

---

# **ROS-PyBullet Interface**

**Christopher E. Mower**

**Nov 15, 2022**



## CONTENTS

<b>1</b>	<b>Overview</b>	<b>3</b>
<b>2</b>	<b>Installation</b>	<b>5</b>
<b>3</b>	<b>Examples</b>	<b>7</b>
<b>4</b>	<b>Main Configuration</b>	<b>11</b>
<b>5</b>	<b>Create Virutal Worlds</b>	<b>17</b>
<b>6</b>	<b>Communication with ROS</b>	<b>21</b>
<b>7</b>	<b>Additional Features</b>	<b>23</b>
<b>8</b>	<b>Cite</b>	<b>29</b>
<b>9</b>	<b>Development</b>	<b>31</b>
<b>10</b>	<b>Acknowledgements</b>	<b>33</b>



The ROS-PyBullet Interface is a bridge between the popular physics library [PyBullet](#) and the [Robot Operating System \(ROS\)](#).



## OVERVIEW

Reliable contact simulation is a key requirement for the community developing (semi-)autonomous robots. Simulation, data collection, and straight-forward interfacing with robot hardware and multi-modal sensing will enable the development of robust machine learning algorithms and control approaches for real world applications. The ROS-PyBullet Interface is a framework that provides a bridge between the popular Robot Operating System (ROS) with a reliable impact/contact simulator Pybullet. Furthermore, this framework provides several interfaces that allow humans to interact with the simulator that facilitates Human-Robot Interaction in a virtual world. Tutorial slides found [here](#).

The main features of the framework is summarized as follows.

1. Online, full-physics simulation of robots using a reliable simulator. The framework uses Pybullet to enable well founded contact simulation.
2. Integration with ROS ecosystem. Tracking the state of the world using Pybullet is integrated with ROS so that users can easily plugin their work in a similar way that a real system might be controlled.
3. Several interfaces for humans. Providing demonstrations from humans requires an interface to the simulated environment. In addition, utilizing a haptic interface the human can directly interact with the virtual world.
4. Modular and extensible design. Our proposed framework adopts a modular and highly extensible design paradigm using Python. This makes it easy for practitioners to develop and prototype their methods for several tasks.
5. Data collection with standard ROS tools. Since the framework provides an interface to ROS we can use common tools for data collection such as ROS bags and `rosbag_pandas`.
6. Easily integrates with hardware. Tools are provided to easily remap the virtual system to physical hardware.





## INSTALLATION

The following describes how to install the ROS-PyBullet Interface.

### 2.1 Requirements

- ROS Noetic
  - Melodic, and past version should work fine (only Melodic has been tested) so long as you configure ROS to use Python 3
  - ROS2 is currently unsupported, however this is in-development
  - `catkin_tools`
  - `roscpp`
  - `roscpp`
- Ubuntu 20.04
  - Other versions should work, provided Python 3 is installed
- Python 3
- `urdf_parser_py`, see [here](#).

### 2.2 From binaries

Currently, this is *in-progress*.

### 2.3 From source

1. Create a `catkin` workspace or use an existing workspace. `catkin_tools` is the preferred build system.
2. `cd` to the `src` directory of your catkin workspace.
3. Clone this repository: `$ git clone https://github.com/ros-pybullet/ros_pybullet_interface.git`
4. Install source dependencies: `$ roscpp . --catkin --nobuild`
5. Install binary dependencies: `$ rosdep update ; rosdep install --from-paths ./ -iry`
6. Compile the workspace: `$ catkin build -s`

7. Source the workspace: `$ source $(catkin locate)/devel/setup.bash`

Now you should be able to run the *examples*.

## EXAMPLES

The examples for the ROS-PyBullet Interface are collected in a dedicated ROS package [rpbi\\_examples](#). The following gives details for each example and shows how to run them.

### 3.1 Basic Examples

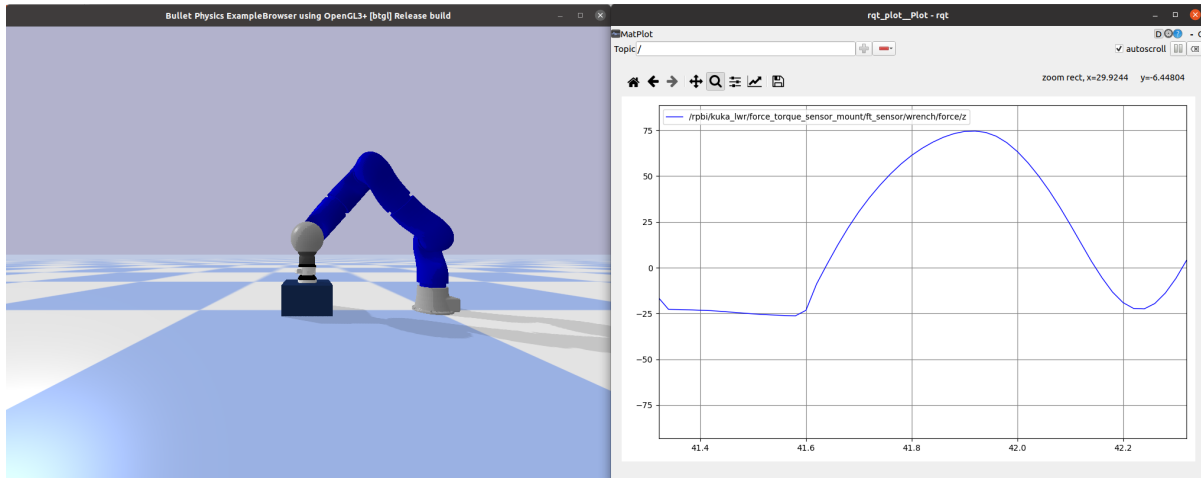
```
$ roslaunch basic_example_[NAME].launch
```

The basic examples simply demonstrate the current robots that can be loaded into PyBullet out-of-the-box. Each example loads the given robot, and a [node](#) that generates a standardized motion on the robot. Some of the basic examples demonstrate different features of the library (e.g. recording a video, loading a URDF from the ROS parameter `robot_description`). The following list links to the launch file for all the currently available basic examples.

- [Kuka LWR](#)
- [Talos](#)
- [Kinova](#)
- [Human model](#)
- [Nextage](#)

*Note*, the Kuka LWR example additionally demonstrates how to start recording videos and also how to attach a Force-Torque sensor to a robot joint.

## 3.2 Human Interaction

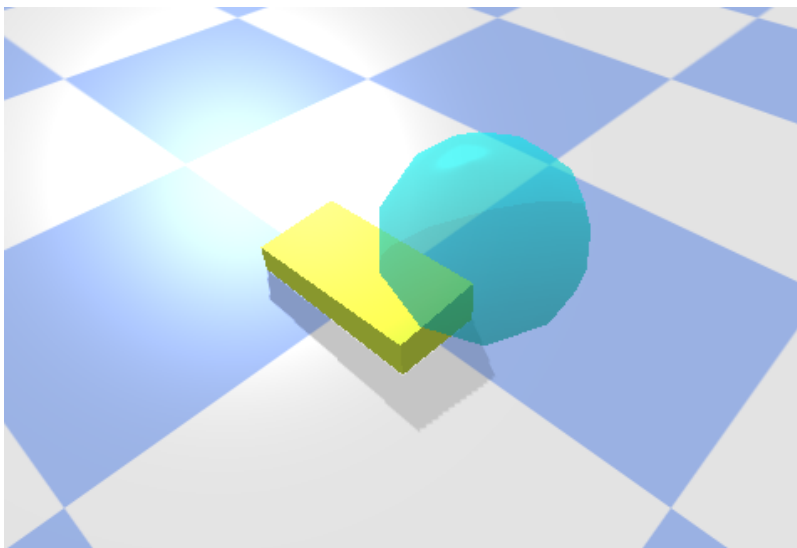


```
$ roslaunch rpbi_examples human_interaction.launch
```

The goal of this example is to demonstrate how virtual forces, generated by the PyBullet simulator, can be rendered to the human via a haptic device. We have integrated the [3D Systems Touch X Haptic Device](#) into the framework, this is a necessary piece of hardware to run the example. Ensure you connect your device and that it is setup by following the instructions in the [RViMLab/geomagic\\_touch\\_x\\_ros](#) repository.

When the example is launched, allow the haptic device to move to a nominal configuration. After it is calibrated and initialized you will be able to move the device up and down. The robot will track your movements in the z-axis until it reaches a block below the end-effector. The forces detected from a Force-Torque sensor, attached to the robot wrist, is rendered as feedback to the user - you will be able to feel the reaction forces through the device.

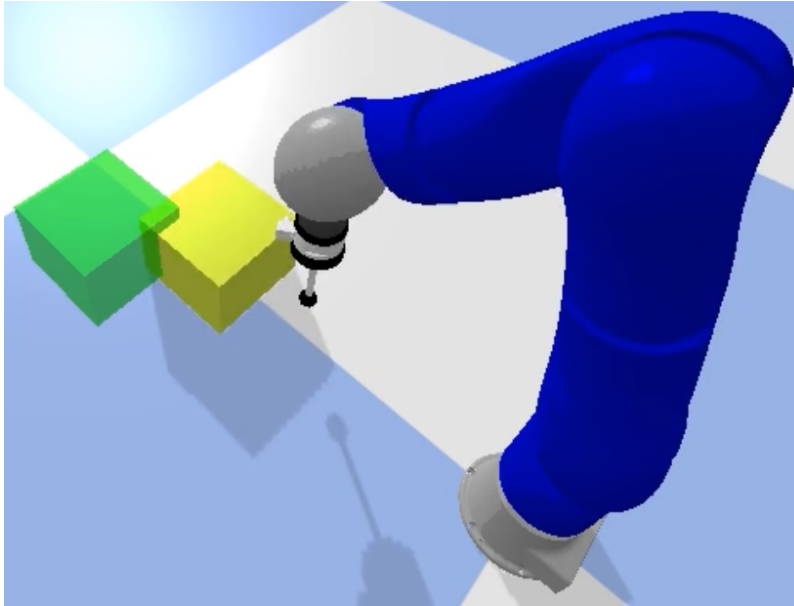
## 3.3 Adding/removing PyBullet Objects Programmatically



```
$ roslaunch rpbi_examples pybullet_objects_example.launch
```

Pybullet objects of all types can be specified in the launch file, or they can be added or removed programmatically (or even from the command line). This example demonstrates the ability of the ROS-PyBullet Interface to handle different objects. In this example, a collision object is loaded (the floor), a visual object is attached to a `tf2` frame moving in a figure-of-eight (the blue sphere), and a dynamic object is programmatically added and removed (the yellow box).

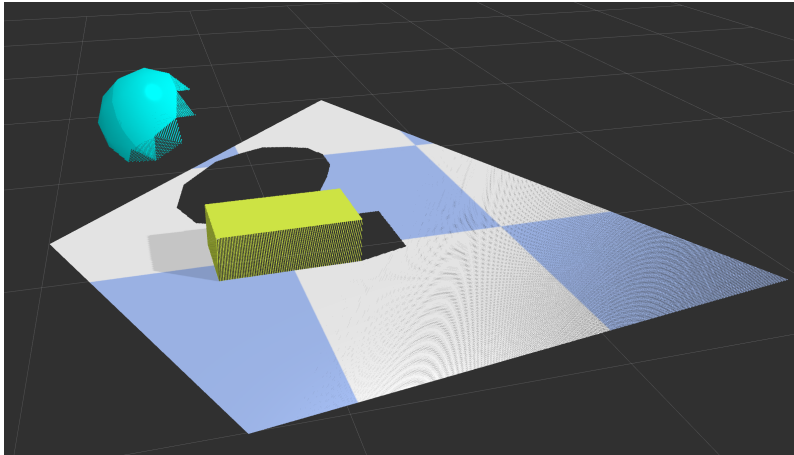
### 3.4 Learning from demonstration and teleoperation



```
$ roslaunch rpbi_examples lfd.launch
```

In this example, we demonstrate how to easily connect the ROS-PyBullet Interface with an external ROS library. The goal of the task is for the robot to push the yellow box into the green goal. When the example is launched, the robot is initialized. You can interact with the demo using the keyboard - ensure the small window (the keyboard server) is in focus. Press *key 1* to send the robot to the initial position. Press *key 2* to start and stop teleoperation - when this is activated the robot states are being recorded (used as a demonstration to learn the DMP). Press *key 3* once to learn the DMP from the demonstration, and then again to plan and execute motion using the learned DMP. *Note*, the starting position for the DMP is always random.

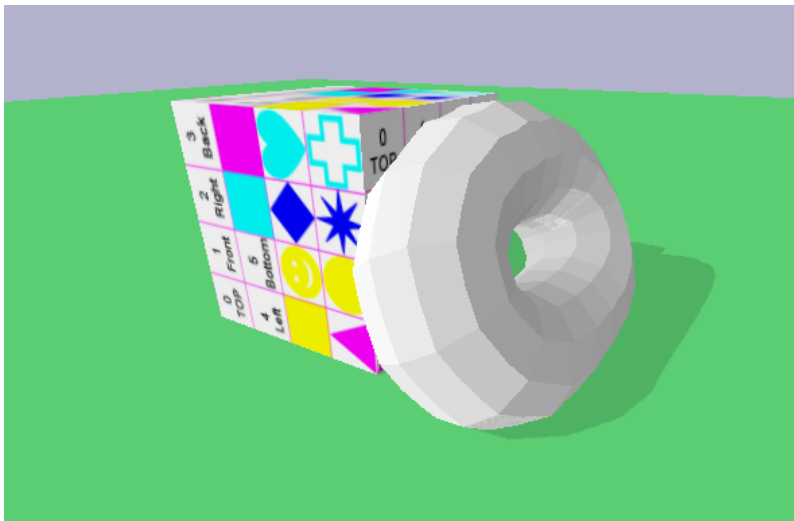
### 3.5 RGBD Sensor



```
$ roslaunch rpbi_examples soft_body.launch
```

In this example, we show how to setup an RGBD camera. This can be attached to any `tf2` frame, i.e. it could be attached to a robot link, for example. For this example, a similar scene is setup as in the `pybullet objects` example above. However, in addition, we include an RGBD camera where the camera orbits the scene. The projected point cloud is rendered in RVIZ as in the figure above.

### 3.6 Soft bodies



```
$ roslaunch rpbi_examples soft_body.launch
```

This simple example demonstrates how soft bodies can be loaded into Pybullet. In addition, this example highlights how to load objects using the `PybulletURDF` object type - this is for loading objects (not robots) from a URDF file. The torus is a soft body, and the box and floor plane are loaded from URDF. It is also possible to load soft bodies from a URDF.

## MAIN CONFIGURATION

The ROS-PyBullet Interface is launched from a ROS `.launch` file. Configuring the interface amounts to passing a `.yaml` file as follows.

```
<node pkg="ros_pybullet_interface" name="ros_pybullet_interface" type="ros_pybullet_
↪interface_node.py" output="screen">
  <rosparam param="config" file="path/to/config.yaml"/>
</node>
```

In the example above, the `config.yaml` file specifies the virtual world. This is the file where you can list what objects and robots to load, specify visualization options, and configure PyBullet physics parameters.

### 4.1 Configuration files

We use `yaml` as the format for configuration files. Many of the tags used are directly passed to PyBullet functions. For example, take the `pybullet.setGravity` method. In the ROS-PyBullet Interface node, during initialization of the PyBullet instance, if the user provides the `setGravity` tag, as follows, in the main `yaml` configuration file (full details in next section) then the `pybullet.setGravity` method is called.

```
setGravity:
  gravX: 0.0
  gravY: 0.0
  gravZ: -9.81
```

Notice in the above example that the tag and sub-tags match the interface for the PyBullet method - this is deliberate. *Under-the-hood* of the interface, the values given in the configuration file are directly passed to the corresponding method in the PyBullet library. Please note that in all cases for the library functions, most methods take an optional argument `physicsClientId` (for cases when multiple PyBullet servers are setup), this **should not** be passed - only a single PyBullet server can be created. In all cases, the default values for the PyBullet library methods are used. There are method in the PyBullet library that take a `flags` option. In this case, you can pass the strings (e.g. in the `loadURDF` method, you can specify `flags` as URDF_MERGE_FIXED_LINKS|URDF_USE_INERTIA_FROM_FILE). Note that not all methods are exposed in this way.`

The PyBullet library uses the `camelCase` style for its methods/parameters, we use this style for methods/parameters that directly correspond to a library function. There are some tags that are used to generate different behavior with regards to the interface itself. In this case, we use `snake_case` style to differentiate these parameters. The benefit for this style-guide is that it allows you to easily tell the difference between parameters that are linked to PyBullet and those that are linked to the interface. Furthermore, for the `camelCase` functions/parameters, you can look these up in the [PyBullet Quickstart Guide](#). Note, there are some exceptions to the rules however these are documented when necessary.

Our basic “hello world example” is launched using `basic_example_kuka_lwr.launch`. The main configuration file, `config.yaml`, is given as follows.

```
#
# Basic example
#

#
# Pybullet instance
#
connect:
  connection_mode: 'GUI'
  options: '--mp4=$HOME/basic_example_kuka_lwr.mp4'

setGravity:
  gravX: 0.0
  gravY: 0.0
  gravZ: -9.81

timeStep: 0.01
start_pybullet_after_initialization: true
status_hz: 50

#
# Pybullet visualizer
#
configureDebugVisualizer:
  enable: 0
  flag: 'COV_ENABLE_GUI'

resetDebugVisualizerCamera:
  cameraDistance: 2.0
  cameraYaw: 0.0
  cameraPitch: -45.0
  cameraTargetPosition: [0.0, 0.0, 0.0]

#
# Pybullet objects
#
collision_objects:
  - "{rpbi_examples}/configs/floor.yaml"
robots:
  - "{rpbi_examples}/configs/basic_example_kuka_lwr/kuka_lwr.yaml"

#
# Sensors
#

rgb_sensor:
  name: 'rgb_sensor'
  hz: 30
```

(continues on next page)



(continued from previous page)

```
# this will project the depth to a point cloud
# pointcloud: True
intrinsics:
  width: 640
  height: 480
  fov: 40
  range: [0.01, 10000]
object_tf:
  tf_id: 'rpbi/camera'
```

You will notice that there are four main sections in the main configuration file: PyBullet instance, PyBullet visualizer, PyBullet objects, and sensors. Details for each of these are given in the following sub-sections.

Note, the order in which these sections or the parameters themselves are listed does not necessarily need to be in any particular ordering. However, we suggest you follow this convention so that configuration files are more readable.

## 4.2 PyBullet Instance

This section of the main configuration file allows you to setup the main PyBullet instance. The type of settings you can set in this section relate to physical parameter (e.g. gravity), or time (e.g. simulator time-step), etc. Some parameters are expected, and others are optional. The full list of possible parameters are listed as follows.

- `connect` (required), see PyBullet documentation. Note, the `connection_mode` can be passed as a string. Also note, a very useful feature is recording videos of the interface - see the `options` parameter.
- `setAdditionalSearchPath`, see PyBullet documentation. Note, you can pass the string `"pybullet_data_path"`, this will add the additional search path given by `pybullet_data.getDataPath()`. Also note, that a list of paths can be given, these will all get added.
- `resetSimulation`, see PyBullet documentation.
- `setGravity`, see PyBullet documentation.
- `timeStep` [float], Each time PyBullet is stepped time-step will proceed with by this duration (secs). Default is `0.02`.
- `setPhysicsEngineParameter`, see PyBullet documentation.
- `step_pybullet_manually` [bool], this is always `true` when the connection mode is `DIRECT`. Otherwise, you can specify PyBullet to be stepped manually inside a ROS Timer at the rate specified by the `timeStep` parameter. Otherwise, PyBullet will step itself internally. Differences have been observed, however it is not clear exactly what is happening inside the Bullet simulator source code.
- `status_hz` [int], this is the frequency that the status publisher is broadcast to ROS.

## 4.3 Visualizer

The main GUI visualization camera can be adjusted in this section. Parameters that correspond to the visualization are listed as follows.

- `configureDebugVisualizer`, see the PyBullet documentation.
- `resetDebugVisualizerCamera`, see the PyBullet documentation. Note, the pose of the camera can be adjusted by publishing new states to the ROS topic `rpbi/reset_debug_visualizer_camera` using the message type `ros_pybullet_interface/ResetDebugVisualizerCamera`.

## 4.4 PyBullet Objects

There are several object types that are supported by the ROS-PyBullet Interface: robots, collision objects, dynamic objects, visual objects, soft bodies, and objects loaded directly from a URDF file. This section of the main configuration file allows you to specify all the objects you want in your virtual world by listing the path to the filename. You can specify these as follows.

```
robots:
- "{ros_package}/path/to/robot.yaml"

collision_objects:
- "absolute/path/to/collision_obj.yaml"

dynamic_objects:
- "{ros_package}/path/to/dynamic_obj.yaml"

visual_objects:
- "{ros_package}/path/to/visual_obj.yaml"

soft_objects:
- "{ros_package}/path/to/soft_body.yaml"

urdfs:
- "{ros_package}/path/to/urdf_obj.yaml"
```

*Note:*

- all the above tags are optional,
- multiple objects can be listed for each object type, and
- each filename can be specified with an absolute path (see `collision_objects` above), or by a relative path to a ROS package using curly brackets `{ros_package}` (as above in all other examples).

All the object types have a different required/optional settings that must be given in the specified yaml configuration files. The details for all these are given in the next section of the documentation.

## 4.5 Sensors

There are two main types of sensors that can be simulated in the ROS-PyBullet Interface: Force-Torque sensors, and RGBD cameras. The Force-Torque sensors must be connected to a robot link, see the following section of the documentation for details on how to setup this sensor. An RGBD camera can also be specified. Currently, the interface is limited to only a single camera.

If desired, the RGBD camera can be specified in the main configuration file by adding the tag `rgb_sensor` (as in the basic Kuka LWR example above). The parameters used to configure the RGBD camera are listed as follows.

- `name` (required), the name of the sensor. Each PyBullet object is given a name, all these must be unique - more details are given in the next section of the documentation.
- `intrinsics`, camera intrinsic parameters
  - `width` [int], width of camera image. Default is 640.
  - `height` [int], height of the camera image. Default is 480.
  - `fov` [int], field of view. Default is 40.
  - `range` [list[float]], depth range. Default is [0.01, 100.0].
- `pointcloud` [bool], when true the depth camera is projected as a point cloud and published to ROS. *Note*, due to the computation required this will slow the simulation. Standard ROS packages can efficiently compute this outside the simulator (as in the examples). It is recommended that you **do not** use this option. We originally added it for experimentation. Default is false.
- `hz` [int], frequency that the RGBD sensor is updated. Default is 30.
- `object_tf` (required), the pose of the camera must be attached to a `tf2` (transform) frame
  - `tf_id` [str] (required), the `tf2` frame ID that defines the camera pose. This frame **must** be defined with respect to the `rpbi/world` frame.
  - `hz` [int], the frequency that the pose is queried. Default is 30.



---

## CREATE VIRUTAL WORLDS

A virtual world can be created by listing the objects in the main configuration file (see the previous section). In addition, PyBullet objects can be added programatically (see the next section of the documentation) or even from the command line using `$ rosservice call`. We give full details for the the available PyBullet object types in the following sub-sections.

For *every* PyBullet object type, the configuration file **must** contain a `name` tag [str]. The name given to the object **must** be unique with respect to all other PyBullet objects (even of different types).

Futuremore, all objects are defined with respect to a global base coordinate frame called `rpbi/world`.

### 5.1 Robot

A robot can be specified using a URDF file. Currently, this is the only format accepted by the ROS-PyBullet Interface. The parameters for specifying a robot are listed as follows.

- `loadURDF` (required), see the PyBullet documentation. Note the `fileName` can be given relative to a ROS package using curly brackets `{ros_package}`. In addition, if the `fileName` is given as `robot_description` then the URDF file is retrieved from the `robot_description` ROS parameter - of course, this must be set in the launch file or a script somewhere.
- `setJointMotorControlArray` (required), see the PyBullet documentation. Only the `controlMode` is required to be specified. **Do not** specify `jointIndices`, `targetPositions`, `targetVelocities`, `forces`, or the `physicsClientId`.
- `initial_joint_positions` [dict[str: float]], the initial joint positions for the robot. You can specify any joints you wish by giving the joint name and initial joint position value. Unspecified joints default to `0.0`.
- `initial_revolute_joint_positions_are_deg`, when `true` the initial joint positions for revolute joints are assumed to be given in degrees, otherwise radians. Default is `True`.
- `joint_state_publisher_hz` [int], frequency that the robot joint states are published to ROS on the topic `rpbi/NAME/joint_states` with type `sensor_msgs/JointState` where `NAME` is the PyBullet object name.
- `broadcast_link_states` [bool], when `true` the robot links for the robot are broadcast to ROS as `tf2` frames.
- `broadcast_link_states_hz` [int], the frequency that the links of the robot are broadcast to ROS.
- `enabled_joint_force_torque_sensors` [list[str]], list of joint names that have Force-Torque sensors enabled. Names of joints should correspond to those defined in the URDF. When these sensors are enabled, they are published to ROS with topic name `rpbi/NAME/JOINTNAME/ft_sensor` with type `geometry_msgs/WrenchStamped` where `NAME` is the PyBullet object name, and `JOINTNAME` is the given joint name.
- `is_visual_robot` [bool], when `true` the robot is treated a visual object, i.e. it will not react to other objects in the environment and other objects will not react to the robot. This can be useful for debugging and also visualizing a real robot. The default value is `false`.

- `do_log_joint_limit_violations` [bool], when the robot is a visual robot if `true` then the joint limit violations are reported to the terminal.
- `log_joint_limit_violations_hz` [int], the frequency that the joint limit violations are checked.
- `start_ik_callback` [bool], when true a subscriber is started for the topic `rpbi/NAME/ik` of message type `ros_pybullet_interface/CalculateInverseKinematicsProblem` where `NAME` is the name of the Pybullet object. This allows you to implement task space controller. Rather than streaming target joint states to the robot, you can stream goal states (defined as a `ros_pybullet_interface/CalculateInverseKinematicsProblem` message). This option can only be used when the robot is in the `POSITION_CONTROL` or `VELOCITY_CONTROL` control modes.
- `color_alpha` [float], the alpha value for the robot in range `[0.0, 1.0]`. This allows you to make the robot transparent. By default, this option is not used.

You can move the robot in several ways. The most common way is to stream target joint states by publishing to the topic `rpbi/NAME/joint_states/target` where `NAME` is the name of the Pybullet object. Note, that joint state messages must include the name parameter, i.e. a list of joint names that specify the order of the `position/velocity/effort` attributes. Another way to generate motion is to stream task space targets, see the tag `start_ik_callback` above. Finally, several services are provided that will move the robot to desired states - see below.

Several ROS services are started when a PyBullet robot is instantiated. These are listed as follows. *Note*, in the following `NAME` is the name of the Pybullet object.

- `rpbi/NAME/robot_info` [`ros_pybullet_interface/RobotInfo`], returns information about the robot (i.e. the name, link/joint names, body unique ID, number of joints, number of degrees of freedom, joint information from PyBullet `pybullet.getJointInfo` method, see the [documentation](#), enabled Force-Torque sensors, and the current joint state).
- `rpbi/NAME/ik` [`ros_pybullet_interface/CalculateInverseKinematics`], compute a single IK. The target joint state is returned.

The following ROS services are only created for robots that are *not visual* (i.e. the tag `is_visual_robot`, see above, is omitted or set to `false`).

- `rpbi/NAME/move_to_joint_state` [`ros_pybullet_interface/ResetJointState`], given a target joint state and duration the robot is moved from the current state to the goal. The duration (in seconds) is the time it will take for the robot to move from the current state to the goal state. *Note*, there is no collision avoidance.
- `rpbi/NAME/move_to_init_joint_state` [`ros_pybullet_interface/ResetJointState`], moves the robot to the initial joint state specified in the yaml configuration file under the tag `initial_joint_positions` (see above). *Note*, we re-use the `ros_pybullet_interface/ResetJointState` service type here. That means when you call the service you will need to include the duration (i.e. time it takes for the robot to move from the current configuration to the goal) and an empty `sensor_msgs/JointState` message - the joint state message will be ignored. *Note*, there is no collision avoidance.
- `rpbi/NAME/move_to_eff_state` [`ros_pybullet_interface/ResetEffState`], given a task space target and a duration the robot is moved from the current configuration to a goal configuration (computed using PyBullet's Inverse Kinematics feature). *Note*, there is no collision avoidance.

## 5.2 Collision Object

For the collision object, other objects will react to it, but it will remain unaffected. Objects such as walls, doors, ceilings, floors should be modelled using this object type. The parameters to setup this object are listed as follows.

- `createVisualShape` (required), see PyBullet documentation.
- `createCollisionShape` (required), see PyBullet documentation.
- `changeDynamics`, see PyBullet documentation.
- `object_tf`, if unspecified the default is the identity.
  - `tf_id` [str], the tf2 frame ID that defines the camera pose. This frame **must** be defined with respect to the `rpbi/world` frame.
  - `hz` [int], the frequency that the pose is queried. Default is 30.

## 5.3 Dynamic Object

You can simulate virtual objects using a dynamic object. In this case, the objects motion is completely defined by Pybullet. The parameters for this object type are as follows.

- `createVisualShape` (required), see PyBullet documentation.
- `createCollisionShape` (required), see PyBullet documentation.
- `changeDynamics` (required), see PyBullet documentation.
- `baseMass` [float] (required), mass of the base.
- `basePosition` [list[float]], base position in the `rpbi/world` frame.
- `baseOrientation` [list[float]], base orientation in the `rpbi/world` frame (as a quaternion).
- `resetBaseVelocity`, see PyBullet documentation. Note, the `bodyUniqueId` does not need to be passed. This will specify the initial velocity of the object.
- `broadcast_hz` [int], this is the frequency that the object pose is broadcast to tf2. Default is 0 (i.e. the pose is not broadcast). The frame is always published with respect to the `rpbi/world` frame and given the name `rpbi/NAME` where `NAME` is the name of the PyBullet object.

## 5.4 Visual Object

A visual object is used primarily for visualizing real world objects or for debugging. These simply visualize objects, other objects will not react to this object and it will not react to other objects. To specify this object the following parameters can be used.

- `createVisualShape` (required), see the PyBullet documentation. Note the `fileName` can be given relative to a ROS package using curly brackets `{ros_package}`. Also, the `shapeType` parameter can be passed as a string.
- `object_tf`, if unspecified the default is the identity.
  - `tf_id` [str], the tf2 frame ID that defines the camera pose. This frame **must** be defined with respect to the `rpbi/world` frame.
  - `hz` [int], the frequency that the pose is queried. Default is 30.

## 5.5 Soft bodies

PyBullet also implements deformable object and cloth simulation. Soft bodies can be setup using the `pybullet.loadSoftBody` method, or from a URDF file. For the URDF, see the next section. When using the `loadSoftBody` approach, you can specify the following tags.

- `loadSoftBody` (required), see PyBullet documentation.
- `createSoftBodyAnchor [list[list[float/int]]]`, pin vertices of a deformable object to the world. *Note*, the PyBullet documentation for `createSoftBodyAnchor` is limited. It is not clear what is exactly the interface. The soft body unique ID will be passed automatically, but any other parameters must be supplied. Some potential resources:
  - [https://github.com/bulletphysics/bullet3/blob/master/examples/pybullet/examples/deformable\\_anchor.py](https://github.com/bulletphysics/bullet3/blob/master/examples/pybullet/examples/deformable_anchor.py)
  - <https://github.com/bulletphysics/bullet3/discussions/4088>
  - <https://github.com/bulletphysics/bullet3/blob/7dee3436e747958e7088dfdcea0e4ae031ce619e/examples/pybullet/pybullet.c#L2280-L2326>

## 5.6 Loading from URDF

This interface allows you to load objects directly from a URDF. The only required tag is as follows.

- `loadURDF` (required), see the PyBullet documentation.

*Note*, for this object type there is no ROS communication available. Future work will include updated feature set for this object type.



## COMMUNICATION WITH ROS

In this section, we discuss generally the kind of communication that the interface has with ROS. This is mainly in terms of topics/services. For PyBullet object specific ROS communication, see previous sections.

### 6.1 Services

Several services are instantiated when the ROS-PyBullet Interface node is launched. These allow you to either programmatically interact with PyBullet from your own code, or from the command line. These services are detailed as follows.

#### 6.1.1 `rpbi/start (std_srv/Trigger)`

The simulation is started, if it has been stopped.

#### 6.1.2 `rpbi/step (std_srv/Trigger)`

If the simulation is stopped, then it is stepped by one time-step. The duration of the time-step is given by the `timeStep` parameter in the main configuration file.

#### 6.1.3 `rpbi/stop (std_srv/Trigger)`

The simulation is stopped, if it is running.

#### 6.1.4 `rpbi/get_debug_visualizer_camera` (ros\_pybullet\_interface/ `GetDebugVisualizerCamera`)

When this service is called, the response returns the current parameters for the main visualizer: i.e. `cameraDistance`, `cameraYaw`, `cameraPitch`, and `cameraTargetPosition`.

### 6.1.5 rpbi/add\_pybullet\_object (ros\_pybullet\_interface/AddPybulletObject)

An object is added to PyBullet. The service allows you to either load from file or pass the configuration for the object. The input for the service expects a `ros_pybullet_interface/PybulletObject` message - see [here](#).

For both cases (i.e. load from filename or configuration), an `object_type` must be given. Either `PybulletObject.VISUAL`, `PybulletObject.COLLISION`, `PybulletObject.DYNAMIC`, `PybulletObject.ROBOT`, `PybulletObject.SOFT`, or `PybulletObject.URDF`.

*Load from filename:* the `PybulletObject.filename` variable must be set. You can specify relative filenames by giving a ROS package name in the format `{ros_package}/path/to/file.yaml`.

*Load from configuration:* if you want to pass the configuration in the service then you need to send it as a string, that is ultimately a yaml file. In Python, the best way to do this is to specify the configuration in a dict (in the same way a yaml file is loaded) and convert it to a string using the `config_to_str` method provided in `custom_ros_tools.config`. See example below.

```
from custom_ros_tools.config import config_to_str
from ros_pybullet_interface.msg import PybulletObject

# make config
config = {}
# ...

# Setup request
req = PybulletObject(config=config_to_str(config))
```

**Note:** loading from a filename takes precedence - if you want to load by passing the configuration then the `filename` parameter must not be set.

### 6.1.6 rpbi/remove\_pybullet\_object (cob\_srv/SetString)

Given the PyBullet object name as the only parameter, the object is removed from PyBullet, and all ROS communication for that object is closed.

## 6.2 PyBullet Status

From when the ROS-PyBullet node is launched, the state of PyBullet is published to the topic `rpbi/status` (with type `std_msgs/Int64`). By default, this is published at 50Hz, however you specify the frequency through the `status_hz` parameter in the main configuration file. The message on the topic indicates whether the simulator is running or not. If it is running, then the value of the message is 1, otherwise it is 0.

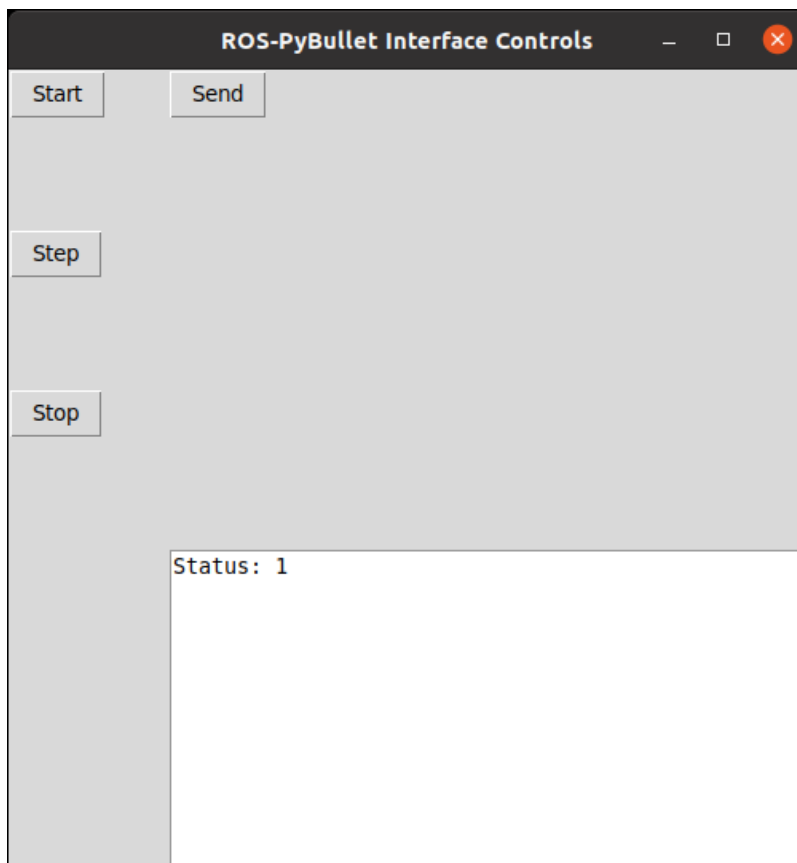
## 6.3 Time and ROS Clock

It is possible to synchronize the ROS clock time with the PyBullet simulation time. This means that when you call `rospy.Time.now()` in Python or `ros::Time::now()` in C++, the value will be equal to the simulation time in PyBullet. This can only be done when the ROS-Pybullet Interface node is configured to use manual time stepping (the reason for this is because [the Pybullet simulator time is not currently exposed](#)). To synchronize the PyBullet simulator time and ROS clock time, set the Boolean ROS parameter `use_sim_time` to `true`.

## ADDITIONAL FEATURES

### 7.1 GUI Controls

The GUI controls can be used to perform basic interactions with the interface. The following is an image of the GUI controls that you can expect to see when it is launched.



There are four buttons defined, and a text box. The effect of the buttons are as follows.

- *Start*: starts the simulation clock.
- *Step*: steps the simulation clock by the given sampling frequency specified in the [main configuration](#)
- *Stop*: stops/pauses the simulation clock.
- *Send*: sends the robot to the initial configuration as specified in its configuration file. If the initial configuration is unspecified then this defaults to zero.

The text box displays the status of the simulation. If it is 1, then the simulation is running. If it is 0 then the simulation is paused. Ultimately, what is displayed here is reporting the current value of the `rpbi/status` topic with type `std_msgs/Int64`.

### 7.1.1 Launch

Add the following to your launch file.

```
<node pkg="rpbi_utils" name="controls" type="rpbi_controls_node.py" output="screen">
  <rosparam param="config" file="path/to/config.yaml"/>
</node>
```

### 7.1.2 Configuration

The configuration for the GUI controls can be added to the *main yaml configuration*. An example is shown below.

```
controls:
  robot_name: "ROBOT_NAME"
```

The `robot_name` tag should be the name of the robot as specified in the pybullet object configuration. If the configuration is not given, then the *Send* button will have no effect.

### 7.1.3 Limitations and future development

Currently, only a single robot is supported per node. If you want to control multiple robots then you will need to launch multiple control nodes.

In the future we aim to re-implement the controls as an RQT plugin. If you would like to contribute this please [submit a pull request](#).

## 7.2 Interpolation

The interpolation interface provides the functions to generate smooth trajectories based on a sequence of waypoints. Input is a 2D array of waypoints via a topic (default value is: `ros_pybullet_interface/waypt_traj`). Output is streamed as TF in `/tf`. The waypoints (input) are provided as a 2D `[Float64MultiArray]` via a topic. The first row of the 2D array holds the relative time of the waypoints and the rest of the rows are the dimensions of the waypoints. The columns of the 2D array are the number of waypoints. The interpolation is typically used in the task space (can be modified to be used in the configuration space ) and provides linear, cubic interpolation for 1D, for euler angles (as 3 independent axis), for axis angle (1D around a specified axis), and also for quaternions.

The process steps done in the node are:

1. Input process
  - a. Read input (2D array) from topic `ros_pybullet_interface/waypt_traj` (it is suggested that this topic name is remaped)
  - b. Input is a sequence of waypoints.
2. Interpolation process
  - a. Interpolate each independent dimension separately and if there are coupled dimensions (e.g. quaternions) interpolate them jointly.

- b. Sample these interpolated functions with a frequency specified by `[inter_dt]` (see details on the parameter below)
  - c. Store these samples in a list
3. Output process
  - a. At a given frequency, specified by `[consuming_freq]`, extract (and delete) the first from the list.
  - b. Publish the extracted sample as TF to /tf with `[header_frame_id]` and `[msg_child_frame_id]` as specified below.

The interpolation node requires the following parameters to be set in the .launch file.

- `[traj_config]` `[str]` (required), specifies a yaml file which holds a number of parameters used for the interpolation.
- `[consuming_freq]` `[float]` (required), specifies the frequency of publishing the output (TF). In other words, how often a TF sample is published from the interpolated data.

input related parameters

- `[motion_dimensions]` `[number]` `[int]` (required), specifies the number of dimensions of the waypoints.

input related parameters (linear task space dimensions)

- `[motion_dimensions]` `[trans]` `[translation_x]` `[float]` (optional), specifies a fixed value of the x dimension of the task motion.
- `[motion_dimensions]` `[trans]` `[translation_x_index]` `[int]` (optional), used if `translation_x` is empty and the index of the row where the x variable is held in the 2D array with the waypoints.
- `[motion_dimensions]` `[trans]` `[translation_y]` `[float]` (optional), specifies a fixed value of the y dimension of the task motion.
- `[motion_dimensions]` `[trans]` `[translation_y_index]` `[int]` (optional), used if `translation_y` is empty and the index of the row where the y variable is held in the 2D array with the waypoints.
- `[motion_dimensions]` `[trans]` `[translation_z]` `[float]` (optional), specifies a fixed value of the z dimension of the task motion.
- `[motion_dimensions]` `[trans]` `[translation_z_index]` `[int]` (optional), used if `translation_z` is empty and the index of the row where the z variable is held in the 2D array with the waypoints.

input related parameters (angular task space dimensions)

- `[motion_dimensions]` `[rotation]` `[rotation_repr]` `[str]` (optional), options are: none, theta, quat, euler and it specifies the type of representation of the angular motion.
- `[motion_dimensions]` `[rotation]` `[rotation_vec_index]` `[list[int]]` with 1 (for theta) or 2 (for euler and quat) elements (required if theta or euler or quat), indicates the indexes of the row where the angular variables are held in the 2D array with the waypoints.
- `[motion_dimensions]` `[rotation]` `[rotation_vec]` `[list[int]]` with 3 elements (required if theta or none), specifies a fixed axis of rotation. Needs to be a normalized vector.
- `[motion_dimensions]` `[rotation]` `[rotation_angle]` `[float]` (required if none), a fixed value of the angle along the fixed axis of rotation of the task motion.

interpolation related parameters

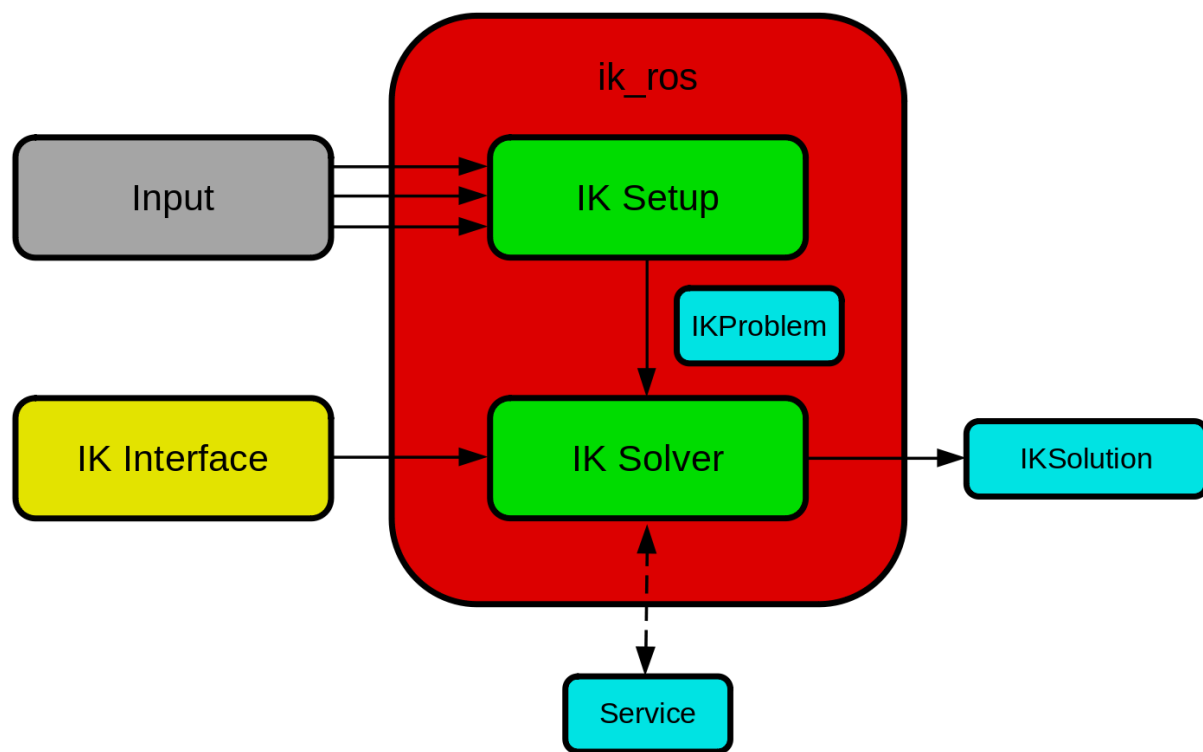
- `[interpolation]` `[nochange_window_length]` `[int]` (required), default value is 1. Advanced: specifies a window of samples that cannot be changed when new waypoints are received. It is used to ensure smoothness if the waypoints are changed on the fly.

- `[interpolation][use_interpolation] [bool]` (required), specifies whether the waypoints should be interpolated or not.
- `[interpolation][inter_dt] [float]` (required), specifies the dt between the interpolated points. In other words, frequency of the interpolation samples.

output related parameters

- `[communication][publisher][header_frame_id] [str]` (required), specifies `header_frame_id` of the TF streamed in /tf.
- `[communication][publisher][msg_child_frame_id] [str]` (required), specifies `msg_child_frame_id` of the TF streamed in /tf.

### 7.3 ik\_ros



The `ik_ros` package is a standardized interface for inverse kinematics using ROS. Input data (e.g. end-effector task space goals) are directed to a problem setup node, that collects the information into a single message. The setup node then publishes a problem message at a given frequency. A solver node, that interfaces via a standardized plugin to an IK solver, then solves the problem and publishes the target joint state.

## 7.4 safe\_robot

A low-level ROS package for the safe operation of robots. Easily setup with a single launch file. The `safe_robot_node.py` acts as a remapper. Target joint states are passed through several safety checks, if safe then the command is sent to the robot, otherwise they are prevented. Possible checks

- joint position limits
- joint velocity limits
- end-effector/link box limits
- self-collision check

## 7.5 custom\_ros\_tools

The `custom_ros_tools` package provides a collection of generic useful tools for ROS. The package is extensively used in the ROS-PyBullet Interface.





## CITE

If you use the ROS-PyBullet Interface in your work, please consider citing us using the following.

```
@misc{https://doi.org/10.48550/arxiv.2210.06887,  
doi = {10.48550/ARXIV.2210.06887},  
url = {https://arxiv.org/abs/2210.06887},  
author = {Mower, Christopher E. and Stouraitis, Theodoros and Moura, João and Rauch, ↵  
↵Christian and Yan, Lei and Behabadi, Nazanin Zamani and Gienger, Michael and ↵  
↵Vercauteren, Tom and Bergeles, Christos and Vijayakumar, Sethu},  
keywords = {Robotics (cs.RO), Machine Learning (cs.LG), FOS: Computer and information ↵  
↵sciences, FOS: Computer and information sciences},  
title = {ROS-PyBullet Interface: A Framework for Reliable Contact Simulation and Human- ↵  
↵Robot Interaction},  
publisher = {arXiv},  
year = {2022},  
copyright = {Creative Commons Attribution 4.0 International}  
}
```



## DEVELOPMENT

### 9.1 Contributing

We are more than happy to accept bug fixes, new features, suggestions, comments, or any other form of feedback. If you have an issue using the interface, or would like a new feature added please [submit an issue](#). For pull requests, please [fork the repository](#), create a new branch, and submit your pull request.

### 9.2 Future work

- Port GUI controls to RQT.
- Add additional features to GUI interface, e.g. move to custom joint states.
- Support loading from SDF and MuJoCo configuration files.
- Update ROS communication features for loading objects from URDF configuration files.

### 9.3 Known issues

- Objects with alpha color values less than 1.0 are rendered in RGB images but not the depth image for the RGBD sensor simulation. To make sure the depth image contains the object, ensure the alpha value is set to 1.0.



## ACKNOWLEDGEMENTS

This research is supported by [The Alan Turing Institute](#), United Kingdom and has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101017008, [Enhancing Healthcare with Assistive Robotic Mobile Manipulation \(HARMONY\)](#). This work was supported by core funding from the Wellcome/EPSRC [WT203148/Z/16/Z; NS/A000049/1]. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101016985, [Functionally Accurate RObotic Surgery \(FAROS project\)](#). This research is supported by [Kawada Robotics Corporation](#) and the Honda Research Institute Europe.



THE UNIVERSITY  
*of* EDINBURGH

KING'S  
*College*  
LONDON

# The Alan Turing Institute



## Harmony

Assistive robots for healthcare



## FAROS

Functionally Accurate  
Robotic Surgery

The main contributors to the development of the ROS-PyBullet Interface are as follows.

- [Christopher E. Mower](#), King's College London, UK.
- [Theodoros Stouraitis](#), Honda Research Institute, Offenbach, Germany.
- [Lei Yan](#), Harbin Institute of Technology, Shenzhen, China.
- [João Moura](#), University of Edinburgh, Edinburgh, UK.
- [Christian Rauch](#), University of Edinburgh, Edinburgh, UK.
- [Nazanin Zamani Behabadi](#), London, UK.
- [Michael Gienger](#), Honda Research Institute, Offenbach, Germany.
- [Tom Vercauteren](#), King's College London, UK.
- [Christos Bergeles](#), King's College London, UK.
- [Sethu Vijayakumar](#), University of Edinburgh, Edinburgh, and The Alan Turing Institute, UK.