1, 'WORLDPOS': 0}`

motion_done(*joints: JOINTS | List[JOINTS] | JOINTS_IDS | List[JOINTS_IDS] | None = None*, *check_collision=True*) → bool

Checks whether the motion is done.

Parameters

- **joints** (*int or list, optional, default=None*) – joint or list of joints to get the state of
- **check_collision** (*bool, optional, default=True*) – whether to check for collision during motion

Returns
True when motion is done, false otherwise

Return type
bool

move_cartesian(*pose: Pose*, *wait: bool | None = True*, *velocity: float | None = 1*, *check_collision: bool | None = True*, *timeout: float | None = None*) → None

Move the robot in cartesian space by computing inverse kinematics and running position control

Parameters

- **pose** (`utils.Pose`) – desired pose of the end effector
- **wait** (*bool, optional, default=True*) – whether to wait for movement completion
- **velocity** (*float, optional, default=1*) – joint velocity to move with
- **check_collision** (*bool, optional, default=True*) – whether to check for collisions during motion
- **timeout** (*float, optional, default=10*) – timeout for the motion

move_position(*joints: JOINTS | List[JOINTS] | JOINTS_IDS | List[JOINTS_IDS]*, *positions: float | List[float]*, *wait: bool | None = True*, *velocity: float | None = 1*, *set_col_state: bool | None = True*, *check_collision: bool | None = True*, *timeout: float | None = None*) → None

Move the specified joints to the given positions

Parameters

- **joints** (*int, str, list of int, list of str*) – joint or list of joints to move
- **positions** (*float or list; same length as joints*) – position or list of positions to move the joints to

- **wait** (*bool, optional, default=True*) – whether to wait until the motion is done
- **velocity** (*float, optional, default=1*) – velocity to move the joints with
- **set_col_state** (*bool, optional, default=True*) – whether to reset collision state
- **check_collision** (*bool, optional, default=True*) – whether to check for collision during motion
- **timeout** (*float, optional, default=10*) – timeout for the motion

move_velocity(*joints: JOINTS | List[JOINTS] | JOINTS_IDS | List[JOINTS_IDS], velocities: float | List[float]*) → None

Move the specified joints with the specified velocity

Parameters

- **joints** (*int or list*) – joint or list of joints to move
- **velocities** (*float or list; same length as joints*) – velocity or list of velocities to move the joints to

prepare_log() → str

Prepares the log string

Returns
log string

Return type
str

print_collision_info(*c: list | None = None*)

Help function to print collision info

Parameters
c (*list, optional, default=None*) – one collision; if None print all collisions

run_vhacd(*robot: bool | None = True*) → None

Function to run VHACD on all objects in loaded URDFs, and to create new URDFs with changed collision meshes

Parameters
robot (*bool, optional, default=True*) – whether to run VHACD on the robot

static scale_bbox(*bbox: list, scale: float*) → Tuple[array, array]

Function to scale the bounding box

Parameters

- **bbox** (*list*) – list of min and max bbox
- **scale** (*float*) – scale factor

Returns
new min and max bbox

Return type
list, list

stop_robot(*joints: JOINTS | List[JOINTS] | JOINTS_IDS | List[JOINTS_IDS] | None = None*) → None

Stops the robot

`toggle_gravity()` → None

Toggles the gravity

`update_simulation`(*sleep_duration: float | None | None = -1*) → None

Updates the simulation

Parameters

sleep_duration (*float or None, optional, default=-1*) – duration to sleep before the next simulation step

`visualShapeData = {'COLOR': 7, 'DIMS': 3, 'FILE': 4, 'GEOMTYPE': 2, 'ID': 0, 'LINK': 1, 'ORI': 6, 'POS': 5, 'TEXTURE': 8}`

`wait_motion_done`(*sleep_duration: float | None = 0.01*, *check_collision: bool | None = True*) → None

Help function to wait for motion to be done. Can sleep for a specific duration

Parameters

- **sleep_duration** (*float, optional, default=0.01*) – how long to sleep before running simulation step
- **check_collision** (*bool, optional, default=True*) – whether to check for collisions during motion

3.2 utils

Utils for pyCub simulator

Author

Lukas Rustler

`class icub_pybullet.utils.Config`(*config_path: str*)

Bases: `object`

Class to parse and keep the config loaded from yaml file

Parameters

config_path (*str*) – path to the config file

`set_attribute`(*attr: str*, *value: Any*, *reference: int*) → int

Function to recursively fill the instance variables from dictionary. When value is non-dict, it is directly assigned to a variable. Else, the dict is recursively parsed.

Parameters

- **attr** (*str*) – name of the attribute
- **value** (*str, float, int, dict, list, ... - and other that can be loaded from yaml*) – value of the attribute
- **reference** (*pointer or whatever it is called in Python*) – reference to the parent class. "self" for the upper attributes, pointer to namedtuple for inner attributes

Returns

0

Return type

int

class icub_pybullet.utils.**CustomFormatter**(*fmt=None*, *datefmt=None*, *style='%'*, *validate=True*)

Bases: `Formatter`

Custom formatter that assigns colors to logs From https://stackoverflow.com/a/56944256

Initialize the formatter with specified format strings.

Initialize the formatter either with the specified format string, or a default as described above. Allow for specialized date formatting with the optional datefmt argument. If datefmt is omitted, you get an ISO8601-like (or RFC 3339-like) format.

Use a style parameter of '%', '{' or '$' to specify that you want to use one of %-formatting, `str.format()` (`{}`) formatting or `string.Template` formatting in your format string.

Changed in version 3.2: Added the `style` parameter.

FORMATS = {10: '\x1b[97;10m%(module)s %(levelname)s: %(message)s\x1b[0m', 20: '\x1b[97;10m%(module)s %(levelname)s: %(message)s\x1b[0m', 30: '\x1b[33;20m%(module)s %(levelname)s: %(message)s\x1b[0m', 40: '\x1b[31;20m%(module)s %(levelname)s: %(message)s\x1b[0m', 50: '\x1b[31;1m%(module)s %(levelname)s: %(message)s\x1b[0m'}

bold_red = '\x1b[31;1m'

format(*record: LogRecord*) → str

Format the specified record as text.

The record's attribute dictionary is used as the operand to a string formatting operation which yields the returned string. Before formatting the dictionary, a couple of preparatory steps are carried out. The message attribute of the record is computed using LogRecord.getMessage(). If the formatting string uses the time (as determined by a call to usesTime(), formatTime() is called to format the event time. If there is exception information, it is formatted using formatException() and appended to the message.

grey = '\x1b[97;10m'

red = '\x1b[31;20m'

reset = '\x1b[0m'

yellow = '\x1b[33;20m'

class icub_pybullet.utils.**Pose**(*pos: list*, *ori: list*)

Bases: `object`

Mini help class for Pose representation

Init function that takes position and orientation and saves them as attributes

Parameters

- **pos** (*list*) – x,y,z position
- **ori** (*list*) – rpy orientation

to_string() → str

class icub_pybullet.utils.**URDF**(*path: str*)

Bases: `object`

Class to parse URDF file

Parameters

path (*str*) – path to the URDF file

`ROOT_TAGS = []`

dereference() → None

Make parent/child again as names to allow urdf write

find_root_tags() → None

Finds tags that are 'root', i.e., they have child 'inside'

fix_urdf() → None

Fix the URDF file by converting non-mesh geometries to mesh and saving them as .obj files. If changes were made, write the new URDF to a file.

make_references() → None

Make parent/child in joint list as references to the given link

read(*el: <module 'xml.etree.ElementTree' from '/usr/lib/python3.8/xml/etree/ElementTree.py'>*, *parent: <module 'xml.etree.ElementTree' from '/usr/lib/python3.8/xml/etree/ElementTree.py'>*) → None

Recursive function to read the URDF file. When there are no children, it reads the attributes and saves them.

Parameters

- **el** (*xml.etree.ElementTree.Element*) – The current element in the XML tree.
- **parent** (*xml.etree.ElementTree.Element*) – The parent element in the XML tree.

write_attr(*attr_name: str*, *attr: Any*, *level: int | None = 1*, *skip_header: bool | None = False*) → None

Write an attribute to the new URDF string.

Parameters

- **attr_name** (*str*) – The name of the attribute.
- **attr** (*any*) – The attribute value.
- **level** (*int, optional, default=1*) – The indentation level for the attribute.
- **skip_header** (*bool, optional, default=False*) – Whether to skip writing the attribute header.

write_urdf() → None

Write the URDF object to a string.

3.3 visualizer

Visualization utils for pyCub simulator

Author

Lukas Rustler

class icub_pybullet.visualizer.**Visualizer**(*client: pyCub*)

Bases: `object`

Class to help with custom rendering

Parameters

client (*pyCub*) – pyCub instance

class EyeWindow(*eye: str*, *parent: Visualizer*)

Bases: `object`

Class to handle windows for eye rendering

Parameters

- **eye** (*str*) – name of the eye
- **parent** (`Visualizer`) – The parent class (Visualizer).

MENU_IDS = {'l_eye': [2, 3, 8], 'r_eye': [4, 5, 9]}

POSITIONS = {'l_eye': [320, 560], 'r_eye': [0, 560]}

get_depth_image() → None

Small function to get image from open3d

Returns

Return type

get_image() → None

Small function to get image from open3d

Returns

Return type

on_close() → bool

Small function to delete the window from the parent class

on_mouse(*event: MouseEvent*) → int

Small function to ignore mouse events

Parameters

event (*gui.MouseEvent*) – Mouse event

save_depth_image(*im: Image*) → None

Callback to get images from open3d

Parameters

im (*o3d.geometry.Image*) – the image to be saves

save_image(*im: Image*) → None

Callback to get images from open3d

Parameters

im (*o3d.geometry.Image*) – the image to be saves

save_images() → None

Function to save stream of images to file

unproject(*u*, *v*, *d*)

class MenuCallback(*menu_id: int*, *parent: Visualizer*)

Bases: `object`

Class to handle menu callbacks.

Initialize the MenuCallback class.

Parameters

- **menu_id** (*int*) – The id of the menu.
- **parent** (*pointer to the class of visualizer.Visualizer type*) – The parent class (Visualizer).

input_completed(*text: str | None = None*)

Callback for the dialog

Parameters

text (*str*) – input text

Returns

Return type

save_image(*im: Image*, *mode: int*) → None

Save the image. It shows FileDialog to find path for image save. It saves it with the current resolution of the window.

Parameters

- **im** (*o3d.geometry.Image*) – The image to be saved.
- **mode** (*int*) – The mode of the image. 0 for RGB, 1 for depth.

wait_for_dialog_completion() → None

Help function to keep the gui loop running

find_xyz_rpy(*mesh_name: str*, *urdf_name: str | None = 'robot'*) → Tuple[list, list, float, str]

Find the xyz, rpy and scales values.

Parameters

- **mesh_name** (*str*) – The name of the mesh.
- **urdf_name** (*str, optional, default="robot"*) – The name of the urdf.

Returns

The xyz, rpy, and scales, link_name

Return type

list, list, float, str

read_info(*obj_id: int*) → int

Read info from PyBullet

Parameters

obj_id (*int*) – id of the object; given by pybullet

Returns

0 for success

Return type

int

render() → None

Render all the things

show_first(*urdf_name: str | None = 'robot'*) → None

Show the first batch of meshes in the visualizer. It loads the meshes and saves the to dict for quicker use later

Parameters

urdf_name (*str, optional, default="robot"*) – The name of the urdf to be used.

show_mesh() → None

Function to parse info about meshes from PyBullet

CHAPTER

FOUR

EXAMPLES

4.1 push_the_ball_cartesian

Example of moving the robot in cartesian space to push the ball. It is more robust than the pure joint control.

Author
Lukas Rustler

`icub_pybullet.examples.push_the_ball_cartesian.main()` → NoReturn

Main function to run the example

Returns

Return type

`icub_pybullet.examples.push_the_ball_cartesian.push_the_ball(`*client:* pyCub`)` → None

Example function to move the ball with cartesian control.

Parameters
`client` (pyCub) – instance of pyCub

Returns

Return type

4.2 push_the_ball_pure_joints

Example of how to push the ball from the table using only pure joint control. It works without planner of collisions detection/avoidance. It is not very robust, and it is laborious, but it is a good starting point for your own experiments.

Author
Lukas Rustler

`icub_pybullet.examples.push_the_ball_pure_joints.main()` → NoReturn

Main function to run the example

Returns

Return type

`icub_pybullet.examples.push_the_ball_pure_joints.push_the_ball(`*client:* pyCub`)` → None

Example function to push the ball from the table with joint control.

Parameters
`client` (pyCub) – instance of pyCub

Returns

Return type

4.3 skin_test

Script to the test the skin sensors. Balls falling to the skin and turning activated point to green should be seen.

Author
Lukas Rustler

`icub_pybullet.examples.skin_test.main()` → NoReturn

Main function to run the example

Returns

Return type

PYTHON MODULE INDEX

i

INDEX

K

L

M

O

P

R

S

T

U

V

W

Y