Institute for Cognitive Systems
Technische Universität München
Dr.-Ing. Rogelio Guadarrama

Modeling and control of legged robots
Tutorial 2: Modeling and simulation

1 Introduction

Bullet 1 is an open-source physics engine for soft and rigid body simulations. It is supported by y large comunity of developers and applied on different comercial products and research projects. Pybullet is a Python module that wraps the Bullet simulator and provides useful tools for robotics and machine learning applications.

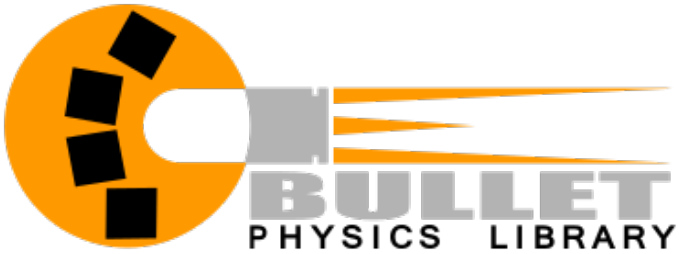

Figure 1: Bullet physics library.

Extensive documentation on the functionality of Pullet and Pybullet can be found in it main web page pybullet.org. As it is developed for several applications, we will focus on the features needed to implement simulations of humanoids and legged robots. Pybullet requires several steps to set up a simulation and access the state of the system and command the actuators of a robot. In this tutorial, we will create a bridge between Pybullet and the Pinocchio library to load, and control a full-size humanoid robot.

a Our Pybullet wrapper

Find the following items in the template materials of this tutorial:

- A ROS meta-package called `reemc`. It contains the descriptors to simulate the REEM-C robot from Pal Robotics.
- A ROS meta-package called `talos`. It contains the descriptors to simulate the Talos robot from Pal Robotics.
- The needed files to create a ROS package called `simulator`. It contains a python module that wraps Pinocchio and Pybullet.
- An example python script called `t2_temp.py`. Read carefully the code in this example.

Pybullet uses internally a different model than Pinocchio. The `simulator` package was designed to cope with this problem. For the control aplications that we will cover in this course, we will use the standards of Pinocchio. The Pybullet wrapper will help you read the state from Pybullet and send the actuator commands that your controller compute. Let us then revise the data structures of Pinocchio.

b Pinocchio data

Consider a robot with n actuators whose joint state is defined by $q \in \mathbb{R}^n$. If the robot is a free floating base system, the non actuated degrees of freedom of the floating base must be appended at the beginning of the position state vector as follows

$$\mathbf{q} = [\mathbf{p}_b \; \mathbf{Q}_b \; q]^\top$$

where $\mathbf{p}_b \in \mathbb{R}^3$ is the Cartesian position of the floating base link w.r.t. the world coordinate frame, $\mathbf{Q}_b \in \mathbb{Q}$ is a quaternion describing the floating base link orientation and $\mathbf{q}$ is the full system position state. Look at the example script in the template on how the `q_home` is constructed. You can get this vector from the Pybullet wrapper with the access function `robot.q()`.

The velocity state vector of the system in the Pinocchio library is defined as follows

$$\nu = [\mathbf{v}_b \; \omega_b \; \dot{q}]^\top$$

where $\mathbf{v}_b \in \mathbb{R}^3$ is the linear velocity of the floating base, $\omega_b \in \mathbb{R}^3$ is the angular velocity of the floating base, and $\dot{q} \in \mathbb{R}^n$ is the joint velocity vector. You can get this vector from the Pybullet wrapper with the access function `robot.v()`.

When you design controllers, the actuators commands must be stored in an n (τ) size vector and feed into the simulator with the function `robot.setActuatedJointTorques(tau)`.

All these vectors are internally handled as numpy arrays by the different modules. Therefore, all the vector and matrix operators of numpy can be applied. You can access the Pinocchio model of the `robot` wrapper as `robot._model` and get all the functionality of the Pinocchio python library.

2 Homework

You can use the workspace created in Tutorial 1 to implement the solutions of this tutorial. Before starting with the new tasks, copy the `reemc` and `talos` packages in the `src` folder of your workspace. Then, build your workspace once.

a Exercise 1: Create a simulation

1. Use the content of the `simulator` folder in the template to create a ROS Python package in the workspace that exports a python module / library with the same name.
2. Create a package in the workspace called `bullet_sims` that depends on the `reemc_description`, `talos_description`, and the `simulator` packages.

3. Copy the `t2_temp.py` script in the `scripts` folder of the new package and declare it in the `CMakeLists.txt` file as rospy node.
4. Build the workspace once and run the `t2_temp.py` script. You should see the Talos robot loaded in Pybullet and collapsing on the ground due to the gravity. The robot is commanded $\tau = 0$, therefore no torque is driven in the actuators.
5. Read the documentation of the Pinocchio library for the functions `ccrba`, `crba`, and `nonLinearEffects`.
6. Add the following line after building the model to build the Pinocchio data structure of the model.

```
data = robot._model.createData()
```

7. Compute and printout the complete Inertia Matrix, and the non linear effects vector of the model once.
8. Is the `t2_temp.py` script a ROS node? why?

b Exercise 2: Joint space controller

1. Make a copy of `t2_temp.py` with the name `t21.py` and declare it in the `CMakeLists.txt` file. Apply the following steps to `t21.py`.
2. Implement a joint space PD controller of the form:

$$\tau = K_p(q_d - q) - K_d\dot{q}$$

where $K_p, K_d \in \mathbb{R}^{n\times n}$ are gain matrices. Set $q_d = 0$

3. Run the simulator and see the results.
4. Tune K_p, K_d to enable a stiff robot body posture without much overshot. (Hint) the legs hold larger load, therefore the gains acting on them must be stronger than the upper body (around 3 times higher at least).

c Exercise 3: Home posture controller

1. Make a copy of `t21.py` with the name `t22.py` and declare it in the `CMakeLists.txt` file. Apply the following steps to `t22.py`.
2. Implement a function to spline the join positions from an initial state q_{ini} to a home posture q_{home} in a specified amount of time t. You can use the `interpolate` function from Pinocchio.
3. Build q_{home} with $[0, 0, -0.44, 0.9, -0.45, 0]$ as joint positions for both legs and $[0, -0.24, 0, -1, 0, 0, 0, 0]$ as joint positions for both arms. This posture is stable for the Talos Robot.
4. Modify you controller to feed the q spline as q_d.
5. Play the simulation again and see the robot setting its home position. If the controller is properly tuned, the robot should not fall.

d Exercise 4: Robot state visualization

1. Make a copy of `t22.py` with the name `t23.py` and declare it in the `CMakeLists.txt` file. Apply the following steps to `t23.py`.
2. Implement a ROS publisher to publish q, $\dot{q}$, and τ in the `joint_states` topic. You can get the list of joint names from the Python wrapper with the access function `robot.actuatedJointNames()`.
3. Call your publishing function at the end of the `while` loop in the main execution thread. Keep in mind that the simulator runs at 1 kHz update rate and the publishing of a large topic takes several cycles and can exceed the capacity of your PC. Therefore, you should throttle down the publishing to 30 Hz only.
4. Take a look a the documentation of the `robot_state_publisher` package and the RobotModel Rviz plugin.
5. Create a launch file in your `ros_visuals` package that loads the same URDF you used for Pinocchio in the `robot_description` parameter, launches the `robot_state_publisher` and the Rviz window with a config file called `talos.rviz`. Save your launch file as `talos_rviz.launch`.
6. Launch your simulation, and the rviz launch file in another terminal. In the Rviz window, load a RobotModel plugin to visualize the robot motions. Add the TF tree, and set a transparency of 0.8 in the robot model to visualize the coordinate frames of the robot links.
7. Save the changes in the rviz configuration file.

e Optional: Try the Reemc

Try loading the REEM-C robot in the Pybullet simulation with the Pinocchio bridge, adapting the exercises 1-4.

3 What to deliver?

This tutorial belongs to the first block deliverable. You must deliver the `ros_visuals`, `simulation` and `bullet_sims` packages with all the python scripts, config and launch files including your answers to the questions in a text file.

- Write clean readable code with comments highlighting the sections solving the steps of this tutorial.
- Do not deliver compiled binaries (the `build` and `devel` folders of the workspace). Only deliver the source code and needed files (launch, config, rviz configs, etc.).
- Do not copy the solution from other students.
- You can use any free online tool to help you solve this tutorial.
- It is recommended to use a python debug console (such as the one included in vscode) to solve this tutorial.
- Do not submit the solution of this tutorial yet. After the next tutorial, you will deliver the cumulated solution of block one.