Robotic Manipulation Simulator Documentation

This document contains documentation for the following components of the simulator:

Documentation for Provided Gym Environments

Module `base_env`

Classes

Class `BaseGymEnv`

```
class BaseGymEnv(config_file)
```

Base gym environment. A subclass must implement the following abstract functions according to the desired functionality: check_termination(..) get_reward(..) get_state(..) set_space_attributes(..) step(..) The following attributes must also be set in a subclass (the idea is for them to be set in the abstract function set_space_attributes(..): observation_space action_space reward_range Any instances of a robotic arm, etc. should be put into the array robots as defined here.

Initializes a base gym environment (though the gym interface is not fully implemented in this class - see subclases) by parsing config file, setting necessary parameters, initializing a Bullet client, and getting an instance of a scene class. Note: nothing is loaded into the Bullet client until reset is called.

Parameters

`config_file : str` Path to yaml file consisting of simulation configuration.

Ancestors (in MRO)

- gym.core.Env

Class variables

Variable `metadata`

Variable `scene_modes`

Methods

Method `check_termination`

```
def check_termination(self)
```

Checks whether or not the simulation has reached a terminating state (method by which this is done depends on the implementation in a given environment and the config file).

Returns

`tuple` A boolean value and a dictionary. The boolean value represents whether or not the simulation is in a terminating state and the dictionary contains auxillary diagnostic info (useful for debugging and sometimes training).

Method `close`

```
def close(self)
```

Performs any necessary cleanup after the simulation has completed.

Method `configure`

```
def configure(self, config_file)
```

Opens and reads the config file and returns the data in dictionary form.

Parameters

`config_file : str` Path to yaml file consisting of simulation configuration.

Returns

`dict` The config file data as a dictionary.

Method `configure_renderer`

```
def configure_renderer(self, disable_rendering)
```

Configures the OpenGL renderer if in GUI mode.

Parameters

`disable_rendering : bool` Whether to disable certain rendering configurations (if in gui mode rendering should be turned off while objects are being loaded and then turned back on after the objects have been loaded). If true rendering, etc. is turned off, otherwise rendering is turned on.

Method `get_reward`

```
def get_reward(self, info)
```

Calculates the reward based on the current state of the agent and the environment.

Parameters

`info : dict`

Returns

`float` Value of the reward.

Method `get_state`

```
def get_state(self)
```

Gets the current state of the robots(s) in the environment.

Returns

`object` Observation of the current state of the robot(s) in the environment

Method `initialize_client`

```
def initialize_client(self)
```

Gets a Bullet client instance via GUI or headless connection and stores it as an instance attribute.

Returns

`tuple` The first element of the tuple is an instance of pybullet_utils.bullet_client.BulletClient, which is a Pybullet client. The second element of the tuple is either the unique id of the EGL plugin if sim_mode='gpu-headless', otherwise it is None.

Method `load_egl_plugin`

```
def load_egl_plugin(self)
```

Loads the EGL plugin which allows the simulator to use the OpenGL renderer without an X11 context.

Returns

`tuple` The first element of the tuple is an instance of pybullet_utils.bullet_client.BulletClient, which is a Pybullet client. The second element of the tuple is the unique id of the EGL plugin.

Method `load_scene`

```
def load_scene(self, mode)
```

Loads a scene in the franka gym env and stores it as an instance attribute.

Parameters

`mode : str` Name of the scene to load.

Method `recompute_view_and_projection_matrix`

```
def recompute_view_and_projection_matrix(self, eye_position, target_position,
fov, near_plane, far_plane, up_vector=None, render_width=None, render_height=None)
```

Recomputes the view matrix and the projection matrix that the pybullet API uses to render images. Will change the configuration of the camera in the simulation and in turn the rendered images.

Parameters

`eye_position : list` The eye position of the camera in Cartesian world coordinates, list is of length 3.
`target_position : list` The focus point of the camera in Cartesian world coordinates, list is of length 3.
`up_vector : list` The up vector of the camera in Cartesian world coordinates (i.e. [0,0,1] means z-axis is up), list is of length 3.
`fov : float` Field of view.
`near_plane : float` Near plane distance.
`far_plane : float` Far plane distance.
`render_width : int` The horizontal resolution of the rendered image (in pixels).
`render_height : int` The vertical resolution of the rendered image (in pixels).

Method `render`

```
def render(self, mode='human', get_depth_data=False)
```

Renders the current state of the environment.

Parameters

`mode : str` The desired render mode (i.e. human or rgb).
`get_depth_data : bool` Whether to return the camera depth data as well. False by default.

Returns

`numpy.ndarray` The rgb render data (shape is height_resolution x width_resolution x 3) if mode='rgb_array' or None if mode='human'.

Method `reset`

```
def reset(self, keep_object_type=False)
```

Removes all objects from the simulation and resets class-relevant attributes. Then respawns the scene. Also resets the camera view and projection matrices to the according to the values specified in the config

file. Note: rendering is disabled during the loading process if gui mode is enabled (see BaseScene.reset()) in order to speed up the loading process.

Parameters

`keep_object_type : bool` Whether or not we want to keep the same type of object in the scene.

Method `seed`

```
def seed(self, seed=None)
```

Sets the seed for the environments rng.

Parameters

`seed : int` Desired seed for the environments rng, must be positive.

Returns

`list` The seed of the environments rng as the sole element of a list (length 1).

Method `set_space_attributes`

```
def set_space_attributes(self)
```

Defines the action space, observation space, and reward range in terms of the types provided by python package gym.spaces (i.e. spaces.Dict(), spaces.Box()) and sets them as instance attributes.

Method `step`

```
def step(self, action)
```

Must be implemented in subclass. Runs one step of the simulation.

Parameters

`action : object` The action to apply to the model

Returns

`tuple` Observation (object), reward (float), done (bool), info (dict). Where observation represents the current state of the environment, reward is the amount of reward for the previous action, done is whether or not to terminate the simulation (based on termination config), and info contains auxillary diagnostic info (useful for debugging and sometimes training).

Generated by *pdoc* 0.7.5 (https://pdoc3.github.io). pybullet build time: Apr 14 2020 20:56:53

Module `franka_base`

Classes

Class `FrankaBaseEnv`

```
class FrankaBaseEnv(config_file)
```

Franka base gym environment. A subclass must implement the following abstract functions according to the desired action and observation spaces with respect to the number of arms that will exist in the environment. set_space_attributes(..) reset(..) - any subclass specific attributes that should be reset and an observation returned step(..) The following attributes must also be set in a subclass (the idea is for them to be set in the abstract function set_space_attributes(..): observation_space action_space reward_range

Initializes a base franka gym environment (though the gym interface is not fully implemented in this class, see subclases). Sets up parameters required to load the scene - reset() must be called to load the environment.

Parameters

`config_file : str` Path to yaml file consisting of simulation configuration.

Ancestors (in MRO)

- gym_env.envs.base_env.BaseGymEnv
- gym.core.Env

Methods

Method `check_arm_obj_collision`

```
def check_arm_obj_collision(self, b, obj)
```

Checks if an arm is grabbing the object of interest (obj) and if so, depending on whether ensure-grab-is-stable either checks if the grab is stable by shaking the arm and then ensuring that the arm is still grabbing the object of interest OR by just deeming it a successful grab. The grasped object is removed after a successful grab.

Parameters

`b : gym_env.components.bin.Bin` The bin in which the object of interest is/was in.
`obj : int` Unique object id of object to check if an arm is in contact with.

Returns

`bool` A boolean value representing whether or not a successful grab has taken place.

Method `check_collision_threshold`

```
def check_collision_threshold(self, contact_points)
```

For each of the given contact points, checks if the contact distance is less that the collision threshold. This ensures that the simulator does not say an arm has collided with a bin when there is still separation between the two.

Parameters

`contact_points : tuple` A list of contact points (as returned by pybullet.getContactPoints).

Returns

`bool` A boolean value representing whether or not the contact distance of a contact point is less than the collision threshold.

Method `check_for_collision_with_bin`

```
def check_for_collision_with_bin(self, arm_id, enable)
```

Given an arm id, checks if that arm has collided with any bin in the environment.

Parameters

`arm_id : int` Id of arm to check for collisions between any bin and it.
`enable : bool` Whether the terminate-on-collision-with-bin is enabled in the config file.

Returns

`bool` True if the arm has collided with a bin, False otherwise.

Method `check_for_successful_grab`

```
def check_for_successful_grab(self)
```

Checks if any arm has successfully grabbed an object by looking at all objects that are at least a bins height above the bin with the greatest z value and height, and for each of those objects, checks if they are in contact with either end effector on the Franka arm. If ensure-grab-is-stable is True in the config file the arm will move up and down a short distance and then the method will check to ensure that the object is still grabbed firmly.

Returns

`tuple` Two bool values; the first corresponds to whether or not the number of successful grabs of unique objects that have happened exceed the number of grabs before termination as specified in the config file; the second corresponds to whether or not a successful grab has taken place.

Method `check_termination`

```
def check_termination(self)
```

Checks whether or not the simulation has reached a terminating state. In this environment a terminating state is either when when an arm collides with a bin (as implemented below, this function checks if there are any contact points between any arm and any bin) or when the simulation has exceeded the specified number of successful grabs. These terminating states can be enabled/disabled and certain aspects of them can be adjusted in the config file. All termination info is kept track of in a dictionary and returned for reward calculation purposes.

Returns

`tuple` A boolean value and a dictionary. The boolean value represents whether or not the simulation is in a terminating state and the dictionary contains auxillary diagnostic info including a boolean value regarding whether any arm has collided with a bin, whether the grab limit has been exceeded, and the number of completed grabs.

Method `get_current_number_of_objects`

```
def get_current_number_of_objects(self)
```

Gets the current number of objects that exist in the scene.

Returns

`int` Number of objects currently in the scene.

Method `get_reward`

```
def get_reward(self, info)
```

Calculates the reward based on the current state of the agent and the environment.

Parameters

`info : dict`

Returns

`float` Value of the reward.

Method `get_state`

```
def get_state(self)
```

Gets the rgb and depth camera render data as a representation of the current state of the env.

Returns

`tuple` Two numpy arrays that correspond to the rgb and depth camera data respectively. RGB array has shape (height x width x 3) where the 3 corresponds to the number of colors. Depth array has shape (height x width). In general terms it is an observation of the current state of the franka arm in the environment (same format as described by this envs observation_space) - shape is given by

Method `mount_camera_on_robot`

```
def mount_camera_on_robot(self, robot_id, mount_joint_index, grasp_target_index,
mount_position_offset=0.02, target_position_offset=0.02, fov=40, near_plane=0.001,
far_plane=1)
```

Mounts the camera to the specified robot and the specified joint on the robot.

Parameters

`robot_id : int` Unique id of robot to mount camera on.
`mount_joint_index : int` Index of joint to mount camera on.
`grasp_target_index : int` Index of joint that corresponds to the robots grasp target.
`mount_position_offset : float` The z-axis offset that should be added to the mount joint's position to get the camera's eye position.
`target_position_offset : float` The z-axis offset that should be added to the grasp target's position to get the camera's target position.
`fov : float` Field of view of camera.
`near_plane : float` Distance to the near plane of the camera.
`far_plane : float` Distance to the far plane of the camera.

Method `remove_objects_not_in_bins`

```
def remove_objects_not_in_bins(self)
```

Calls the scene method to remove any objects that are not in the vicinity of the volume given by length x width x (2 * height) for all bins.

Method `remove_single_object_from_bin`

```
def remove_single_object_from_bin(self, obj)
```

Given a bin and an object that belongs to the bin, removes the object.

Parameters

`obj : int` Unique object id of the object to remove.

Method `reset`

```
def reset(self, keep_object_type=False)
```

Resets the simulation and class-relevant environment attributes to their default settings. Then reloads the scene. Possible to reload the scene with the same type of objects that were in the bin(s) prior to calling reset. This method also ensures that the number of grabs before termination is always less than or equal to the initial number of objects in the simulation.

Parameters

`keep_object_type : bool` Whether or not the scene should be reloaded with the same object type in each bin or a different object type.

Method `set_default_arm_configs`

```
def set_default_arm_configs(self)
```

As specified in the configuration file, loads the specified number of arms in their specified positions and orientations. Converts specified orientation in config file (given in euler and degree form) to quaternion form.

Method `set_space_attributes`

```
def set_space_attributes(self)
```

Defines the action space, observation space, and reward range in terms of the types provided by python package gym.spaces (i.e. spaces.Dict(), spaces.Box()) and sets them as instance attributes.

Method `step`

```
def step(self, action)
```

Must be implemented in subclass. Runs one step of the simulation.

Parameters

`action : object` The action to apply to the model

Returns

`tuple` Observation (object), reward (float), done (bool), info (dict). Where observation represents the current state of the environment, reward is the amount of reward for the previous action, done is whether or not to terminate the simulation (based on termination config), and info contains auxillary diagnostic info (useful for debugging and sometimes training).

Generated by *pdoc* 0.7.5 (https://pdoc3.github.io). pybullet build time: Apr 14 2020 20:56:53

Module `franka_v0`

Classes

Class `Franka0Env`

```
class Franka0Env(config_file)
```

Franka gym environment for one arm, positional action space, and observation space consisting of the rgb representation of the camera render data.

Initializes a franka gym environment for one arm and the space attributes as specified in set_space_attributes(). Sets up parameters required to load the scene - reset() must be called to load the environment.

Parameters

`config_file : str` Path to yaml file consisting of simulation configuration.

Ancestors (in MRO)

- gym_env.envs.franka_base.FrankaBaseEnv
- gym_env.envs.base_env.BaseGymEnv
- gym.core.Env

Methods

Method `reset`

```
def reset(self, keep_object_type=False)
```

Resets the simulation and class-relevant environment attributes to their default settings. Then reloads the scene. Possible to reload the scene with the same type of objects that were in the bin prior to calling reset.

Parameters

`keep_object_type : bool` Whether or not the scene should be reloaded with the same object type in each bin or a different object type.

Returns

`object` Observation of the current state of this env in the format described by this envs observation_space.

Method `set_space_attributes`

```
def set_space_attributes(self)
```

Defines the action space, observation space, and reward range. The action space supports positional control for a franka arm (9 dimension box with respective lower and upper positional bounds for each joint or end effector). Observation space consists of the rgb and depth camera render data.

Method `step`

```
def step(self, action)
```

Runs one step of the simulation. Applies a position control action to the franka arm and gets an observation.

Parameters

`action : object` The action to apply to the model

Returns

`tuple` Observation (object), reward (float), done (bool), info (dict). Where observation represents the current state of the environment, reward is the amount of reward for the previous action, done is whether or not to terminate the simulation (based on termination config), and info contains auxillary diagnostic info (useful for debugging and sometimes training).

Generated by *pdoc* 0.7.5 (https://pdoc3.github.io). pybullet build time: Apr 14 2020 20:56:53

Module `franka_v1`

Classes

Class `Franka1Env`

```
class Franka1Env(config_file)
```

Franka gym environment for one arm, velocity action space, and observation space consisting of the rgb representation of the camera render data.

Initializes a franka gym environment for one arm and the space attributes as specified in set_space_attributes(). Sets up parameters required to load the scene - reset() must be called to load the environment.

Parameters

`config_file : str` Path to yaml file consisting of simulation configuration.

Ancestors (in MRO)

- gym_env.envs.franka_base.FrankaBaseEnv
- gym_env.envs.base_env.BaseGymEnv
- gym.core.Env

Methods

Method `reset`

```
def reset(self, keep_object_type=False)
```

Resets the simulation and class-relevant environment attributes to their default settings. Then reloads the scene. Possible to reload the scene with the same type of objects that were in the bin prior to calling reset.

Parameters

`keep_object_type : bool` Whether or not the scene should be reloaded with the same object type in each bin or a different object type.

Returns

`object` Observation of the current state of this env as described by this envs observation_space.

Method `set_space_attributes`

```
def set_space_attributes(self)
```

Defines the action space, observation space, and reward range. The action space supports velocity control for a franka arm (9 dimension box with respective lower and upper bounds for each joint or end effector). Observation space consists of the rgb and depth camera render data.

Method `step`

```
def step(self, action)
```

Runs one step of the simulation. Applies a position control action to the franka arm and gets an observation.

Parameters

`action : object` The action to apply to the model

Returns

tuple Observation (object), reward (float), done (bool), info (dict). Where observation represents the current state of the environment, reward is the amount of reward for the previous action, done is whether or not to terminate the simulation (based on termination config), and info contains auxillary diagnostic info (useful for debugging and sometimes training).

Generated by *pdoc* 0.7.5 (https://pdoc3.github.io). pybullet build time: Apr 14 2020 20:56:53

Module `franka_v2`

Classes

Class `Franka2Env`

```
class Franka2Env(config_file, da=0.25, dv=0.016, downsample_height=64,
downsample_width=64)
```

Franka gym environment for one arm. The action space consists of the desired change in position of the Franka robot's gripper, the desired change in angle of the Franka robot's gripper, and whether to open or close the gripper. The observation space consists of the current position of the Franka robot's gripper in world coordinates, an array of rgb image data (visual representation of the environment), and whether the gripper is open or closed (1 if open, 0 if closed).

Initializes a franka gym environment for one arm and the space attributes as specified in set_space_attributes(). Sets up parameters required to load the scene - reset() must be called to load the environment.

Parameters

`config_file : str` Path to yaml file consisting of simulation configuration.

Ancestors (in MRO)

- gym_env.envs.franka_base.FrankaBaseEnv
- gym_env.envs.base_env.BaseGymEnv
- gym.core.Env

Methods

Method `check_termination`

```
def check_termination(self)
```

Checks whether the current episode is in a state that satisfies a terminating condition.

Returns

`bool` True if the current episode is in a state that satisfies a terminating condition. False otherwise.

Method `get_image_observation`

```
def get_image_observation(self)
```

Gets an array consisting of a rgb image representation of the current state of the environment and then downsamples the image to the specified resolution.

Returns

`numpy.ndarray` Array consisting of the downsampled rgb image data. Shape is (downsample_height, downsample_width).

Method `get_info`

```
def get_info(self)
```

Gets auxillary info corresponding to the current state of the environment.

Returns

`dict` Consists of extra info corresponding to the current state of the environment.

Method `get_reward`

```
def get_reward(self)
```

Checks if the franka robot has completed a successful grab and returns an award accordingly.

Returns

`int` The value of the reward to give based on the current state of the environment.

Method `get_state`

```
def get_state(self)
```

Returns an observation of the environment as described by this env's observation space. Specifically the current position of the Franka robot's gripper in world coordinates, an array of rgb image data (visual representation of the environment), and whether the gripper is open or closed (1 if open, 0 if closed).

Returns

`dict` Information pertaining to the end effector position, an array of rgb image data, and the status of the gripper.

Method `reset`

```
def reset(self, keep_object_type=False)
```

Resets the simulation and class-relevant environment attributes to their default settings. Then reloads the scene. Possible to reload the scene with the same type of objects that were in the bin prior to calling reset.

Parameters

`keep_object_type : bool` Whether or not the scene should be reloaded with the same object type in each bin or a different object type.

Returns

`object` Observation of the current state of this env as described by this envs observation_space.

Method `set_space_attributes`

```
def set_space_attributes(self)
```

Defines the action space, observation space, and reward range in terms of the gym.spaces module. The action space is an array with shape (5,) and corresponds to the following commands [dx,dy,dz,da,open/close gripper] where dx, dy, and dz are the desired change in position of the franka robot's end effector, da is the desired change in top-down gripper angle delta, and the last element is a 1 or a -1 in order to open or close the gripper, respectively. The observation space is a dictionary consisting of the current position of the franka robot's gripper, an array consisting of the rgb image data (visual representation of the environment), and whether the gripper is open or closed.

Method `step`

```
def step(self, action)
```

Runs one step of the simulation. Applies a position control action to the franka arm and gets a position/velocity observation.

Parameters

`action : object` The action to apply to the model

Returns

`tuple` Observation (object), reward (float), done (bool), info (dict). Where observation represents the current state of the environment, reward is the amount of reward for the previous action, done is whether or not to terminate the simulation (based on termination config), and info contains auxillary diagnostic info (useful for debugging and sometimes training).

Generated by *pdoc* 0.7.5 (https://pdoc3.github.io). pybullet build time: Apr 14 2020 20:56:53

Documentation for Scenes

Module `base_scene`

Classes

Class `BaseScene`

```
class BaseScene(bullet_client, config)
```

Basic implementation of a scene. A subclass must load the specific scene and changed the self.loaded flag to True upon doing so.

Initializes a basic scene. The physics engine params that are configured in the config file can cause undefined behaviour if not configured properly (observe changes in gui mode after modifying any of the params to ensure that the simulation is accurate). A quick description of them are as follows: timestep is the amount of, simulated time that is progressed each stepSimulation() call, number of sub steps represent the number of steps that each simulation step is divided into (trades performance over accuracy), number of solver iterations is the max number of constraint solver iterations.

Parameters

`bullet_client : pybullet_utils.bullet_client.BulletClient` Pybullet client.
`config : dict` Data from the simulation configuration file.

Methods

Method `configure_renderer`

```
def configure_renderer(self, disable_rendering=True)
```

Configures the OpenGL renderer. Other configurations can be applied - see the pybullet quick start guide for more options. Additional GUI features (x, y, z axis and the camera images/view), shadows, and rendering are enabled by default - change the value from 0 to 1 or vice versa to disable or enable certain features. Note: this only applies in GUI mode.

Parameters

`disable_rendering : bool` Whether to disable certain rendering configurations (if in gui mode rendering should be turned off while objects are being loaded and then turned back on after the objects have been loaded). If true rendering, etc. is turned off, otherwise rendering is turned on.

Method `reset`

```
def reset(self, keep_object_type)
```

Applies default environment settings to the scene by setting physics engine params. The renderer is configured if the simulation requires rendering a GUI.

Parameters

`keep_object_type : bool` Whether or not the scene should be reloaded with the same object type in each bin or a different object type. Irrelevant for this class.

Module `bin_scene`

Classes

Class `BinScene`

```
class BinScene(bullet_client, config)
```

Bin scene implementation (child class of BaseScene).

Initializes a bin scene.

Parameters

`bullet_client : pybullet_utils.bullet_client.BulletClient` Pybullet client.
`config : dict` Data from the simulation configuration file.

Ancestors (in MRO)

- gym_env.scenes.base_scene.BaseScene

Class variables

Variable `plane_urdf`

Methods

Method `load_bins`

```
def load_bins(self)
```

Creates instances of the BaseBin and PickBin (1 of each) classes. Stores the objects as an instance attribute and for each BinPicking object, a random number (as specified in the config file) of a random type of object is loaded into the simulation. Also updates the class attribute current_number_of_objects.

Method `load_ground`

```
def load_ground(self)
```

Loads the plane.urdf as the ground into the simulation and then adjusts its dynamics and visual shape. Stores the unique id of the ground as an instance attribute.

Method `remove_objects_not_in_bins`

```
def remove_objects_not_in_bins(self)
```

Removes any objects that belong to the bin picking bin that are outside the vicinity of the volume given by length x width x (2 * height) of the bin. Also updates the class attribute current_number_of_objects.

Method `remove_single_object_from_bin`

```
def remove_single_object_from_bin(self, obj)
```

Given an object and the bin that it belongs to, removes the object from the simulation. Also updates the class attribute current_number_of_objects.

Parameters

`obj : int` Unique object id of the bin.

Method `reset`

```
def reset(self, keep_object_type)
```

Applies default environment settings to the scene and loads the ground as well as bin(s) and their objects (unless the bin to load is a drop-bin) as specified in the config file. The keep_object_type option only applies if the bins have already been created. If the bins have been created, the bins are reset and the objects are reloaded as either the same type or a new random type depending on keep_object_type. Also updates the class attribute current_number_of_objects.

Parameters

`keep_object_type : bool` Whether or not the scene should be reloaded with the same object type in each bin or a different object type.

Documentation for Components

Module `franka_arm`

Classes

Class `FrankaArm`

```
class FrankaArm(arm_base_position, arm_base_orientation, bullet_client,
default_joint_positions, get_end_effector_force_sensor_readings, clip_delta_actions_to_workspace
workspace_origin=None, workspace_dimensions=None)
```

Class providing functionality and support for franka arm control and interaction in a simulator (designed for pybullet). Note that the gripper on the robot consists of two end effectors so all sections of the code that pertain to something end effector specific are labelled as such, while sections that refer to the gripper (both end effectors) as a whole as labelled as such.

Initializes a franka arm with a fixed base at the given position and orientation as stated in the configuration file. First reads the specified initial joint poses as specified in the configuration file and sets the arm to that position, then creates a list of the indices of the move-able joints in the franka arm and a list of the max velocities of each move-able joint.

Parameters

`arm_base_position : list` The position to load the base of the arm at in the form [x, y, z], length 3.
`arm_base_orientation : list` The orientation of the base of the arm in in the form [x, y, z, w] (world space quaternion), length 4.
`bullet_client : pybullet_utils.bullet_client.BulletClient` Pybullet client.
`default_joint_positions : list` Default pose for each joint the in arm, length 12.
`get_end_effector_force_sensor_readings : bool` Whether to get the force readings from the sensors on the arms end effectors while applying an action.
`clip_delta_actions_to_workspace : bool` Whether to clip delta actions to the workspace.
`workspace_origin : list` The coordinates of the center of the workspace. If the workspace is a bin then the center of the workspace would be the origin of the bin. List has length 3.
`workspace_dimensions : list` The length x width x height of the workspace, length 3. If the workspace is a bin then the workspace dimensions would be the dimensions of the bin.

Class variables

Variable `acceptable_gripper_proximity_to_target`

Variable `acceptable_proximity_to_target`

Variable `end_effector_indices`

Variable `gripper_index`

Variable `max_gripper_force`

Variable `max_steps`

Variable `revolute_joints`

Methods

Method `apply_action`

```
def apply_action(self, control_mode, target)
```

Applies action to this Franka arm using either position or velocity control. If position control is used, will call self.at_target() which steps the physics engine until the given number of joints in this arm are within a specified proximity to the target position or until the specified max number of allowed physics engine steps are reached. Also if using position control end effectors are moved with a max force of 80. Also returns the reaction forces sensed by each end effector (but only calculates them if get-end-effector-force-sensor-readings is enabled in the config file). If velocity control is used, the physics engine is stepped once after the velocities have been applied to the joints and end effectors.

Parameters

`control_mode : int` Means of applying action to arm (self.bc.POSITION_CONTROL or self.bc.VELOCITY_CONTRO

`target : list` Target pose or velocity for each move-able joint in arm (including end effectors), length 9.

Returns

`dict` A dictionary consisting of a key, value pair where each key corresponds to one of the end effectors and the value consists of a list of lists corresponding to the reaction forces sensed by the respective end effector at each simulation step.

Method `apply_action_delta`

```
def apply_action_delta(self, delta_action)
```

Takes the desired offset specified by the delta_action parameter and adds it to the current end effector position and orientation. Inverse kinematics is then used to calculate the poses of the joints in the rest of the arm. The action is then applied to the arm and the gripper is opened or closed depending on the command specified by delta_action. Optionally returns the reaction forces sensed by each end effector (if get-end-effector-force-sensor-readings is enabled in the config file).

Parameters

`delta_action : list` [dx, dy, dz, dalpha, open/close gripper]. Specifically the change in position and orientation to apply to the gripper of the franka robot as well as whether to open or close the gripper.

Returns

`dict` (optional) A dictionary consisting of a key, value pair where each key corresponds to one of the end effectors and the value consists of a list of lists corresponding to the reaction forces sensed by the respective end effector at each simulation step.

Method `apply_action_pos`

```
def apply_action_pos(self, target_joint_poses)
```

Moves all (move-able) joints of this arm to their specified target angle/position. Checks if the desired target position of each move-able joint is within the allowed range and if so the original target positions are used, otherwise the upper or lower bound (if target position is too large or too small, respectively) is used. Optionally returns the reaction forces sensed by each end effector (if get-end-effector-force-sensor-readings is enabled in the config file).

Parameters

target_joint_poses : list Target joint pose for each moveable joint in arm (including end effectors), length 9.

Returns

dict (optional) A dictionary consisting of a key, value pair where each key corresponds to one of the end effectors and the value consists of a list of lists corresponding to the reaction forces sensed by the respective end effector at each simulation step.

Method apply_action_to_gripper

```
def apply_action_to_gripper(self, target_position_a, target_position_b)
```

Moves only the end effectors of this arm to a target position. Steps the physics engine until either the max number of steps are reached or until the position of the end effectors are within a certain proximity of their target position.

Parameters

target_position_a : float Target position of end effector a (index 9).
target_position_b : float Target position of end effector b (index 10).

Method apply_action_vel

```
def apply_action_vel(self, target_joint_velocities)
```

Moves all (move-able) joints of this arm using velocity control and the specified target velocities. Checks if the desired target velocity of each move-able joint is within the allowed range and if so the original target velocities are used, otherwise the upper or lower bound (if target velocity is too large or too small, respectively) is used.

Parameters

target_joint_velocities : list Target velocity for each moveable joint in arm (including end effectors), length 9.

Method at_target

```
def at_target(self, target, num_joints_to_check=9)
```

Checks to see whether the current position of the given number of joints in this arm are within a specified proximity to the target position. If not, steps the physics engine and then checks again. This is repeated until self.max_steps are reached. Returns the reaction forces sensed by each end effector along their z-axis (but only calculates them if get-end-effector-force-sensor-readings is enabled in the config file).

Parameters

target : list Target pose or velocity for each move-able joint in arm (including end effectors), length 9.

num_joints_to_check : int The number of joints to check if within the acceptable proximity (i.e. if 7 then the positions of the first 7 joints will be checked to ensure that they are within the acceptable proximity and the position of the last 2 (end effectors) will be ignored).

Returns

`dict` A dictionary consisting of a key, value pair where each key corresponds to one of the end effectors and the value consists of a list of the reaction forces sensed by the respective end effector at each simulation step.

Method `check_grab`

```
def check_grab(self, obj_id)
```

Checks if there are any contact points between a specified object and both end effectors on this arm.

Parameters

`obj_id : int` Id of object to check if this arm's end effectors are in contact with.

Returns

`bool` True if either end effector is in contact with the specified object, False otherwise.

Method `check_target_position_bounds`

```
def check_target_position_bounds(self, target_pos_a, target_pos_b)
```

Checks if the desired target position of the gripper is within the allowed range and if so the original target positions are returned, otherwise the upper or lower bound (if target position is too large or too small, respectively) is returned.

Parameters

`target_pos_a : float` Target position of end effector a (index 9).
`target_pos_b : float` Target position of end effector b (index 10).

Returns

`tuple` Two floats. target_pos_a is the new target position guaranteed to be within the end effectors range and target_pos_b is the new target position guaranteed to be within the end effectors range.

Method `clip_to_workspace`

```
def clip_to_workspace(self, desired_target_pos)
```

Clips the desired target position of the end effector so that it is within the scope of the workspace. For example, if the desired target position is below the workspace, that action will be shifted up so the desired target position is within the workspace.

Returns

`numpy.ndarray` Array with shape (3,) consisting of the clipped target position

Method `close_gripper`

```
def close_gripper(self)
```

Closes the gripper on the franka robot. Gets the lower limit of each end effector's allowed range (should be 0 or very close to it) and moves both end effectors to that position, which closes the gripper. Also updates the gripper's state.

Method `enable_joint_force_torque_sensor_on_end_effectors`

```
def enable_joint_force_torque_sensor_on_end_effectors(self)
```

Enables a force/torque sensor in both end effectors.

Method `get_end_effector_reaction_forces`

```
def get_end_effector_reaction_forces(self)
```

Gets the reaction forces sensed by each end effector along the z-axis of their reference frame. Currently the other sensed forces [Fx,Fy] and sensed torques [Mx,My,Mz] are ignored.

Returns

`tuple` Two floats; the first float corresponds to the reaction force sensed by end effector a (joint index 9) along its z-axis and the second float corresponds to the reaction forces sensed by end effector b (joint index 10) its z-axis.

Method `get_gripper_state`

```
def get_gripper_state(self)
```

Gets the coordinates of the current position of the gripper as well as the current orientation of the gripper.

Returns

`tuple` Two numpy arrays, the first consisting of coordinates of the current position of the gripper, shape (3,) and the second consisting of the current orientation of the gripper in euler form, shape (3,).

Method `get_joint_poses_ik`

```
def get_joint_poses_ik(self, target_ee_position, target_ee_orientation=None,
use_null_space=False)
```

Computes the joint angles that makes the head joint reach a specified target position with a specified target orientation (inverse kinematics). It is important to note that the target position corresponds to the link coordinate of the head joint (not the center of mass coordinate). Optional null-space support is available (preferred).

Parameters

`target_ee_position : list` Target position of the arm's head in the form [x, y, z], length 3.

`target_ee_orientation : list` Target orientation of the head in the form [x, y, z, w] (quaternion), length 4.

`use_null_space : bool` Determines whether the specified joint limits and rest poses - as found in null_space_limits(…) are accounted for in the inverse kinematics calculation.

Returns

`numpy.ndarray` Array consisting of the target joint poses for all move-able joints in the arm as calculated by ik, length 9

Method `get_position_velocity_state`

```
def get_position_velocity_state(self)
```

Describes the current state of this arm instance in terms of the positions and velocities of each joint.

Returns

`numpy.ndarray` Array describing the current state: observation[0-6] -> joint angle of each moveable joint in the franka arm. observation[7-8] -> position of each end effector. observation[9-15] -> joint velocity of each moveable joint in the franka arm. observation[16-17] -> velocity of each end effector.s

Method `get_workspace`

```
def get_workspace(self, workspace_origin, workspace_dimensions)
```

Creates a list consisting of elements that correspond to the lower and upper limits of each dimension of the workspace. An error margin is used to ensure that no collisions with the edge of the workspace occur. The end effector tip offset is the distance between the bottom of each end effector and the grasp target location on the end effector (which is approx. 0.75 of a cm). Further there is no upper limit on the workspace.

Parameters

`workspace_origin : list` The coordinates of the center of the workspace. If the workspace is a bin then the center of the workspace would be the origin of the bin. List has length 3.

`workspace_dimensions : list` The length x width x height of the workspace, length 3. If the workspace is a bin then the workspace dimensions would be the dimensions of the bin.

Returns

`list` A list consisting of 3 lists, each of length 2 corresponding to the lower and upper limits of each dimension (x, y, z) of the workspace.

Method `null_space_limits`

```
def null_space_limits(self)
```

Finds the null space limits (lower limits, upper limits, joint ranges, rest poses) of each move-able joint in this arm and sets the instance attributes accordingly. The rest pose is set to the default position of the arm. Note: a negative position index means the joint is fixed.

Method `open_gripper`

```
def open_gripper(self)
```

Opens the gripper on the franka robot. Opens each end effector so that they are in their default position. The recommended distance to move each end effector is 0.02 m as this will move them a distance apart such that the diameter between the end effectors is 1 cm larger than the diameter of the largest object (3 cm). Also updates the gripper's state.

Method `read_joint_positions`

```
def read_joint_positions(self, default_joint_positions)
```

Loads the joint positions as specified in the configuration file into the instance attribute default_joint_pos.

Parameters

`default_joint_positions : list` Default joint pose for each joint in arm, length 12.

Returns

`list` The default joint pose for each joint in arm as a list.

Method `reset`

```
def reset(self)
```

Resets all the joints of this arm to their default position as stored in the instance attribute default_joint_positions. Resets the gripper's state to open.

Method `shake`

```
def shake(self)
```

Shakes the arm by moving the head up a small distance and then back down a small distance to the original position. Use of this function is to validate a successful firm grab by shaking the arm and then ensuring that the object is still being held by the arm.

Module `camera`

Classes

Class `Camera`

```
class Camera(camera_config, bullet_client)
```

A class for rendering a pybullet environment. Depending on configs, different render settings will be used.

Initializes a camera object with the configuration data.

Parameters

`camera_config : dict` Camera configuration data.
`bullet_client : pybullet_utils.bullet_client.BulletClient` Pybullet client.

Class variables

Variable `render_modes`

Methods

Method `compute_view_and_projection_matrix`

```
def compute_view_and_projection_matrix(self)
```

Computes the view and projection matrix for pybullets getCameraImage() API, and stores them as instance attributes.

Method `get_image`

```
def get_image(self, sim_mode, get_depth_data=False)
```

Gets the image via GUI render engine (OpenGL) or headless render engine (TinyRenderer). Can also return the camera depth data as a greyscale image if desired (set get_depth_data to True). The true z-value is calculated from the depth pixels using the equation given in the documentation for pybullet.getCameraImage(..). The resulting array is then scaled such that it is possible for a greyscale image to be created from it.

Parameters

`sim_mode : string` The mode of the simulation (i.e. one of 'gpu-gui', 'gpu-headless', or 'cpu-headless').
`get_depth_data : bool` Whether to return the camera depth data as well. False by default.

Returns

`tuple` Either a single numpy array consisting of the rgb render data (shape is height_resolution x width_resolution x 3) or a tuple consisting of the rgb render data and the depth render data (shape of the depth render data is (height_resolution x width_resolution).

Method `render`

```
def render(self, sim_mode, mode, get_depth_data=False)
```

Renders the current pybullet environment. The modes 'human' and 'rgb_array' are supported and do the following: 'human' mode returns nothing and continues to display the simulation if in GUI mode (usually for human consumption). 'rgb_array' returns a numpy.ndarray with shape (x, y, 3), representing RGB values for an x-by-y pixel image, suitable for turning into a video. Can also return the camera depth data as a greyscale image if desired (set get_depth_data to True).

Parameters

`sim_mode : string` The mode of the simulation (i.e. one of 'gpu-gui', 'gpu-headless', or 'cpu-headless').
`mode : str` The desired render mode (i.e. human or rgb).
`get_depth_data : bool` Whether to return the camera depth data as well. False by default.

Returns

`numpy.ndarray` Array consisting of the rgb render data (shape is height_resolution x width_resolution x num_colors). Num_colors is 3 for an rgb image.

Module `base_bin`

Classes

Class `BaseBin`

```
class BaseBin(bin_config, bullet_client)
```

Class representing a drop-bin. Has support for loading a bin and creating the collision and visual shape for objects. The default dimensions of this class are 160x160x40 mm or 0.16x0.16x0.04 m.

Initializes a bin with the specified mesh scale, position, and orientation into the relevant pybullet client.

Parameters

`bin_config : dict` Consists of bin configuration data.
`bullet_client : pybullet_utils.bullet_client.BulletClient` Pybullet client.

Class variables

Variable `default_bin_file_name`

Variable `default_dimensions`

Methods

Method `create_collision_and_visual_shape`

```
def create_collision_and_visual_shape(self, file_name, mesh_scale, frame_offset,
rgba_color=None, load_bin=False)
```

Creates the collision and visual shape of a Wavefront OBJ file so that a multibody can be created and loaded into the simulation. If the relevant object is a bin, the collision shape is created using the GEOM_FORCE_CONCAVE_TRIMESH flag which essentially allows the bin to have a hollow interior (only stable if the object is fixed: mass = 0). Otherwise the collision shape will be created normally.

Parameters

`file_name : str` Relative or absolute file path to Wavefront OBJ file.
`mesh_scale : list` Desired scale of bin object, length 3 (use this to adjust size of obj).
`frame_offset : list` Translational offset of collision shape with respect to the link frame, length 3.
`rgba_color : list` Desired color of object to load, length 4.
`load_bin : bool` Whether or not the object to create a collision and visual shape for is a bin.

Returns

`tuple` Two ints which consist of the unique id for the collision shape and the unique id for the visual shape respectively.

Method `create_constraint`

```
def create_constraint(self)
```

Creates a constraint for shaking the bin.

Returns

`int` The unique id of the constraint.

Method `load_bin`

```
def load_bin(self)
```

Loads a bin into the simulation, with the specified orientation and scale, in the specified position. Makes the bins color a dark grey.

Returns

`int` Unique object id of bin.

Method `reset`

```
def reset(self, keep_object_type)
```

Reloads this bin into the relevant pybullet client.

Parameters

`keep_object_type : bool` Whether or not the scene should be reloaded with the same object type in each bin or a different object type. Irrelevant for this class.

Module `pick_bin`

Classes

Class `PickBin`

```
class PickBin(bin_config, bullet_client, ml_mode, steps_for_objects_to_fall=500)
```

Class representing a bin with objects. Has support for loading a random type and number of objects in random positions within the bin. Also has support for checking whether an object is within the vicinity of the bin and removing one object or multiple if they are not within the bin.

Initializes a bin with the specified mesh scale, position, and orientation into the relevant pybullet client. Also loads objects within the bin and ensures that the min and max number of objects that are specified in the configuration file are structured properly.

Parameters

`bin_config : dict` Consists of bin configuration data.
`bullet_client : pybullet_utils.bullet_client.BulletClient` Pybullet client.
`ml_mode : bool` Whether an agent is being trained or tested (this class selects which folder to pick a random object from based on whether we are currently training or testing an agent).
`steps_for_objects_to_fall : int` The number of times the physics engine is stepped in order to allow the objects to fall within the bin.

Ancestors (in MRO)

- gym_env.components.base_bin.BaseBin

Methods

Method `allow_objects_to_fall`

```
def allow_objects_to_fall(self)
```

After all objects have been loaded, the physics engine is stepped a specified number of times in order to allow the objects to fall within the bin.

Method `contains`

```
def contains(self, obj_position)
```

Checks if an object is within the vicinity of the volume given by length x width x (2 * height) of this bin.

Parameters

`obj_position : tuple` Position of the object in the form [x, y, z], length 3.

Returns

`bool` True if the volume given by length x width x (2 * height) of bin contains the object, False otherwise.

Method `get_list_object_coords`

```
def get_list_object_coords(self, number_of_objs)
```

Generates a list of length number_of_objs which consists of the coordinates to load each object. The objects are loaded with a random x coordinate (within the scope of the bin) and a y coordinate that changes by 5 mm more than max diameter for an object (3 cm) each iteration. When the y coordinate exits the scope of the bin it is set back to its starting position (on the -y side of the inside of the bin) and the z coordinate is incremented by 5 mm more than the max diameter (3 cm) for an object. Note: the

max dimension of all objects in the training_objs and testing_objs folder is guarenteed to be between 1 and 3 cm.

Parameters

number_of_objs : int Number of objects that are to be loaded.

Returns

list [x,y,z] coordinates of each object that is to be loaded, length number_of_objs.

Method `get_random_object_file`

```
def get_random_object_file(self, path)
```

Randomly selects an object file to load from training_objs or testing_objs depending on whether we are currently training or testing an agent.

Parameters

path : str The absolute path to the assets folder.

Returns

str The absolute path to the randomly selected object file.

Method `load_objects`

```
def load_objects(self, keep_object_type)
```

Loads a certain number of uniform objects (as specified in the config file for the respective bin) of a random type into this bin. After all objects have been loaded, the physics engine is stepped a specified number of times in order to allow the objects to fall within the bin.

Parameters

keep_object_type : bool Whether or not the scene should be reloaded with the same object type in each bin or a different object type.

Method `remove_objs_not_in_bin`

```
def remove_objs_not_in_bin(self)
```

Removes any objects that were initially loaded into this bin and are not within the vicinity of the volume given by length x width x (2 * height) of this bin.

Returns

int The number of objects removed.

Method `remove_single_object`

```
def remove_single_object(self, obj)
```

Removes a single object from the simulation.

Parameters

obj : int Unique id of the object to remove.

Method `reset`

```
def reset(self, keep_object_type)
```

All objects in the simulation are removed by BaseGymEnv calling resetSimulation() on the pybullet client. This method reloads a random number of either the same object type that existed in this bin prior to the resetSimulation() call, or a new random object type depending on keep_object_type.

Parameters

`keep_object_type : bool` Whether or not the scene should be reloaded with the same object type in each bin or a different object type.
