Lunar Autonomy Challenge documentation

Updated - 3rd September 2024

Appendices

This PDF version of the documentation is included for convenience and may not be up to date with the latest documentation online. We recommend to use the online challenge documentation found on the Lunar Autonomy Challenge website.

Introduction

Updated - 3rd September 2024

Overview

The Lunar Autonomy Challenge (LAC) tasks you with developing an autonomous agent capable of controlling a virtual lunar rover in mapping the surface of a virtual moon map. At the center of the mapping area is a lunar lander that may be used as a visual reference point to assist with navigation. Your agent will be assessed in its capability to successfully and efficiently navigate the area around the lander while producing an accurate map of surface elevation and rocks in the area within a limited mission time. You will need to manage the rover's battery consumption through careful management of its various components such as motors, cameras and lights to efficiently use the limited battery power. The rover must periodically return to a recharging point on the lander when power is low.

Lunar simulator

The LAC uses a custom version of the CARLA simulator to simulate the moon environment. This will be referred to as the *lunar simulator* throughout this documentation. The lunar simulator provides a Python API with methods to control the movement of the rover and receive sensor data from the rover's 8 cameras and inertial measurement unit (IMU).

> **Note**
>
> Please avoid referring to documentation for the standard CARLA simulator since this may cause confusion. The CARLA API has many additional objects and methods that are not necessary to use the lunar simulator. All the functionality you need for the challenge is described in this document.

Leaderboard

Missions in the lunar environment are executed by a companion software to the lunar simulator named the *Leaderboard*. The Leaderboard software initializes your agent at a random location in a moon map and runs a simulated mission, in which your agent must complete the mapping task. Upon mission completion, the Leaderboard evaluates the correspondence of your estimated terrain map with the ground truth surface and calculates

scores according to a set of predetermined mission evaluation criteria. The mission may be terminated early if, for example, the rover runs out of battery or becomes stuck, in which case your agent will be evaluated on the portion of the mapping task completed. Please read the competition metrics section for more information about scoring and off-nominal mission termination.

The lunar simulator is provided with 2 different moon surface maps for use as training and validation environments for development. The ground truth for these maps is accessible through the output of the Leaderboard software. You may compare your agent's mapping result directly with the ground truth for each mission you run locally.

Leaderboard cloud

The Leaderboard cloud provides an independent environment to test your agent on a moon map for which you are not provided the ground truth, the competition map. Once you are satisfied with the performance of your agent in locally executed missions, you may then submit a copy of your agent to the cloud system through its web interface where your agent's performance is evaluated on the competition map. Please see the sections on packaging your agent in a Docker container and submitting your agent for information about cloud submission.

Competition phases

The competition runs in two phases, a qualification round and the finals. In the qualification phase, competitors are required to map a smaller area of size **9m x 9m** within total mission time of 1 hour. A subset of teams will be chosen to enter the finals based on the scores from the qualifying round. In the final round, your agent will map an area of **27m x 27m** within total mission time of 24 hours.

Mission time and compute time

It is important to distinguish between *mission time* and *compute time*. *Mission time* (or *simulation time*) refers to the virtual mission time modelled within the simulator. *Compute time* refers to the real-world time that passes during the execution and computation of the simulation. Running the simulator on high-spec hardware may render these times to be roughly equal, but in many cases compute time will be a multiple of mission time between 1 and 5. For example, 10 minutes of virtual mission time within the simulator may require 30 real-world minutes to compute. It is important to note that compute time is limited once you

submit to the competition in order to limit cloud compute cost. In the qualification phase, the compute time is limited to 7.5 hours (for 1 hour of mission time) while in the final competition compute time is limited to 180 hours for 24 hours of mission time. Therefore it is important to ensure that your agent's internal logic for processing the sensor data and producing a control decision is as efficient as possible.

The lunar rover

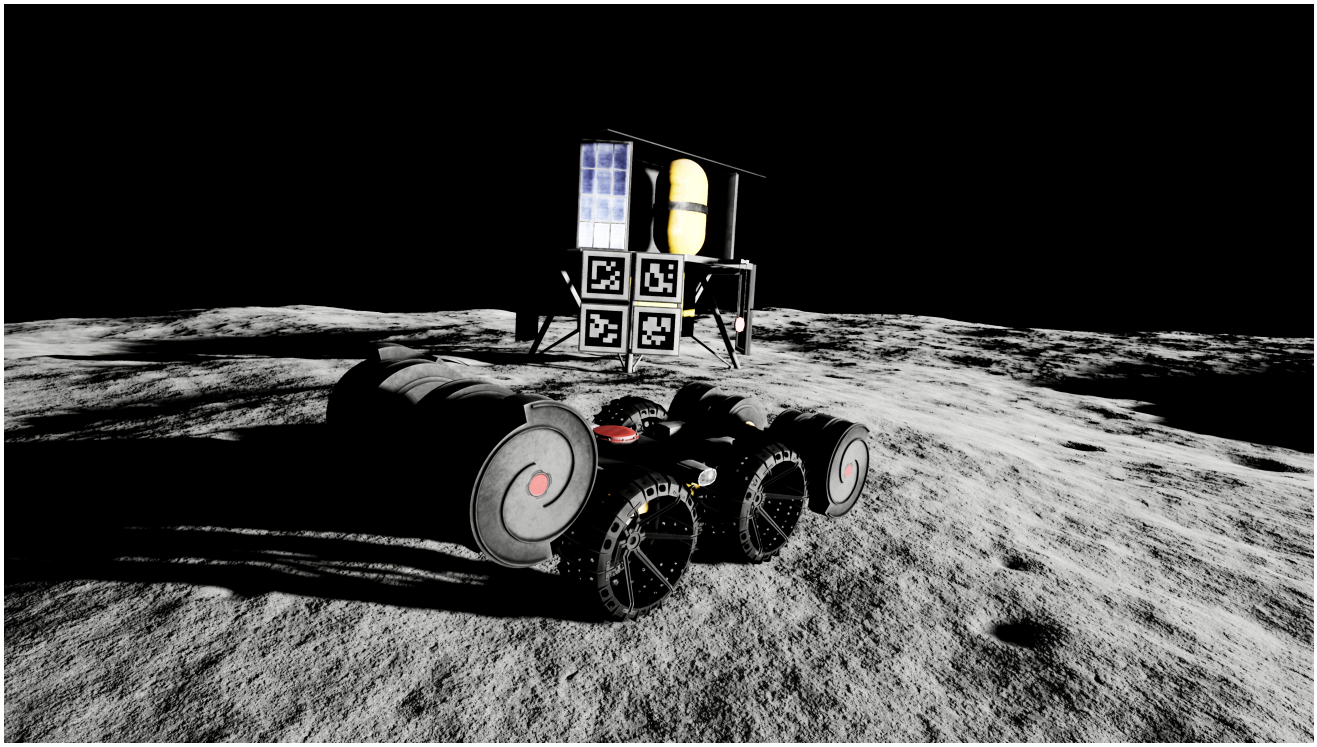


Figure 1.1: Image of the IPEx rover as it appears in the simulator in the foreground with the the lander in the background.

The lunar rover model is based on the IPEx lunar rover design. It is a 4-wheeled rover with differential steering and 8 monochrome cameras, including 2 stereo pairs. At the front and rear of the rover are two articulating arms with rotating drums at each end intended for regolith excavation. *Note that the 2025 Lunar Autonomy Challenge focuses on mapping only; excavation is not part of the challenge.*

The arms may be lowered below the chassis of the rover to lift the rover body, achieving a better vantage point for the cameras. The drums may also be rotated while the rover is raised on the drums, enabling slow movement. Additionally, the arms may be utilized for attempts at righting the rover in the case of a tipping or rolling incident.

Cameras and lights

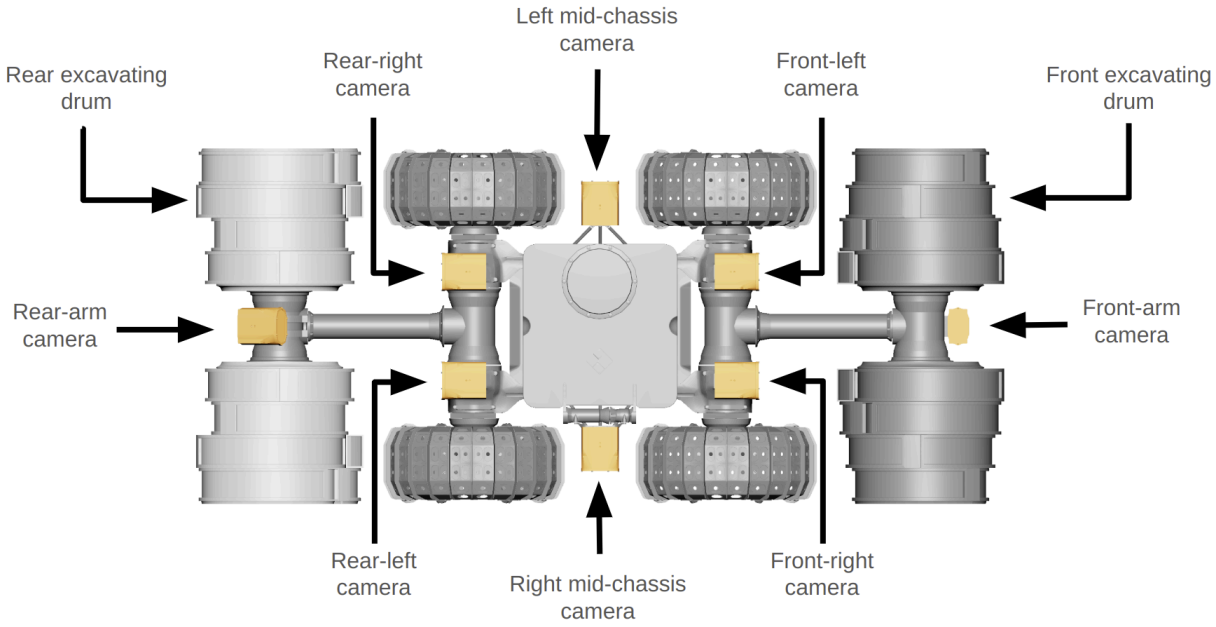


Figure 1.2: Diagram of the IPEx rover looking down from above, showing the positions of the 8 cameras.

The 8 cameras are arranged into 2 stereo pairs, one pointing forwards and one pointing backwards. Two cameras are placed on opposite sides of the middle of the chassis, pointing outwards, enabling both a left and right side facing view. The front arm has a camera on its end that points perpendicular to the axis of the arm, pointing forwards if the arm is raised to 1.57 radians (90 degrees). The rear arm has a camera angled at 0.52 radians (30 degrees) from the axis of the arm pointing forwards. This allows a view over the top of the rover chassis.

Each camera is coupled with an LED light to illuminate the moon surface. Each camera and its respective light may be activated or deactivated at any point during the mission. Active cameras and lights drain power from the battery as long as they are active. Therefore, to preserve battery, you should aim only to use the cameras that are strictly needed for mapping and localization, deactivating unused cameras. You may also choose the resolution of the cameras. Camera resolution affects simulation performance, with higher resolution requiring more compute time. Therefore, the chosen camera resolution should be balanced between the needs of the perception/localization system and performance. There is a maximum limit of 2448x2048 pixels . Please see the geometry section for the exact locations of the cameras on the rover chassis.

The simulator provides complimentary ground truth semantic segmentation data alongside the monochromatic camera data to assist with the development of perception algorithms. The semantic data differentiates 6 distinct classes:

- Moon terrain/regolith - RGB = (81, 0, 81)
- Rocks - RGB = (108, 59, 42)
- Rover - RGB = (0, 0, 142)
- Lander - RGB = (110, 190, 160)
- Fiducials - RGB = (250, 170, 30)
- Earth - RGB = (70, 130, 180)

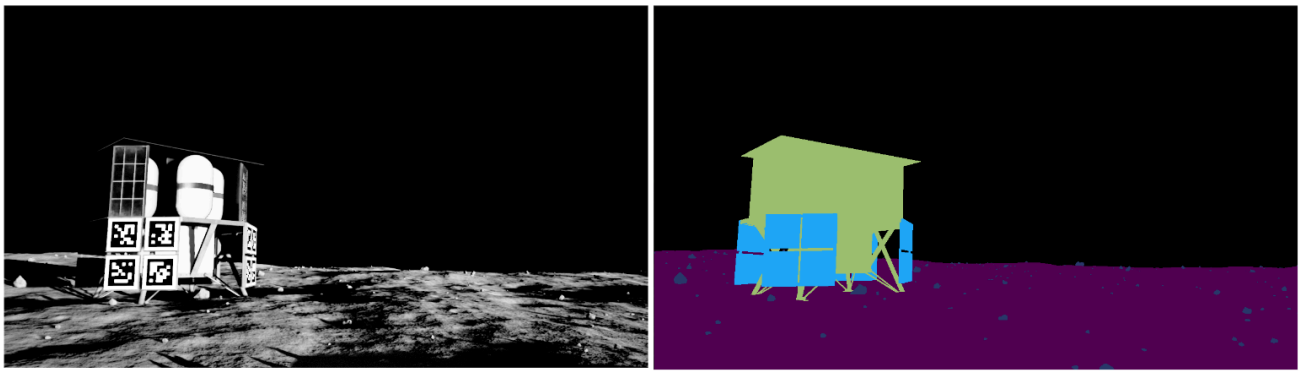


Figure 1.3: IPEx camera view from a forward facing camera. Grayscale or monochrome (left) and semantic (right).

Please note that the semantic cameras are not available in the competition, they may only be used for training and development purposes.

The arms of the rover may be moved to aim the arm cameras or unblock the field of view of the stereo camera pairs.

The IPEx rover has a limited bandwidth on the bus carrying the sensor data, meaning that a maximum of 4 cameras will be simultaneously available. No restrictions are placed on the number of cameras simultaneously available in the simulator. However, solutions that rely on no more than 4 cameras activated simultaneously will be more efficient and more applicable to the real-world use case. We therefore encourage diligent management of the camera activation states and resolution settings. Each camera active in the simulator also adds a computational burden. Therefore the simulator will be more efficient with less active cameras.

The lunar lander

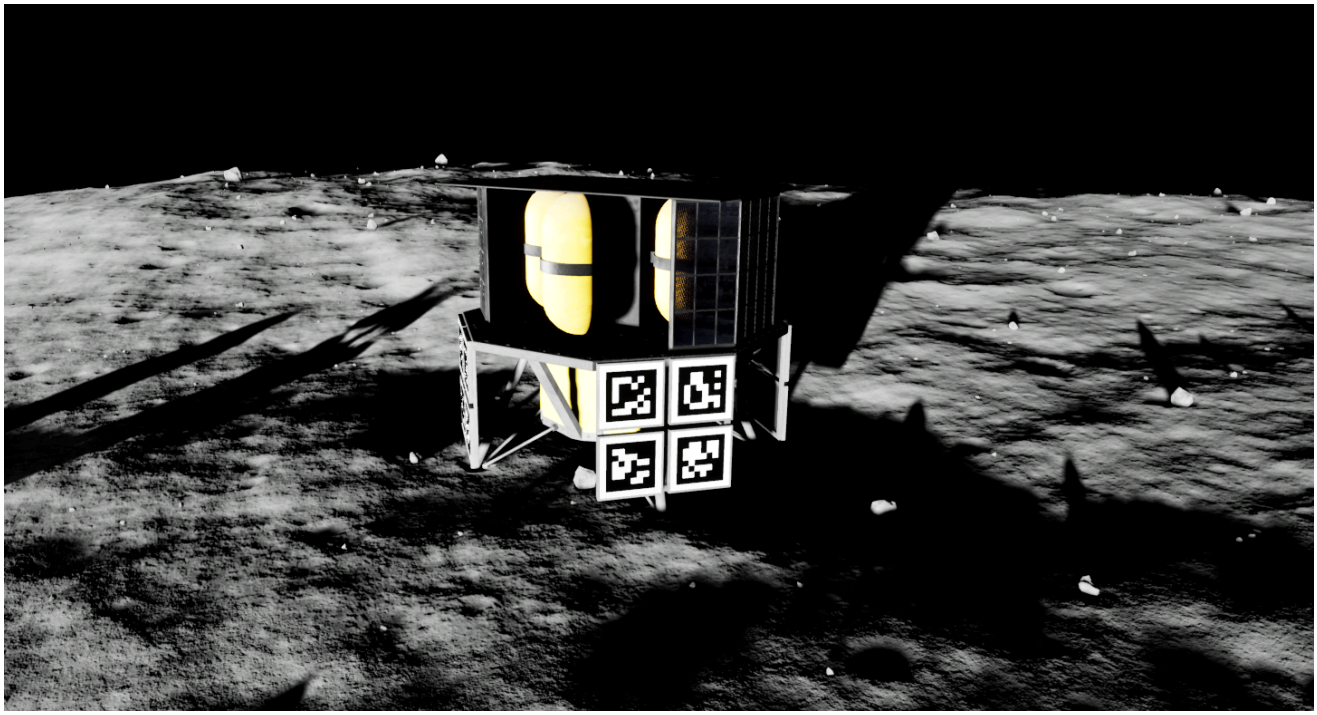


Figure 1.4: Image of the lander as it appears in the simulation. Note the April tag fiducials around the lander.

The lander is the module that deploys the rover on the lunar surface. It is decorated with bright colors and has optional fiducial markers to serve as a visual reference point to assist with the rover's self-localization. **The lander is placed in the middle of the map**, its geometric center may be slightly displaced from the exact origin due to the uneven terrain.

Fiducials

The fiducial markers are patterns of black and white retro-reflective material attached to the lander body to assist with localization. The fiducials are April tags chosen from the **36h11 family**, arranged around the lander in the patterns shown in figure 1.5 to help distinguish each side of the lander and locate the charging antenna:

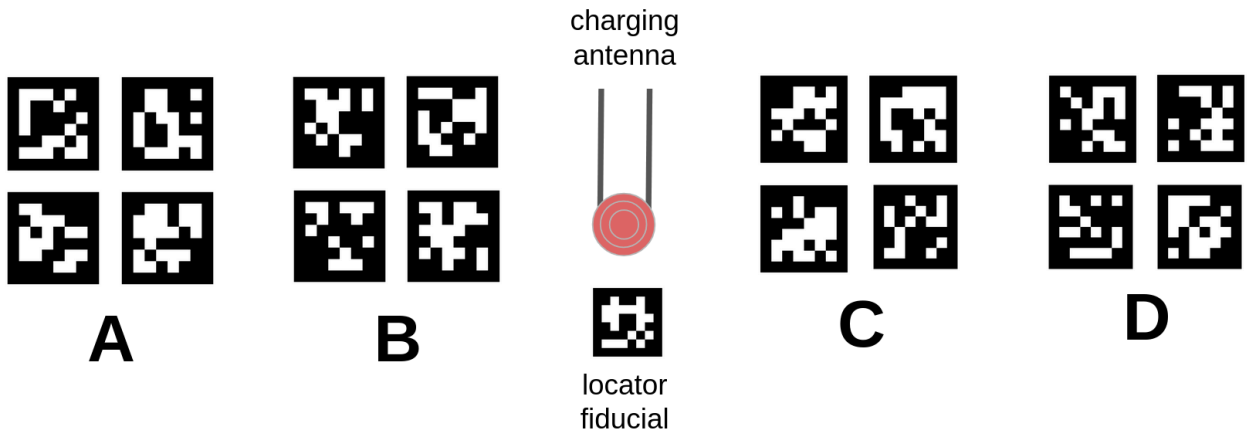


Figure 1.5: Diagram of the April tag fiducial groups with the relative position of the charging antenna.

The fiducial groups are labelled as groups A, B, C and D. A charging antenna is located between groups B and C. Groups A and D are on the opposite side of the lander to the charging antenna. Please see the geometry section for details on the exact position of the fiducial markers with respect to the lander's coordinate frame.

There is an additional April tag located on the lower body of the lander behind the charging antenna for assistance with charging alignment.

Using the fiducials is optional. You can choose whether to use them or not when you configure your autonomous agent. You will score extra points for navigating without them.

Wireless charging

The lander serves as a power source for the rover when it requires additional battery power. A wireless charging antenna protrudes from one side of the lander and the rover has a charge receiving antenna on the top of its radiator cover. The rover must approach the lander's charging antenna and then raise the radiator cover to bring the charging source antenna and receiving antenna within the correct range. If the arms are raised to a high angle, they may obstruct the radiator cover from fully opening.

The receiver must be within 10cm of the source horizontally and there must be an angle of no greater than 0.52 radians (30 degrees) between the axes of each antenna.

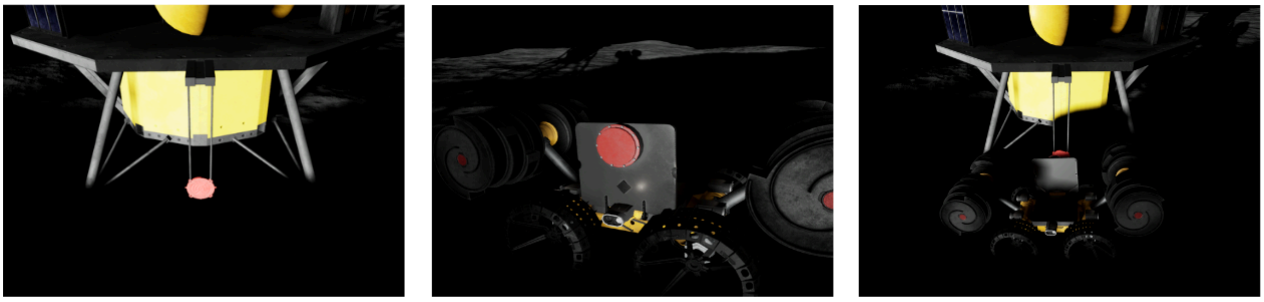


Figure 1.6: Images of the charging antennas on the lander and rover. The image on the right shows the rover in the charging position.

Recharging from an empty battery requires 2 hours of mission time. The recharging time scales linearly from 2 hours to zero according to the remaining power in the battery. For example, recharging from 50% battery will take 1 hour, while recharging from 75% battery will only take 30 minutes. Once the charging ports have been brought within range, the mission time will advance by the required charging time and the battery will return to 100%.

Note that the mission time expended during recharging does not incur additional compute time. In real time and compute time the charging occurs almost instantaneously during a single simulation step, while the mission time inside the simulation is advanced by the computed amount. **You should monitor the charge state of your battery and the mission time through the API to verify that charging has succeeded.**

Please see the geometry section for details on the exact position of the charging antenna with respect to the lander's coordinate frame.

Moon maps

The lunar simulator comes with 2 moon surface maps to use in the development of your agent. They are identified as `Moon_Map_01` and `Moon_Map_02`. The maps are both **40m x 40m** in size with a mapping area of **27m x 27m**. There is a 6.5m buffer between the edge of the map and the edge of the mapping area. The mapping area is divided up into cells of **15cm x 15cm**, hence **180 x 180 cells**, making a total of **32,400 cells** to map. Each cell is associated with an X-coordinate and Y-coordinate at its center, an average height or elevation and a boolean flag indicating the presence of a rock (or part of a rock) in the cell. Note that in the qualifying stage the mapping area is reduced to **9m x 9m**.

It is the task of your agent to assign an estimated height or elevation to each map cell, along with an indication of whether or not your agent has detected a rock in the cell. It's important to note that the lander serves as a visual reference of the center of the mapping area, whether or not you choose to use fiducial markers.

The terrain of the maps is modelled as a deformable surface to mimic lunar regolith. When the rover drives over any section of the map, the wheels may sink 2-3cm into the soil and leave visible wheel tracks. Ground truth surface elevation of the terrain against which your estimates are compared is calculated after the terrain has been deformed by the rover's wheels. However, the terrain deformation caused by the rover has a minimal impact when averaged over a cell. The rocks are immovable and cannot be displaced by the rover.

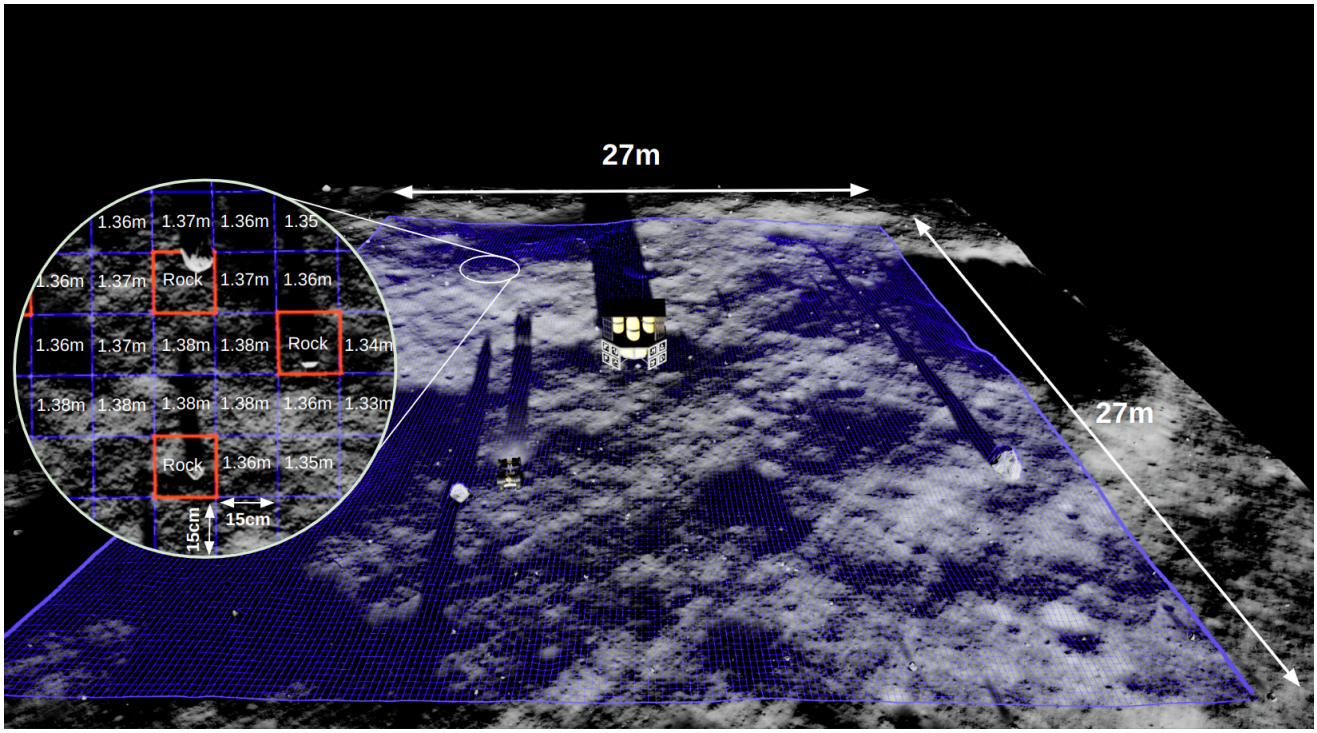


Figure 1.7: Mapping area.

When you choose to submit your agent to the competition, it will be tasked with mapping a 3rd competition map for which the ground truth is not provided, to avoid prior knowledge influencing mapping performance.

Getting started

Updated - 3rd September 2024

Installation

The following section demonstrates how to set up your computer then install the lunar simulator and the Leaderboard software to start the development of your agent.

Recommended minimum hardware and software

Recommended minimum hardware specification

- Intel core i7 or core i9 processor or equivalent with 6+ cores
- NVIDIA high-performance GPU - RTX 3000 series or better
- 32 Gb system memory (RAM)
- 16 Gb graphics memory (VRAM)
- Minimum of 16 concurrent threads available on the CPU
- Hard disk drive with a minimum of 50Gb of free space

Software prerequisites

- Computer running **Ubuntu** version **20.04** or **22.04**
- Python **3.8**, **3.9** or **3.10**
- Python PIP package manager **version 20.3 or higher**
- GPU drivers: install the **latest official drivers** for your GPU model - by default, Ubuntu does not use the official drivers and uses a generic driver which may cause performance issues or crashes

System setup

To run the lunar simulator you will need to set up a system with the appropriate hardware and software to handle all the processing duties with good performance. Inadequate hardware may run the simulator slowly and render development iterations impractically slow. There are several key considerations in the hardware specification of your machine:

- **Central processing unit (CPU)** - the primary processor. This processor runs the operating system and all the currently open software. Modern CPUs have numerous cores so they can parallelize processing tasks.
- **System memory (RAM)** - the system memory is fast-access volatile memory that stores data needed by the operating system and applications. It is connected directly to the CPU.
- **Graphics processing unit (GPU)** - the GPU is a secondary processor optimized to handle graphics calculations. The GPU is highly parallelized since graphics processing tasks can be split into many small, identical processing tasks.
- **Graphics memory (VRAM)** - the graphics memory is fast access volatile memory connected directly to the GPU located on the graphics card. This will hold the 3D data needed to render the lunar scene.
- **Motherboard** - the motherboard is the circuit board to which the CPU and system memory are attached. It must have one or more spare PCI-e slots capable of accepting the graphics card.
- **Graphics card** - the graphics card is the chassis that holds the GPU and VRAM, it is attached to the motherboard through a PCI-e slot. The GPU communicates with the CPU through the PCI bus.
- **Hard disk** - the hard disk is slow-access persistent memory where large volumes of data are stored long term.

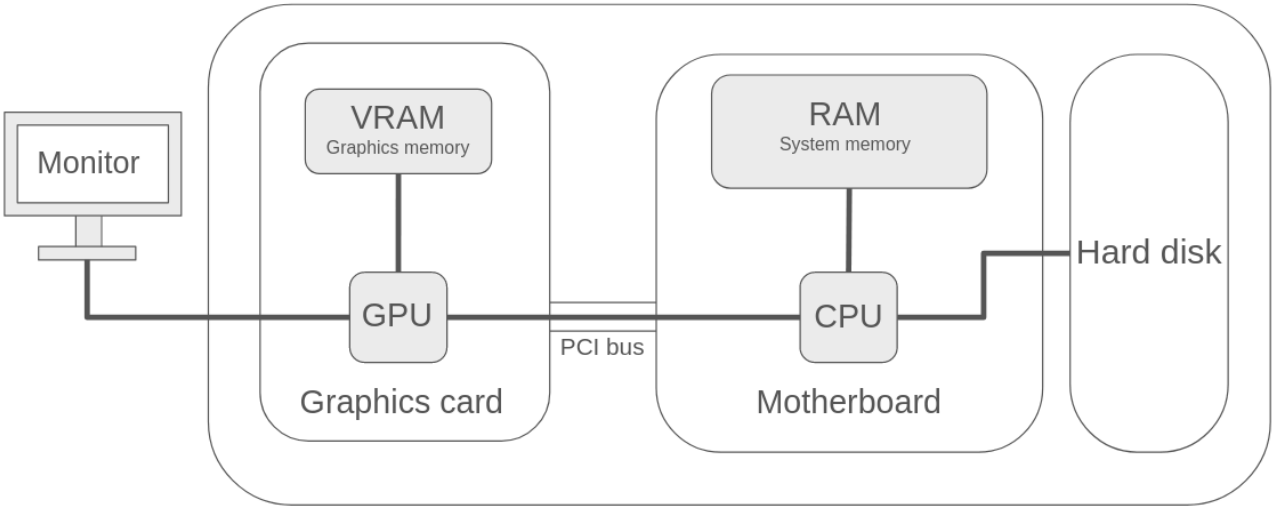


Figure 2.1: Diagram of the important system components required to run the simulator.

You can check your system configuration in `Settings > About` from the Ubuntu system tray (where the power and sleep options are).

The simulation exploits parallel computing in order to simulate how light rays travel around the virtual scene and create a signal in the virtual camera. For this rendering workload, the simulator uses the graphics processing unit (GPU) - a specialized parallel computing device optimized for this task. Therefore, high-performance graphics hardware is required for optimal simulation execution.

The simulator also computes the physical interaction between the rover and the moon surface. This task utilizes the central processing unit (CPU). The workload can be separated into separate parallel tasks that can be executed in multiple threads on a multi-core CPU.

You should use a CPU with at least 4 physical cores. We recommend 6 physical cores or more. Simulation performance will also benefit from hyper-threading. Enable the hyper-threading feature in your BIOS or UEFI settings if available. The computer or virtual machine must be able to run a minimum of 16 concurrent threads.

The 3D models of the virtual moon surface, rover and lander are very detailed. Rendering this 3D data requires a significant amount of memory. Since high-speed repeated access is needed to this data, it cannot be stored on the hard disk, which has slow read/write capabilities, the 3D data must be stored in fast-access graphics memory or VRAM.

It is likely that many solutions for this challenge will utilize deep learning techniques using Python libraries such as TensorFlow, Keras and PyTorch. These libraries normally make use of the GPU to handle network calculations and require a substantial amount of graphics memory to store the network weights. Therefore, the graphics card needs to have enough graphics memory (VRAM) to simultaneously store the 3D data for the simulator and the network weights. We recommend a minimum of 16Gb of graphics memory.

Installing the prerequisites

Ubuntu

If you haven't already, install Ubuntu version 20.04 or Ubuntu version 22.04 using the instructions given here. If your machine already runs Ubuntu, check which version you have installed. Open a terminal and run the following command:

```
lsb_release -a
```

If the output of this command shows that your Ubuntu version differs from 20.04 or 22.04 you need to update to one of the recommended versions to continue. The simulator will not work on older or newer versions of Ubuntu.

Graphics drivers

Next, you should check that you have the appropriate drivers installed for your graphics card. Run the following command:

```
nvidia-smi
```

The output of this command should include the name of your graphics card such as *NVIDIA Geforce RTX 3090*. If you cannot see the name of your graphics card or `nvidia-smi` is not installed, you will need to install (or reinstall) the latest drivers for your graphics card.

You can examine which drivers are currently installed on your system using the following command:

```
sudo ubuntu-drivers list --gpgpu
```

Now install the appropriate GPU driver for your system, then re-run `nvidia-smi` to check the installation:

```
sudo ubuntu-drivers install
```

This should work for a vanilla installation of Ubuntu 20.04 or 22.04. If this doesn't work for your system setup, you may need to manually install the correct driver using the instructions given here.

Python

Next, ensure you have an appropriate version of Python installed. Run the following command in a terminal:

```
python3 --version
```

If you are running Ubuntu 20.04, by default Python 3.8 will already be installed. If you are running Ubuntu 22.04, by default Python 3.10 will be installed. If your system has another version of Python installed, you must install one of versions **3.8**, **3.9**. or **3.10** to use the lunar simulator's Python API. If you wish to use a version of Python other than the system Python, we strongly recommend using a virtual environment manager such as Conda or PyEnv.

If you do not have Python already installed, you may use the following command to install Python and PIP:

```
sudo apt-get install python3 python3-pip
```

If you already have Python installed, you should ensure that you also have the Python package manager PIP installed:

```
pip3 --version
```

PIP must be version **20.3** or higher. If it is not installed, follow these instructions. If the version requires updating, run the following command:

```
pip3 install --upgrade pip
```

LibVulkan

The Lunar simulator also requires LibVulkan, a graphics library, to run optimally. This may have been installed with the graphics drivers. To ensure it is installed Run the following command to install LibVulkan:

```
sudo apt-get install libvulkan1
```

Operating system updates

Operating system updates might interfere with your system setup. **We strongly recommend not to accept system updates while working with the lunar simulator to avoid problems with graphics drivers**.

Download the LAC simulator package

If you haven't already, download the LAC package and extract it. Find an appropriate location for the package on your computer. Open a command terminal inside the extracted folder.

Install the Python dependencies

In a terminal window, navigate to the root folder of the extracted package. Install the Python dependencies with PIP:

```
pip3 install --force-reinstall -r requirements.txt
```

If you are using a virtual environment manager for Python like conda or pyenv, you should activate the relevant virtual environment before running the above command.

> **Note**
>
> If you already have a CARLA Python wheel installed via PIP for another CARLA version please note that the above command will automatically remove it due to the `--force-reinstall` option. Alternatively, you may also use a virtual environment manager like Conda or PyEnv.

Running the lunar simulator

Run the launch script for the lunar simulator. From the root folder of the extracted package run the `RunLunarSimulator.sh` launch script:

```
./RunLunarSimulator.sh
```

> **Note**
>
> **Don't shut this process down**, the simulator needs to be running when the Leaderboard is launched. Only shut this process down when you are finished with your session.

This command should open a new window on your desktop, this is the *spectator* camera view of the lunar simulator. It may be black for several seconds while the simulator initializes, but eventually, you should be able to see the moon map. It will be dark, since the moon maps are simulated with a very low inclination of the sun over the horizon.

The *spectator* is a floating camera that you can use to observe the scene for inspection and debugging purposes. You can examine what has happened to the rover during a mission. Drag the mouse across the window to turn the camera, use the WASD keys (forward, left, backwards and right respectively) and E and Q (up and down) to translate the camera.

The spectator view is quite dark by default to the the low simulated lighting angle and it can be hard to find the rover if it is in a shadowed area of the map. A debug illumination mode is provided that provides more visibility in the scene. To activate the debug illumination mode press L on the keyboard. Note that this will not affect the view of the camera sensors, only the spectator.

> **Note**
>
> The lunar simulator must always be launched before running the Leaderboard. If it crashes or freezes, you should re-launch it before running the Leaderboard.

Running the Leaderboard

Ensure that you have the simulator running in its own terminal, then open another terminal in the root folder of the package and run the Leaderboard using the `RunLeaderboard.sh` convenience script:

```
./RunLeaderboard.sh
```

This will launch the Leaderboard with the *Human Agent* by default (it may take several seconds to load the map and data). The human agent allows you to control the rover manually through the Leaderboard with the keyboard.

You may now manually control the rover with the keyboard using the following commands:

- Arrow keys: ↑ - forward, ↓ - backwards, ← - pivot left, → - pivot right

- WASD keys: W - forward, A - pivot left, S - backwards, D - pivot right
- `X` - open, or `C` - close the radiator cover
- `G` - raise, or `B` - lower the front arm
- `H` - raise, or `N` - lower the rear arm
- Press `i` for an on-screen menu of other key commands

Press `Esc` to finish the mission.

> **Note**
>
> **You should wait before closing the simulator or Leaderboard** after escaping the *Human Agent*. The Leaderboard will take some time to calculate the map elevations and output the evaluation data. This could take up to 1 minute.

The `RunLeaderboard.sh` script is a convenience script used to set up filepaths and parameters to run the Leaderboard. Open the script in a code editor to make adjustments when you are ready to execute the Leaderboard using your own agent. This script is a convenience script that runs the `leaderboard_evaluator.py` Python script. For more options, or to set up debugging, you may wish to run the Python script directly.

The following sections define the parameters of the `RunLeaderboard.sh` script you should modify to run your agent in the Leaderboard and change the map or preset you wish to use for testing. You may also change the operation conditions of the Leaderboard to facilitate faster development.

Agent file

You are now ready to create your own custom agent. We advise you to create your own agent in the `agents` folder, where the `human_agent.py` file is located. We have provided a Dummy Agent in the `dummy_agent.py` file. The dummy agent makes some simple control inputs and fills some cells of the geometric map randomly to demonstrate how to arrange the code for your custom agent. You may use the `dummy_agent.py` file as the basis for your own agent. Please refer to the section on creating an agent for details on writing your own agent.

To run your own agent in the Leaderboard, you should adjust the following line in the RunLeaderboard.sh script to point to your own agent file:

```
#export TEAM_AGENT=$LEADERBOARD_ROOT/leaderboard/autoagents/human_agent.py

export TEAM_AGENT=$LEADERBOARD_ROOT/leaderboard/autoagents/my_agent.py
```

Missions file

You may also want to modify the missions file if you want to change to the other moon map, or run multiple missions:

```
#export MISSIONS=$LEADERBOARD_ROOT/data/missions_training.xml

export MISSIONS=$LEADERBOARD_ROOT/data/my_missions.xml
```

To change to the second moon map, your `my_missions.xml` file should look like this:

```
<missions>
  <mission id="0" map="Moon_Map_02" preset="10"/>
</missions>
```

You can run multiple missions in a single Leaderboard execution:

```
<missions>
    <mission id="0" map="Moon_Map_01" preset="0"/>
    <mission id="1" map="Moon_Map_02" preset="10"/>
</missions>
```

There are 2 moon maps available, `Moon_Map_01` and `Moon_Map_02`, they are the same size but have different surface variations. Each preset defines a different, random distribution of rocks. **Presets 0-10 are for** `Moon_Map_01` **and presets 11 - 13 are for** `Moon_Map_02`. Additionally **presets 0 and 11 have no rocks**. Please avoid using the wrong preset for a map as it will produce an error. We encourage the use of `Moon_Map_01` as the **training environment** and `Moon_Map_02` as the **validation environment**.

Other `RunLeaderboard.sh` options

The `RunLeaderboard.sh` script offers several other options to manage the execution of the Leaderboard.

- **DEVELOPMENT**

The **development mode** skips the calculation of the mission statistics, to speed up debugging and development iterations. If you wish to activate development mode, adjust the following line in the `RunLeaderboard.sh` script:

```
export DEVELOPMENT=1
```

- **QUALIFIER**

The **qualifier mode** launches the mission in qualifier conditions, with a reduced mapping area of 9x9 meters and 1 hour of mission time. If you wish to activate qualifier mode, adjust the following line in the `RunLeaderboard.sh` script:

```
export QUALIFIER=1
```

- **EVALUATION**

The **evaluation mode** launches the mission in the same competition or qualifier conditions (if the QUALIFIER parameter is set) as in the cloud. The use of ground truth data provided by the simulator (such as the ground truth transform or semantic segmentation data) is disabled. Ground truth location through the `get_transform()` method is disabled, it will return `None`. Using semantic camera data will produce an error in evaluation mode. If you wish to activate the evaluation mode, adjust the following line in the `RunLeaderboard.sh` script:

```
export EVALUATION=1
```

You should use this mode to ensure you are not using ground truth information in your agent once you are nearing submission readiness.

- **SEED**

Integer seed for the random initialization of the rover. The rover is initialized in the same randomly chosen location in the map by the Leaderboard with each execution. To initialize the rover in different randomly chosen locations, you can provide different seeds for the random number generator.

```
export SEED=10
```

- **MISSIONS_SUBSET**

This environment variable specifies a subset of missions from the `missions.xml` file to execute. A hyphen indicates to run all missions located between the two given IDs in the file, use a comma between separate missions.

```
export MISSIONS_SUBSET="2-4,6,8"
```

For example, the configuration above runs all missions from 2 to 4, then 6 and 8. The mission IDs are handled as strings, so all missions located between the hyphenated IDs in the file will be executed.

- **REPETITIONS**

Sets the number of repetitions to run for each mission in the `missions.xml` file or mission subset.

- **CHECKPOINT_ENDPOINT**

Sets the directory location for the results files.

- **RECORD**

Instructs the Leaderboard to record the mission as a simulator log (all the movements of the rover), so it can later be replayed in the simulator. To activate the recorder, set the environment variable like so:

```
export RECORD=1
```

Leave the variable empty if you do not wish to use the recorder.

- **RECORD_CONTROL**

Instructs the Leaderboard to record the telemetry of the rover in a json file. To record control data, set the environment variable like so:

```
export RECORD_CONTROL=1
```

Leave the variable empty if you do not wish to record the controls.

- **RESUME**

Instructs the Leaderboard to resume execution of a mission (sub)set from the last completed mission, in the case that the simulator crashes or another problem occurs.

```
export RESUME=1
```

Outputs

Once the Leaderboard has finished executing a mission, it will output the results in the `results` folder at the top level of the package directory (or the directory you specified in the CHECKPOINT_ENDPOINT variable). The files will be named according to the following convention:

```
{map}_{id}_rep{rep_id}.txt (or .dat, .json, .log)
```

By default, 5 files will be created, for example, if we run a mission with ID=0, using Moon_Map_01, then the files for the first repetition will be named like so:

- `results.json` -> the metrics defining your agent's mission performance
- `Moon_Map_01_0_rep0.txt` -> the ground truth height map, in text format for easy inspection
- `Moon_Map_01_0_rep0.dat` -> the ground truth height map in Numpy format, ready to be loaded as a Numpy array
- `Moon_Map_01_0_rep0_agent.txt` -> the height map estimated by your agent, in text format for easy inspection
- `Moon_Map_01_0_rep0_agent.dat` -> height map estimated by your agent in Numpy format, ready to be loaded as a Numpy array

Recording a mission

If you set the RECORD variable, the simulator will record the progress of the simulation, including all movements of the rover along with terrain deformation. These will be recorded into log files in output directory.

This will produce a substantial amount of data, therefore a separate log file will be produced for each 30 minutes of simulation time.

The log files will be named as `Moon_Map_01_0_rep0_chunk{p}.log` where p is an integer that increments for each 30 minute time period.

The telemetry will be recorded in a file named `Moon_Map_01_0_rep0_agent_chunk{p}.json` where p is an integer that increments for each 30 minute time period, if you set the RECORD_CONTROL variable.

Re-player

After making a recording of a mission, the simulation re-player allows users to visually review the agent's behavior during a previously executed mission for debugging purposes. **The re-player can run the playback in real time, including in cases where the simulation itself was significantly slower than real-time due high runtime computational demands**.

Inside the `Leaderboard/scripts` directory you will find the `recorder_replay.py` script that runs the re-player function. Run the script with Python like so:

```
python3 recorder_replay.py -f <PATH_TO_LOG_FILE>
```

The file path should be an absolute path to the log file you wish to re-play.

Optional arguments:

- `--start-time`: desired start time in seconds
- `--end-time`: desired end time in seconds
- `--factor`: playback speed as a multiple of real time, which is 1.0
- `--static-sky`: keeps the Sun and Earth static in the lunar sky during playback

During playback, you may fly the spectator around the scene while the rover moves around the map to inspect the progress of the mission.

While the replayer is running, you may also adjust the playback speed by typing a number at the command prompt. Alternatively you can stop the playback by typing `S` and record a timestamp by typing `R`. You can press the `F` key to toggle spectator follow mode, where the

specator follows the rover. The re-player will continue even after the simulation playback has completed, therefore you should type `s` to terminate the script.

Creating an autonomous agent

Updated - 3rd September 2024

To create an autonomous agent to run in the Leaderboard, you need to create a Python file using the following guidelines. Your agent Python file should be identified in the TEAM_AGENT variable of the `RunLeaderboard.sh` launch script.

In a new Python file, you should implement the following method overrides and any other methods needed for your autonomy solution. After setting up the agent with the setup methods, every time step of the simulation calls the `run_step()` method. Inside this method, your agent will process the sensor data and then compute a control command to send back to the vehicle.

Use the AutonomousAgent class as a base class

The definition of an agent starts by creating a new class that inherits from the AutonomousAgent class of the Leaderboard:

```
leaderboard.autoagents.autonomous_agent.AutonomousAgent
```

Create an entry-point method

You should define a function called `get_entry_point()` that returns the name of your agent class. This will be used by the Leaderboard to instantiate your agent at the beginning of mission execution:

```
from leaderboard.autoagents.autonomous_agent import AutonomousAgent

def get_entry_point():
    return 'MyAgent'

class MyAgent(AutonomousAgent):
...
```

Define the setup procedure

Now you should override the `setup(...)` method inherited from the **AutonomousAgent** class to set your agent up. In this method, you should execute all the procedures necessary to initialize your agent to control the rover. For example, here you may want to initialize neural networks, load weights files and initialize any objects or attributes you need. **It is recommended that at some early point in the mission you should call the `get_initial_position()` method** to begin your mapping process with the correct initial position of the rover. You may want to call it in the setup method.

```
class MyAgent(AutonomousAgent):

...

    def setup(self, path_to_conf_file):
        # Execute setup procedures

...
```

The `path_to_conf_file` variable is optionally used to refer to any configuration file you may want to use to set up your agent. For example, the config file could be a JSON containing the location of a network weights file you wish to use, or configuration parameters for controllers. This enables you to make tweaks and changes to your agent without having to edit the core code. The value for this variable is provided as an argument to the Leaderboard evaluator script and can be set in the `RunLeaderboard.sh` script.

Override the `use_fiducials()` method

Override this method to return True if you wish to use fiducials and False if not. You will score extra points for not using the fiducial markers for navigation. Please see the evaluation metrics section for further details.

```
  ...
  def use_fiducials(self):
    return True
  ...
```

Setup the initial sensor configuration

By default, all the cameras and lights of the rover will be deactivated. You should initialize at least a subset of the cameras and lights to start navigating the map. To do this, override the `sensors(...)` method of the **AutonomousAgent** class. For example, to initialize the rover with all front facing cameras and lights activated you would write the following method:

```
class MyAgent(AutonomousAgent):

    ...

    def sensors(self):

    ...

        # Activate all front facing cameras and lights on...
        sensors = {
            carla.SensorPosition.Front: {
                'camera_active': True, 'light_intensity': 1.0,
                'width': '2448', 'height': '2048', 'use_semantic': False},
            carla.SensorPosition.FrontLeft: {
                'camera_active': True, 'light_intensity': 1.0,
                'width': '2448', 'height': '2048', 'use_semantic': False},
            carla.SensorPosition.FrontRight: {
                'camera_active': True, 'light_intensity': 1.0,
                'width': '2448', 'height': '2048', 'use_semantic': False},
            carla.SensorPosition.Left: {
                'camera_active': False, 'light_intensity': 0,
                'width': '2448', 'height': '2048', 'use_semantic': False},
            carla.SensorPosition.Right: {
                'camera_active': False, 'light_intensity': 0,
                'width': '2448', 'height': '2048', 'use_semantic': False},
            carla.SensorPosition.BackLeft: {
                'camera_active': False, 'light_intensity': 0,
                'width': '2448', 'height': '2048', 'use_semantic': False},
            carla.SensorPosition.BackRight: {
                'camera_active': False, 'light_intensity': 0,
                'width': '2448', 'height': '2048', 'use_semantic': False},
            carla.SensorPosition.Back: {
                'camera_active': False, 'light_intensity': 0,
                'width': '2448', 'height': '2048', 'use_semantic': False},
        }
        return sensors
```

Each sensor position is initialized with a dictionary containing keys for the state of the camera, the light and also the width and height in pixels of the camera. You can activate and deactivate the cameras and lights later in the simulation using the setter methods in the API. The `use_semantic` argument is optional, to activate the semantic counterpart of each camera, it is False by default if not set. **The maximum camera resolution permitted is 2448 x 2048 pixels**, if a higher resolution is requested the resolution will be clipped to the maximum and a warning will be given on the command line.

Implement the logic for a simulation time-step

The simulator runs in discrete time steps at 20Hz in simulation time. The `run_step(...)` method is executed with each simulation time-step by the Leaderboard. It is in this method where your agent receives sensor data from the simulation then processes the data and returns a control instruction to the simulator to move or modify the rover's state. The real time frame rate will depend upon the hardware used to run the simulator. The cameras run at 10Hz, therefore camera data is delivered on every other time-step.

Override the `run_step(...)` method with your agent's sensor and control logic:

```
def run_step(self, input_data):
    """
    Extract the sensor data from the input_data dictionary
    ...
    Process the sensor data.
    ...
    Decide on next control input with:
    lin_v -> Desired linear velocity
    ang_v -> Desired angular velocity
    """

    # Return the control input to the simulator
    return carla.VehicleVelocityControl(lin_v, ang_v)
```

The `input_data` argument consists of a dictionary containing the sensor data collected in the previous simulation step. There are 2 keys for the first level of the dictionary:

- `Grayscale`: access the standard, grayscale camera data
- `Semantic`: access the semantic segmentation data

The second level of the `input_data` dictionary for the `Grayscale` and `Semantic` fields will contain one or more of the following keys for the 8 cameras:

```
carla.SensorPosition.Front
carla.SensorPosition.FrontLeft
carla.SensorPosition.FrontRight
carla.SensorPosition.Left
carla.SensorPosition.Right
carla.SensorPosition.BackLeft
carla.SensorPosition.BackRight
carla.SensorPosition.Back
```

So, for example, to retrieve the data for the standard front left camera, you should use the following structure:

```
camera_data = input_data['Grayscale'][carla.SensorPosition.FrontLeft]
```

You should also check if camera data is delivered in the current time step. The returned data will be `None` if there is no camera data for the current time step.

> **Note**
>
> There will not be keys corresponding to inactive sensors. Sensors will return data in the form of Numpy arrays. Sensors that are active but did not return data in the previous simulation step will contain `None`. Semantic data is only provided for active cameras.

Sensor data

Monochromatic grayscale camera data is given in the form of a 3 dimensional Numpy array of unsigned, 8-bit integers (uint8) of size image-height by image-width by 1. The values scale between 0 and 255. The semantic cameras return a 3 dimensional Numpy array of unsigned, 8-bit integers (uint8) of size image-height by image-width by 3, with each layer of the last dimension representing one of the RGB color channels.

> **Note**
>
> Due to the frame rate of the cameras with respect to the simulation time-step frequency, they only deliver images every other simulation step. They will return `None` in between frames.

IMU data is retrieved through the `get_imu_data()` method of the API for each time-step. IMU data is given in the form of a Python list of length 6, containing the following values:

```
[ accelerometer.x, accelerometer.y, accelerometer.z, \
gyroscope.x, gyroscope.y, gyroscope.z]
```

The accelerometer measures in units of meters/s^2 and the gyroscope measures angular velocity in units of rad/s.

Vehicle control

Once your agent's control logic has consumed and processed the sensor data, you should then prepare a set of control instructions for the rover as attributes of a `carla.VehicleVelocityControl()` object. The `VehicleVelocityControl` object has 2 arguments for the desired linear velocity (in meters per second) and the desired angular velocity (in radians per second). Positive angular velocity turns the rover clockwise and negative angular velocity anti-clockwise, if looking down from above.

For example, to instruct the rover to aim for a linear velocity of 0.2 meters per second and an angular velocity of 3 radians per second, you would use the following:

```
def run_step(...):
    ...
    return carla.VehicleVelocityControl(0.2, 3)
```

You should also update the geometric map at least once during the mission before the simulation ends. However, we would recommend updating it iteratively, since the mission may be terminated unexpectedly and your score will depend upon the cells you have mapped up to that point. Use the `get_geometric_map()` method to obtain a reference to the **GeometricMap** object. The getters and setters of the **GeometricMap** object allow you to query and set cells of the geometric map as you complete the mapping task. Please see the API reference for details of the methods.

You may also want to query or modify information about other aspects of the rover's state using the getter and setter methods provided in the **AutonomousAgent** class.

Once your agent has completed the mapping task, call the `mission_complete()` method to instruct the Leaderboard that you have completed the mission. The simulation will then be terminated and your score will be calculated. The Leaderboard will output the data for your mission.

Override the `finalize(...)` method

You should override the `finalize()` method which will be called when you call `mission_complete()` or the Leaderboard otherwise terminates the simulation. In this method, you should clean up any objects which might be holding things in memory and finalize your

geometric map. It can be considered analogous to the destructor in object oriented programming methodology.

Set the TEAM_AGENT variable in the `RunLeaderboard.sh` script

The last step is to set the `TEAM_AGENT` environment variable in the `RunLeaderboard.sh` script to point to the Python file where you wrote the above code, then launch the Leaderboard with your custom agent:

```
#export TEAM_AGENT=$LEADERBOARD_ROOT/leaderboard/autoagents/human_agent.py
export TEAM_AGENT=$LEADERBOARD_ROOT/leaderboard/autoagents/my_agent.py
```

Submitting your agent to the Lunar Autonomy Challenge Leaderboard cloud

Updated - 3rd September 2024

Once you have trained and validated your agent locally on your own hardware resources, the next step is to submit your agent to the competition through the cloud system. In the cloud, your agent will be executed and scored in a mission on the unseen competition map and your score placed on a ranked results board.

There are two steps to making a submission to the Lunar Autonomy Challenge cloud system.

1. Package your agent in a Docker container
2. Submit the Docker container to the LAC cloud through EvalAI

Packaging your agent in a Docker container

To submit to the Leaderboard cloud your agent must first be packaged into a Docker container. Inside the extracted LAC package, you will find the `docker` folder, a template Docker file named `Dockerfile` is provided in this folder. Open this file in a code editor to make the changes needed to install and run your agent. We have provided the Docker files for a set of base images with some dependencies already installed for convenience.

If you don't already have Docker installed on your system, follow the Docker installation instructions to install the latest version. You may also want to follow the post installation instructions to allow management of Docker by non-root users to avoid relying on the `sudo` command.

First, you need to build the base image with the Leaderboard and its software requirements. Choose your preferred Ubuntu version and Python version and run the following command with the appropriate parameters:

```
./base/make_base_docker.sh --ubuntu-distro <UBUNTU_DIST> --python-version <PYTHON_VESRION
```

There are two choices of Ubuntu versions, **20.04** and **22.04**. For Ubuntu **20.04** you may choose between Python distribution **3.8** and **3.9**. For Ubuntu **22.04** you may use Python version **3.10**. Please avoid altering the base image unless it is strictly necessary, since it could cause unpredictable problems with your submission. Your software requirements can be added in the principal Docker file.

Next, build the image containing your agent upon your chosen base image. There are 3 options of base image, uncomment the relevant line to match the base image that you built in the previous step:

```
# FROM lac-leaderboard:ubuntu20.04-py3.8
# FROM lac-leaderboard:ubuntu20.04-py3.9
FROM lac-leaderboard:ubuntu22.04-py3.10
```

Include commands to install any prerequisites needed by your agent. For example you may want to install Python libraries or Ubuntu packages needed by your agent using the Docker `RUN` command and apt-get or Python PIP:

```
RUN apt-get update && apt-get install -y any-system-dependency
RUN python3 -m pip install torch
```

Then point to the locations of your agent's entrypoint Python file and configuration file (only edit the filename, not the TEAM_CODE_ROOT environment variable):

```
ENV TEAM_AGENT ${TEAM_CODE_ROOT}/<AGENT_ENTRYPOINT_FILENAME>
ENV TEAM_CONFIG ${TEAM_CODE_ROOT}/<CONFIG_ENTRYPOINT_FILENAME>
```

Finally, run the `make_docker.sh` script in the `docker` folder:

```
./make_docker.sh --team-code <PATH_TO_TEAM_CODE_ROUTE> --target-name <IMAGE_NAME>
```

This command will copy your code from the path provided into the Docker context. This will build a Docker image in your local repository called `<IMAGE_NAME>`, which defaults to `lac-user` if this argument is omitted. Ensure that it has been created by running `docker image ls` to list the images in your local repository.

Testing your Docker image

Before you submit your Docker image to the cloud, it is a good idea to test that your agent is working properly inside the Docker image. This helps to detect any problems that might have arisen during preparation, such as missing files or dependencies. A test script is provided to test your image prior to submission.

To run a test, the Docker image will need to share the GPU with the host machine. The *nvidia-container-toolkit* is required to achieve this. Follow the instructions here to install the toolkit with apt and then the instructions here to configure your Docker installation to use the NVIDIA runtime.

To run the test:

1. Start the lunar simulator on the host (running on your local machine as you do during normal development):

```
./RunLunarSimulator.sh
```

1. Run your agent inside the Docker image that you built in the previous steps using the `test_docker.sh` script provided in the `docker` folder:

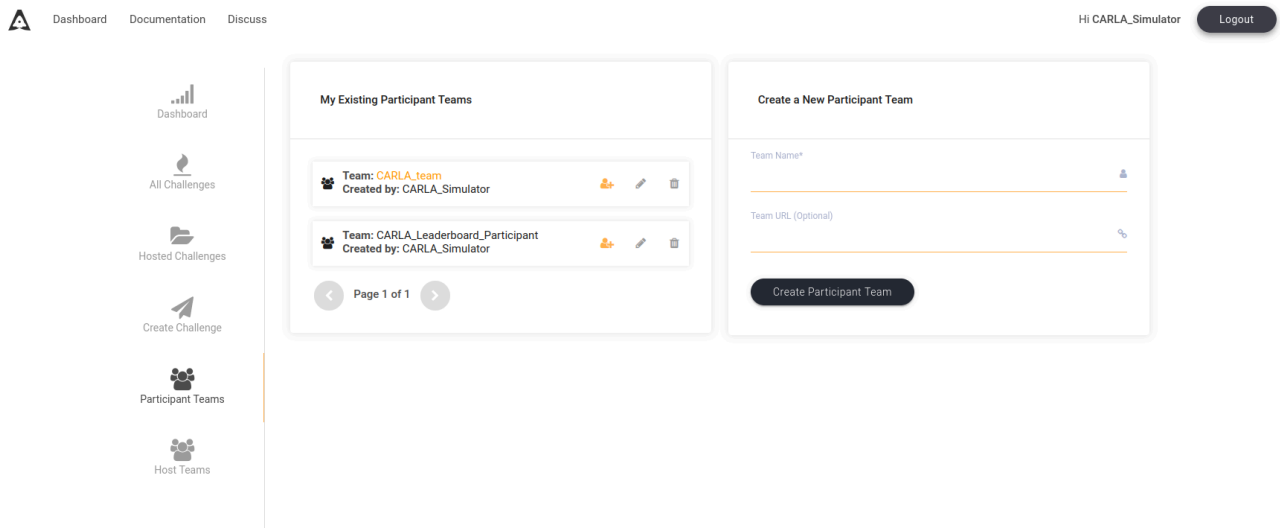
```
./test_docker.sh --image <YOUR_DOCKER_IMAGE> # by default lac-user:latest
```

This will run your agent in test mode, which will allow 10 seconds of execution. You will be able to see in the spectator if your agent starts moving the rover. You should see the same terminal output as you normally do running the Leaderboard. If your agent runs without problems, you are ready to submit your agent to the cloud.

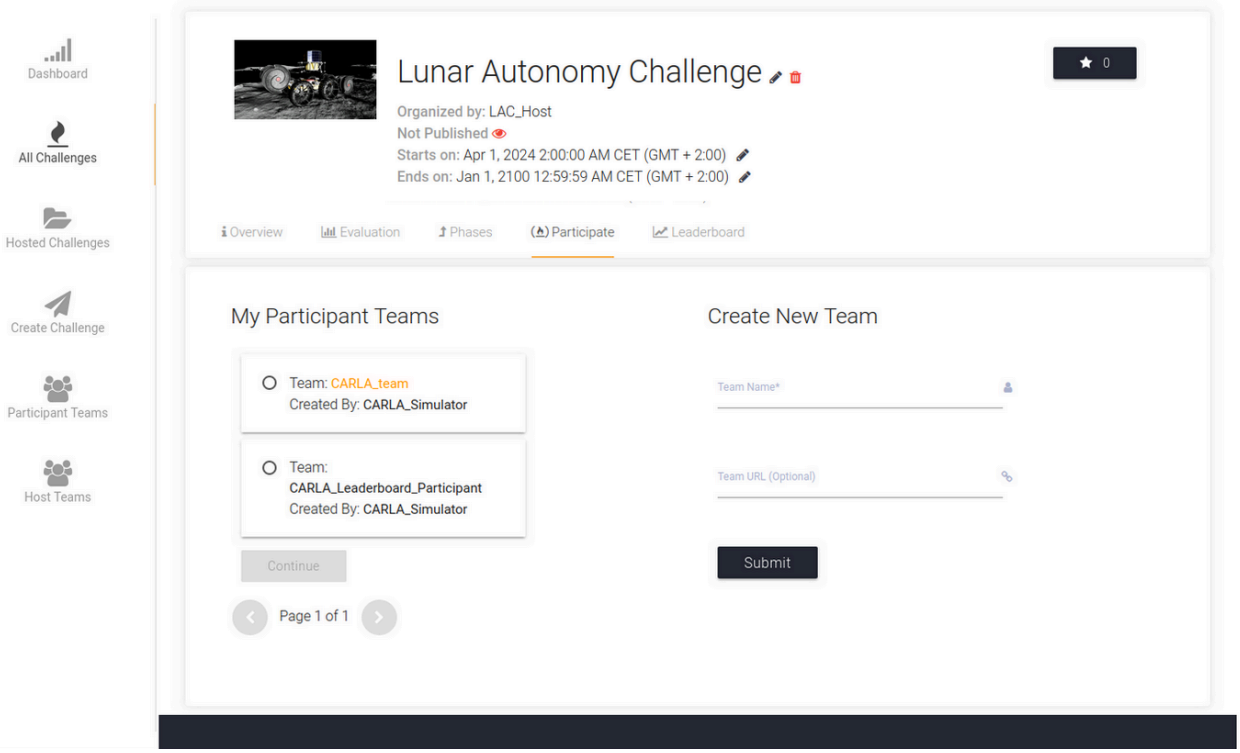
Submitting your agent to the cloud

You submit your agent to the competition Leaderboard through the EvalAI website. Navigate to the EvalAI website in your browser and create a new user.

After registering a new user, create a new team in the **Participant Teams** section:



After registering a team, apply to the Lunar Autonomy Challenge. Go to the **All Challenges** section and locate the Lunar Autonomy Challenge or follow this link. Select **Participate** and then choose the team with which you want to participate. **You will need to wait for your team to be verified by the admins**. Once your team is verified, you can start to make submissions.



Next, make your submission through the EvalAI command line interface (CLI).

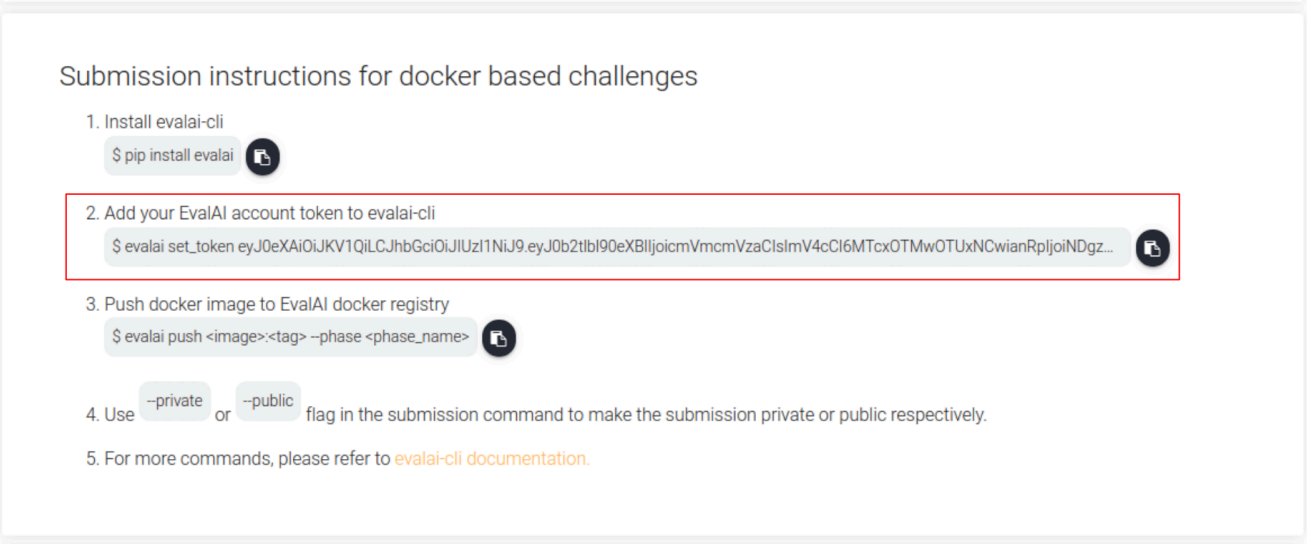
Install the EvalAI CLI with PIP:

```
pip install evalai
```

Now you need to identify your user with a token using the following command:

```
evalai set_token <token>
```

Your token is available from the **Submit** section of the EvalAI website. Copy and paste the command directly from the submission instructions:



> **Note**
>
> Please note that the token is specific to your user, each user will see a different token in this section of the website. Do not share tokens or there will be confusion over the submission author.

When you are ready to submit, use the following command to push your Docker image:

```
evalai push <image_name>:<tag> --phase <phase_id>
```

The `phase_id` can be found in the Submit section of the EvalAI competition page. After pushing your image, you can monitor the progress of your submission in the **My Submissions** section of the EvalAI interface. Each submission will start in the `SUBMITTED` status after successful submission. Once the system has started executing the submission it will progress

to `PARTIALLY_EVALUATED`. It will remain in this status until it either successfully finishes, in which case it will show the status `FINISHED`, or, if there has been an error, it will show the status `FAILED`.

Submissions that have failed can be resumed, in which case it will show the status `RESUMED`. You can choose to cancel a submission at any time, in which case it will show the status `CANCELLED`.

Competition evaluation metrics

Updated - 3rd September 2024

The Leaderboard evaluates the mapping performance of your agent at the end of a mission by comparing your height and rock maps with the ground truth and calculating a score according to the following criteria. Your agent is also subject to mission constraints that, if breached, may invoke an off-nominal mission termination.

Key metrics and weighting

The scoring is divided into 4 categories to characterize different aspects of your agent's performance, the following table outlines the points available in each category. The total score is the sum of the points scored in each section.

Metric	Weight	Target
Geometric map	30%	300 points
Rock map	30%	300 points
Mapping productivity	25%	250 points
Localization	15%	150 points
Total		1000 points

Geometric map

The geometric map is defined as a discrete grid of cells covering the competition area of 27x27 meters. Each cell is 15x15 cm, making a grid of 180x180 cells, with a total of 32,400 cells. Your agent must assign an estimated average height to each cell, it should not be the maximum or minimum height of the map surface inside the cell, but the average. For the qualifier stage, the mapping area is 9x9 meters, the cell size is the same.

The accuracy of height estimations will be affected by the fidelity of surface mapping by the perception system, using information from the cameras. It will also be affected by the accuracy of the self-localization of the rover.

The geometric map metric is calculated by comparing the average height estimated by the agent, h_i, for a cell with the ground truth average height, $h_{\text{truth,i}}$, for that cell. If the absolute value of the difference between these two values is below a predetermined threshold, $\hat{h}_0$, the cell is considered to be successfully mapped and the score for that cell, H_i, is 1 point. If the difference is larger than the set threshold, the cell scores 0 points. Unmapped cells will by default score 0 points. The presence of rocks in cells contributes to the ground truth average height, if the majority of a cell is occupied by a rock, the rock's profile will add significantly to the average height while small rocks will have a minimal impact. You do not need to estimate the average height of the terrain underneath a rock. The ground truth height of the map is calculated taking into account any disruption of the terrain made by the rover wheels.

The chosen threshold, $\hat{h}_0$, is 50mm, which is the maximum traversable change in surface height for the IPEx rover over the dimension of a cell. Mapping inaccuracies greater than this could cause the rover to enter terrain it cannot safely traverse during a mission.

The absolute difference in each cell, $\hat{h}_i$, is calculated as follows:

$$\hat{h}_i = |h_{\text{truth,i}} - h_i|$$

And then the score, H_i, for each cell is calculated as follows:

$$H_i = 1 \quad \text{if} \quad \hat{h}_i \leq \hat{h}_0 \quad \text{or} \quad H_i = 0 \quad \text{if} \quad \hat{h}_i > \hat{h}_0$$

The geometric map metric is then the sum of the scores from all the cells in the map divided by the total number of cells, N_{truth}, and then multiplied by the total geometric map score 300:

$$S_h = \frac{\sum H_i}{N_{truth}} \times 300$$

Rock map

Each map is populated with a set of rocks of varied sizes on the surface. Your agent is tasked with mapping the locations of the rocks. Using the same grid of 15cm x 15cm cells that the geometric map uses, your agent must assign a boolean value to indicate if it estimates that a rock is present within the boundaries of the cell (True) or not (False). The rock map score is then calculated in terms of the following factors:

- True positives (TP) - cells where your agent correctly predicts the presence of a rock
- False positives (FP) - cells where your agent incorrectly predicts the presence of a rock
- False negatives (FN) - cells where your agent does not predict the presence of a rock, but the ground truth cell contains a rock

Unmapped cells are by default assigned as False. The rock map metric is then calculated through the following formula:

$$S_{\text{rock}} = \frac{2 \times TP}{2 \times TP + FP + FN} \times 300$$

This formula is derived from the **F_1 score**, commonly used in machine learning. It emphasizes a balance between *precision* - the fraction of relevant instances (TP) among the total retrieved instances (TP+FP) - and *recall* - the fraction of relevant instances (TP) that were retrieved out of all relevant instances (TP+FN). This metric is multiplied by the total score of 300 to yield the rock map score.

Mapping productivity

The mapping productivity metric measures your agent's capacity to complete the mapping task in an acceptable amount of mission time. The target mission time is 24 hours to measure the total of 32,400 cells. Therefore the target mapping productivity is measured as 32,400 cells / 24 hours = 1350 cells/hr.

Your mapping productivity rate will be calculated as follows:

$$\text{Mapping productivity} = \frac{\text{\# of map cells populated}}{\text{mission time (hours)}}$$

If your calculated mapping productivity is greater than or equal to the target mapping productivity of 1350 cells/hr, you will be awarded the total 250 points. Otherwise the mapping productivity score will be calculated as follows:

$$\text{Mapping productivity score} = \frac{\text{Mapping productivity}}{\text{Target mapping productivity}} \times 250$$

The mapping productivity score is not affected by compute time unless your agent reaches the maximum limit of compute time allowed for the competition, in which case the mission will be terminated and the mapping productivity score will be calculated on the basis of the mission time and mapping progress up to that point.

Localization metric

The localization metric characterizes your agent's ability to localize itself and the lander without the help of fiducial markers. You can choose wether or not to use the fiducial markers in the setup stage of your agent. You will be awarded an extra 150 points for completing the mapping task without the fiducials.

Mission constraints

The following section details the mission constraints and conditions that will result in **off-nominal termination** of the mission. Your agent will be scored on the portion of the geometric map and rock map that you have completed before off-nominal termination.

Mission time

There is a simulation time limit within which your agent must complete the mapping task of **24 hours** of mission time. If the mapping task is not complete within the allocated 24 hours, the simulation will be automatically terminated and your agent will be scored on the basis of the mapping that has been completed. The allocated mission time is **1 hour** for the qualifier stage.

Cloud compute time

For submissions to the cloud, the maximum allowable compute time is **180 hours** for the main competition and **7.5 hours** compute time for the qualifier. Once this limit is reached, the mission will be terminated.

Unrecoverable conditions

Battery exhausted

The IPEx rover model is allocated a limited battery capacity. If the remaining charge in the battery drops below **5 Watt hours**, the mission will be terminated.

Rover inactive

If minimal translational movement (less than 0.1 m/s) is detected in the rover for a period of over **5 minutes**, the mission will be terminated early.

This might happen if, for example:

- the IPEx rolls over
- the IPEx becomes stuck in the deformable terrain or on a slope

Out of bounds

The rover must not stray too far outside of the mapping area. The mission will be terminated if the rover's position exceeds the perimeter at $X, Y = \pm 19.5$ meters (6 meters beyond the perimeter of the competition mapping area).

API reference

Updated - 3rd September 2024

The following utility methods are provided in the **AutonomousAgent** class. You should use these methods in your agent implementation to control and monitor the rover and mission status.

Mission status

- **get_initial_position**()
  - **Returns**: `Transform` object - the initial position of the agent in the first time-step of the mission in the global coordinate system. This function will return the same result at any point during the mission. The returned `transform` has the following attributes (location is in units of meters and rotation in units of radians):
    - `transform.location.x`
    - `transform.location.y`
    - `transform.location.z`
    - `transform.rotation.pitch`
    - `transform.rotation.roll`
    - `transform.rotation.yaw`
    - `transform.transform(x,y,z)` - method that applies the transform to the point (`x` , `y` , `z`)
- **get_initial_lander_position**()
  - **Returns**: `Transform` object as above - the initial position of the lander in the first time-step of the mission relative to the rover in the rover's local coordinate system. This method returns the same result at any point during the mission.
- **get_transform**()
  - **Returns**: `Transform` object as above - the position and rotation of the agent at the current time-step in the right-handed global coordinate system, for debugging and training. **This is disabled in evaluation mode and will return** `None` **, please ensure to remove it from your agent before attempting to submit to the Leaderboard cloud**.
- **get_mission_time**()

    - **Returns**: *float* - the elapsed mission time in seconds.
- **get_current_power**()
    - **Returns**: *float* - remaining power in Watt hours.
- **mission_complete**()
  Call this method when your agent has finished the mapping task to terminate the mission. The Leaderboard will then output the results files.
- **use_fiducials**()
  Override this method to return True if you wish to use fiducials (which incurs a penalty on the final score) and False if you do not wish to use them.

Rover control

Movement

- **get_linear_speed**()
    - **Returns**: *float* - linear speed in meters per second.
- **get_angular_speed**()
    - **Returns**: *float* - angular speed in radians per second (positive is clockwise).

Note that the values returned by these getters are calculated from wheel odometry and may differ from the ground truth velocities.

Cameras

- **get_camera_state**(**camera**)
  Gets a camera's state
    - **Parameters**:
        - `camera`: key for camera location
    - **Returns**: `True` if active, `False` otherwise.
- **set_camera_state**(**camera**, **camera_state**)
  Sets the camera state
    - **Parameters**:

    - `camera`: key for camera location
    - `camera_state`: boolean - True for active, False for inactive
- **get_camera_position**(**camera**)
  Returns the chosen camera position and orientation relative to the rover's coordinate frame.
    - **Parameters**:
        - `camera`: key for camera location
    - **Returns**: `Transform` object:
      The returned `transform` has the following attributes (location is in units of meters and rotation in units of radians):
        - `transform.location.x`
        - `transform.location.y`
        - `transform.location.z`
        - `transform.rotation.pitch`
        - `transform.rotation.roll`
        - `transform.rotation.yaw`
        - `transform.transform(x,y,z)` - method that applies the transform to the point (`x`, `y`, `z`)

IMU

- **get_imu_data**()
    - **Returns**: `list` - a Python list containing the accelerometer measurements in m/s^2 and gyroscope measurements of angular velocity in rad/s.

```
[accelerometer.x, accelerometer.y, accelerometer.z, \
gyroscope.x, gyroscope.y, gyroscope.z]
```

Lights

- **get_light_state**(**light**)
    - **Parameters**:
        - `light`: key for light location

    - **Returns**: *float* between 0.0 (off) and 1.0 (full power)
- set_light_state(light, light_state) Sets the light state
    - **Parameters**:
        - `light`: key for light location
        - `light_state` :*float* between 0.0 (off) and 1.0 (full power)
- get_light_position(light)
  Returns the chosen light position and orientation relative to the rover's coordinate frame.
    - **Parameters**:
        - `light`: key for light location
    - **Returns**: `Transform` object:
      The returned `transform` has the following attributes (location is in units of meters and rotation in units of radians):
        - `transform.location.x`
        - `transform.location.y`
        - `transform.location.z`
        - `transform.rotation.pitch`
        - `transform.rotation.roll`
        - `transform.rotation.yaw`
        - `transform.transform(x,y,z)` - method that applies the transform to the point (`x` , `y` , `z`)

Rover arms and drums

Move front or back arms to `angle` radians (positive values elevate the arms above the rover chassis):

- set_front_arm_angle(angle)
- set_back_arm_angle(angle)

The maximum permitted value is 2.36 radians (135 degrees).

Rotate the drums at `speed` radians per second:

- set_front_drums_target_speed(speed)
- set_back_drums_target_speed(speed)

Positive speed rotates the drums in the *excavating* direction, such that the highest part of each drum turns away from the rover chassis.

Radiator cover

Get or set the position of the radiator cover - open (`carla.RadiatorCoverState.Open`) or closed (`carla.RadiatorCoverState.Close`)

- set_radiator_cover_state(radiator_state)
- get_radiator_cover_angle()
    - **Returns**: radiator cover angle in radians.

Geometric map

The geometric map is the data object that you must populate with your estimates of surface elevation and the rocks that you detect.

Retrieve a reference to the **GeometricMap** object using the `get_geometric_map()` method of **AutonomousAgent**. The **GeometricMap** object has the following utility methods:

- get_map_array()
  Returns the map array
    - **Returns**: *Numpy* array, containing the geometric map data
- get_map_size()
  Returns the size of the map in meters.
    - **Returns**: *float* - the dimension of the map in meters
- get_cell_size()
  Returns the size of an individual cell.
    - **Returns**: *float* - the dimension of an individual cell in meters
- get_cell_indexes(x, y) Returns the indices of a cell for a given world coordinate
    - **Parameters**:

        - `x` : *float* - x-coordinate
        - `y` . *float* - y-coordinate
    - **Returns**:
        - *tuple* if *ints* `(x_index, y_index)`
- **get_cell_data**(**x_index**, **y_index**)
  Returns the center position and cell size
    - **Parameters**:
        - `x_index` : *int* - x-index
        - `y_index` : *int* - y-index
    - **Returns**: *tuple* `((x, y), size)`
- **get_cell_height**(**x_index**, **y_index**)
  Returns the cell height or elevation
    - **Parameters**:
        - `x_index` : *int* - x-index
        - `y_index` : *int* - y-index
    - **Returns**: *float*, `inf` if you have not mapped the cell or `None` if the indices are not valid.
- **get_height**(**x**, **y**)
  Returns the cell height
    - **Parameters**:
        - `x` : *float* - x-coordinate
        - `y` : *float* - y-coordinate
    - **Returns**: *float*, `inf` if you have not mapped the cell or `None` if the coordinates are not valid.
- **get_cell_rock**(**x_index**, **y_index**)
  Returns a value indicating the presence of a rock in the cell
    - **Parameters**:
        - `x_index` : *int* - x-index
        - `y_index` : *int* - y-index

- **Returns**: *bool* - `True` (rock) or `False` (no rock) or `inf` (unmapped)

- **get_rock**(**x**, **y**) Returns a value indicating the presence or absence of a rock
  - **Parameters**:
    - `x` : *float* - x-coordinate
    - `y` : *float* - y-coordinate
  - **Returns**: *bool* - `True` (rock) or `False` (no rock) or `inf` (unmapped)

- **set_cell_height**(**x_index**, **y_index**, **height**)
  Sets the estimated height of a cell
  - **Parameters**:
    - `x_index` : *int* - x-index
    - `y_index` : *int* - y-index

- **set_height**(**x**, **y**, **height**)
  Sets the estimated height of the cell
  - **Parameters**:
    - `x` : *float* - x-coordinate
    - `y` : *float* - y-coordinate

- **set_cell_rock**(**x_index**, **y_index**, **rock_flag**) Sets the rock flag of the cell
  - **Parameters**:
    - `x_index` : *int* - x-index
    - `y_index` : *int* - y-index
    - `rock_flag` : *bool* - `True` (rock) of `False` (no rock)

- **set_rock**(**x**, **y**, **rock_flag**)
  Sets the rock flag of the cell
  - **Parameters**:
    - `x` : *float* - x-coordinate
    - `y` : *float* - y-coordinate
    - `rock_flag` : *bool* - `True` (rock) of `False` (no rock)

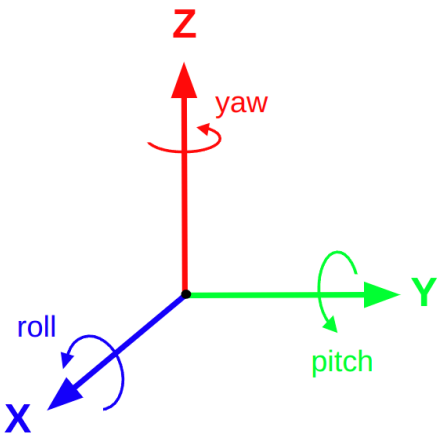
Geometry

Updated - 3rd September 2024

Coordinate system

The lunar simulator utilizes a **right-handed coordinate system**. For rotations, the following definitions apply for an observer located at the origin looking along the positive direction of each respective axis:

- Positive roll defines a clockwise rotation about the X-axis
- Positive pitch defines a clockwise rotation about the Y-axis
- Positive yaw defines a clockwise rotation about the Z-axis



Right-handed coordinate system definition of the lunar simulator.

Geometric map

The geometric map is a coordinate grid defined at mission initialization to assist with storing terrain elevation data during the mapping task and facilitate mission evaluation. The grid is initialized such that the lander is placed at the coordinate origin in X and Y to serve as a visual reference point for the center of the mapping area. Due to the lander's placement on uneven terrain, there is a small offset between the lander's local coordinate origin and the global geometric map coordinate origin.

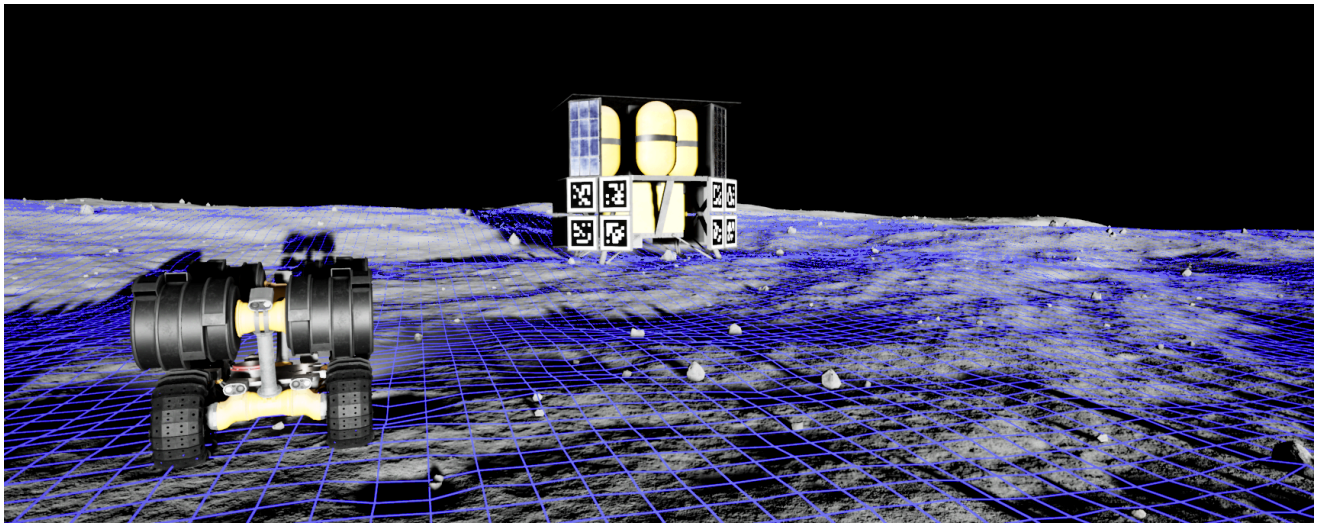


Figure A2.1: Visualization of the geometric map.

Information about the position and orientation of the rover with respect to the global geometric map coordinate system is provided to the agent through the API. The following API methods can be used to establish the relationship of the lander and the rover with the geometric map at the first time-step:

- `get_initial_position()`: returns a transform that defines the position and orientation of the rover in the first time-step of the mission
- `get_initial_lander_position()`: returns a transform that defines the position and orientation of the lander with respect to the rover's local coordinate system in the first time-step of the mission

Your agent should call these functions at the start of the mission to properly align localization logic with the geometric map. These transforms and the information given in the following tables can be used to infer the positions of the fiducials and charging antenna on the lander in the global coordinate system.

Rover and lander geometry

The following tables detail the exact geometrical positions of the functional elements of the lander and rover that are relevant to the localization task. **Each coordinate is defined with respect to the local coordinate frames of the lander and rover respectively, not the global coordinate frame of the map itself**. For convenience, the information given in the following tables is provided in JSON format in the `docs` folder of the simulator package in the file `geometry.json`. The JSON file can be loaded by the built-in Python `json` library as a dictionary.

Rover

The rover's coordinate frame is defined with the same right-handed coordinate convention as the global coordinate frame. The rover's geometry is defined with the rear-front axis of the vehicle pointing along the X-axis. The front of the vehicle is in the positive X direction. The left side of the vehicle is towards the positive Y-axis. The top of the vehicle points towards positive Z.

Cameras

There are 8 cameras on the IPEx rover, arranged in the following configuration:

- forward facing stereo pair
- rear facing stereo pair
- left facing mid-chassis camera
- right facing mid-chassis camera
- camera at the end of the front arm (faces down with the arm parallel to the X-axis)
- camera at the end of the rear arm (faces down with the arm parallel to the X-axis)

The cameras are modelled as **perfect pinhole cameras with square pixels**, there is no lens distortion. Lens flare from the sun is modelled, this should be considered as a potential source of error in segmentation and feature detection. Each camera has the same **field of view of 1.22 radians (70 degrees)**, the resolution is set by the agent upon initialization in the `sensors()` method. **The maximum resolution allowed is 2448 x 2048 pixels**, if a resolution higher than this is requested the resolution will be clipped to the maximum and a warning will be given on the command line.

Figure A2.2 indicates the positions of each camera on the rover and the positive directions of the X and Y axes of its local coordinate system. The image is taken from above the IPEx facing down. The red cross indicates the lateral position of the coordinate origin of the vehicle. The lowest part of the wheels is **0.134 meters** below the origin.

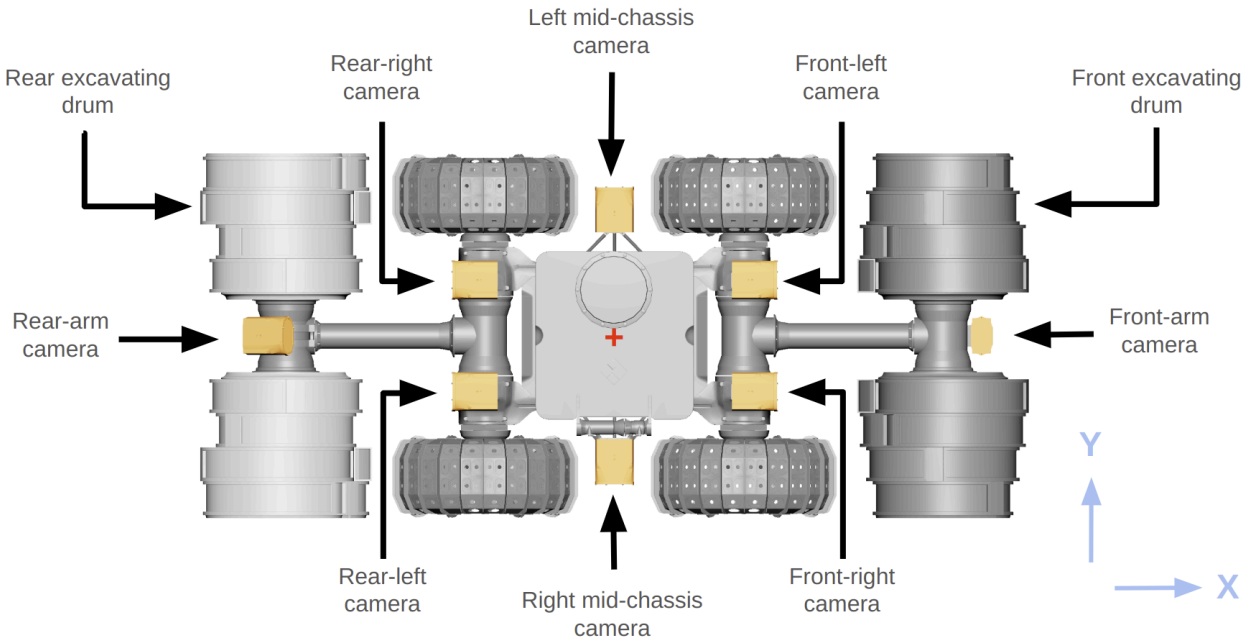


Figure A2.2: Diagram of the IPEx rover taken from above, showing the positions of the 8 cameras and the directions of the local coordinate axes. The red cross indicates the location of the vehicle's coordinate origin.

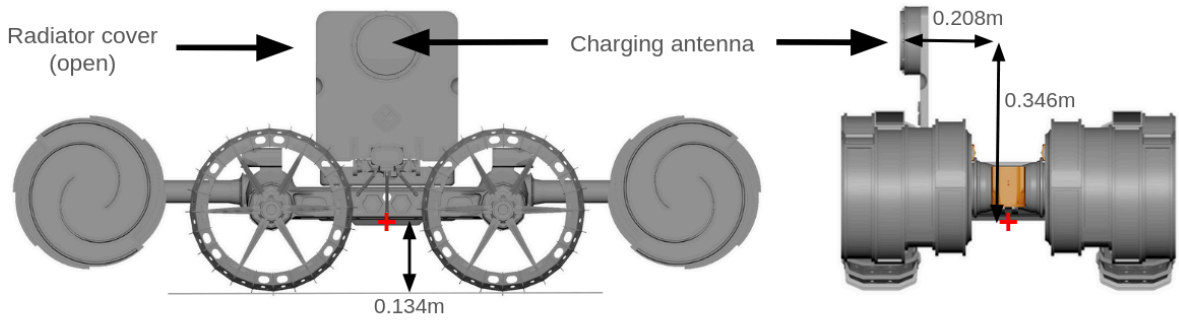


Figure A2.3: Front (right) and side (left) views of the IPEx rover, showing the positions of the origin of the local coordinate system and the charging antenna with the radiator cover open. The red cross indicates the position of the rover's origin of geometry.

The following tables show the geometrical positions of all cameras with respect to the vehicle's origin. The rover's local coordinate system is defined as a right-handed coordinate system with the rear to front axis pointing along the positive X direction and the left hand side of the vehicle towards the positive Y direction, presuming the Z-axis points towards the observer (or the observer is looking down from above).

Forward facing stereo camera pair

The front facing stereo cameras are located just above the front of the rover chassis. The field of view is blocked by the excavating drums when the arms are lowered. **The stereo baseline is 0.162 meters**.

Camera	X (meters)	Y (meters)	Z (meters)	Orientation (unit vector)
Front left	0.28	0.081	0.131	(1, 0, 0)
Front right	0.28	-0.081	0.131	(1, 0, 0)

Rearward facing stereo camera pair

The rear facing stereo cameras are located just above the rear of the rover chassis. The field of view is blocked by the excavating drums when the arms are lowered. **Note that the left and right labels of the stereo pair are assigned on the basis of an observer looking backwards along the X axis from the origin, hence the *left* rear camera is is on the *right* side of the chassis when facing forward. The stereo baseline is 0.162 meters**.

Camera	X (meters)	Y (meters)	Z (meters)	Orientation (unit vector)
Rear left	-0.28	-0.081	0.131	(-1, 0, 0)
Rear right	-0.28	0.081	0.131	(-1, 0, 0)

Mid-chassis cameras

The mid-chassis cameras are located near the center of the length of the vehicle, facing along the positive and negative Y-axis. They are slightly displaced from the X=0 point since each camera is not at the center of its respective enclosure. Each camera shares its enclosure with an LED light. These cameras are particularly useful for aligning with the charging antenna.

Camera	X (meters)	Y (meters)	Z (meters)	Orientation (unit vector)
Left	0.015	0.252	0.132	(0, 1, 0)
Right	-0.015	-0.252	0.132	(0, -1, 0)

Excavation arm cameras

The excavation arm cameras are located at the end of each excavating arm. The following table gives the location of the rotation centers of the arms.

Rotation center	X (meters)	Y (meters)	Z (meters)
Front arm rotation center	0.222	0	0.061
Rear arm rotation center	-0.223	0	0.061

The following tables give the **offset of the cameras from their respective rotation centers in the resting position of the vehicle i.e. with both arms at 0 radians/degrees rotation**.

The front camera points directly downwards in the negative Z direction when the arm is at 0 radians of rotation. When the front arm is raised by a full 1.57 radians (90 degrees), the forward camera points straight forward.

Camera	X (meters)	Y (meters)	Z (meters)	Orientation (unit vector)
Front arm	0.414	0.015	-0.038	(0, 0, -1)

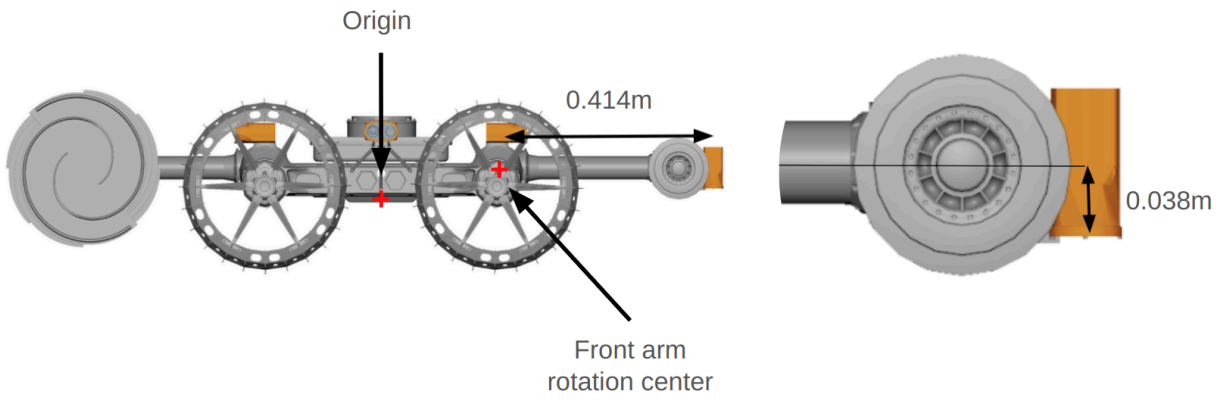


Figure A2.4: Diagram of the camera mounted to the front arm of the rover.

The rear arm camera faces forwards towards the positive X axis with the rear arm at 0 radians of rotation. The angle between the camera axis and the arm is 0.52 radians (30 degrees). If the rear arm is raised to 0.52 radians, the rear arm camera will point straight forwards with a view of the top of the rover chassis. In such position this camera is useful to inspect alignment with the charging antennas.

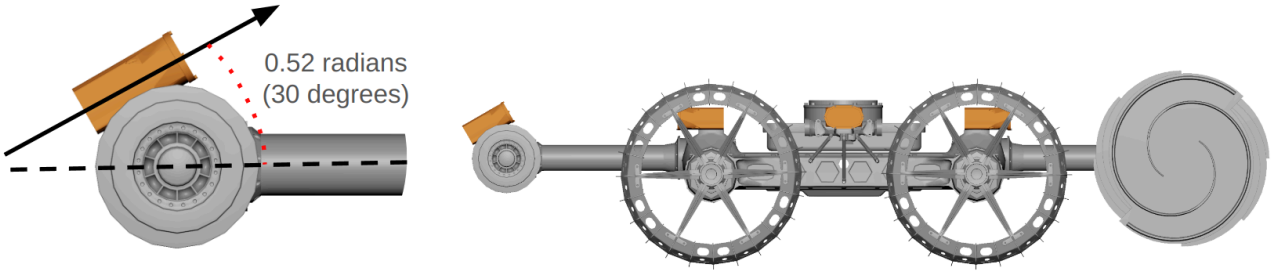


Figure A2.5: Diagram of the camera mounted to the rear arm of the rover.

Camera	X (meters)	Y (meters)	Z (meters)	Orientation (unit vector)
Rear arm	-0.339	0.017	0.083	(0.866, 0, 0.5)

Rover LED lights

Each camera is coupled with an LED light in the same housing to illuminate dark areas of the map in front of the camera and to highlight the retroreflective material of the fiducials. Each light is roughly 0.032 meters away from its respective camera.

Light	X (meters)	Y (meters)	Z (meters)	Orientation (unit vector)
Front left	0.280	0.115	0.131	(1, 0, 0)
Front right	0.280	-0.115	0.131	(1, 0, 0)
Rear left	-0.280	-0.115	0.131	(-1, 0, 0)
Rear right	-0.280	0.115	0.131	(-1, 0, 0)
Left	-0.015	0.252	0.132	(0, 1, 0)
Right	0.015	-0.252	0.132	(0, -1, 0)
Front arm	0.414	-0.015	-0.038	(0, 0, -1)
Rear arm	-0.339	-0.015	0.083	(0.866, 0, 0.5)

Rover charge receiving antenna

The following table represents the geometrical offset of the center of the receiving charging antenna on the rover with the radiator cover raised.

	X (meters)	Y (meters)	Z (meters)	Orientation (unit vector)
Receiving antenna	0	-0.208	0.346	(0, -1, 0)

The charge receiving antenna must be positioned no more than 10cm away in X or Y from the center of the charge source antenna on the lander and the angle between their respective axes must be no more than 0.52 radians (30 degrees). The figure A2.6 shows the angle θ, which must be below this threshold for charging to proceed:

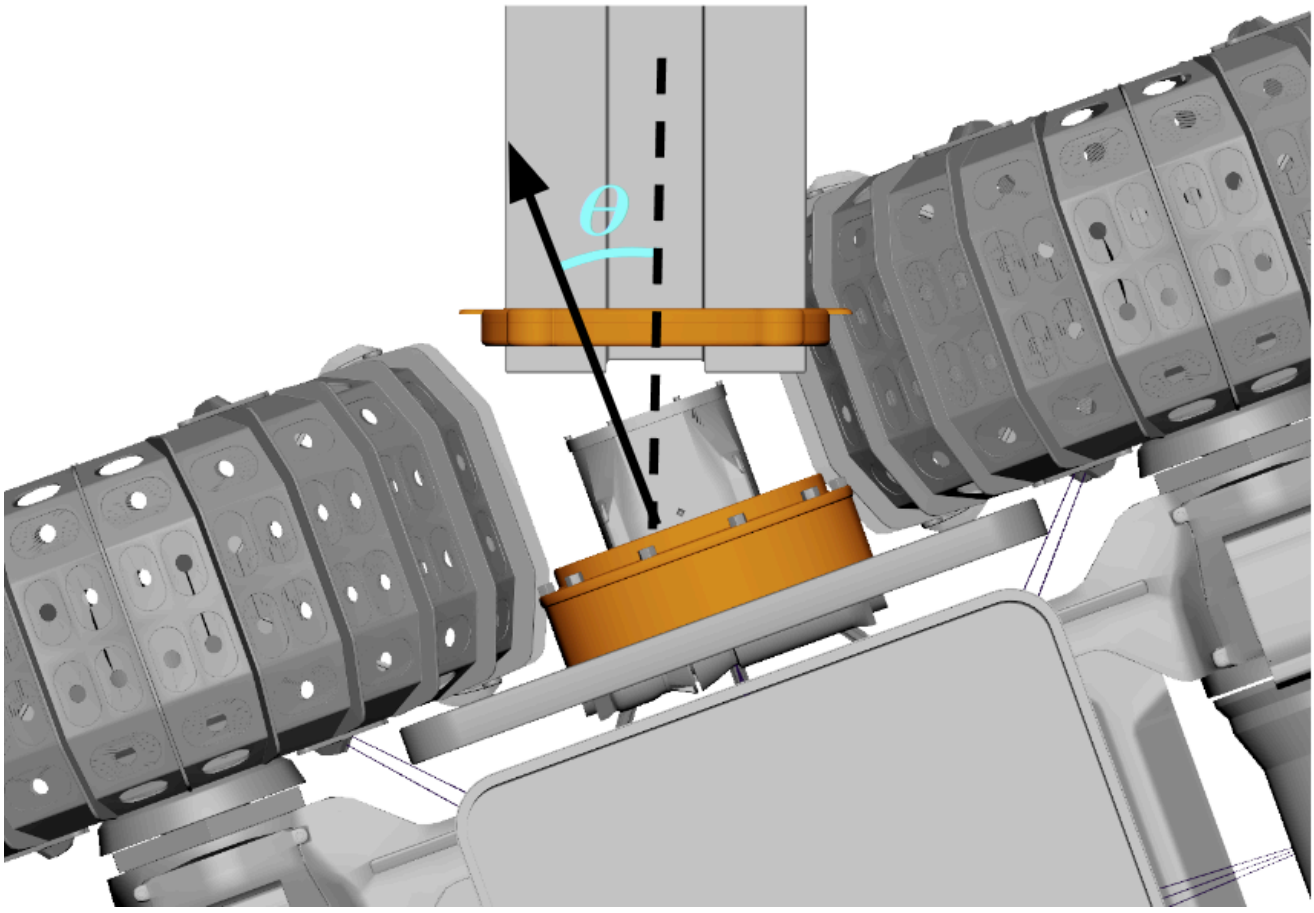


Figure A2.6: Diagram of the angle between the axes of the charging antennas.

Once the chargers have been aligned correctly, the power will be returned to full and the mission time will be advanced. Monitor the charge state of the battery through the API to determine if charging is successful.

Rover inertial measurement unit (IMU)

The IMU is located at the vehicle's coordinate origin.

	X (meters)	Y (meters)	Z (meters)	Orientation (unit vector)
IMU	0.0	0.0	0.0	(1, 0, 0)

Rover wheels

The following table contains the center location of each of the rover's 4 wheels with respect to the vehicle's coordinate origin. Each wheel has a diameter of 0.32 meters and a width of 0.127 meters.

Wheel	X (meters)	Y (meters)	Z (meters)
Front left	0.222	0.203	0.041
Front right	0.222	-0.203	0.041
Rear left	-0.222	0.203	0.041
Rear right	-0.222	-0.203	0.041

Lander

The lander's local coordinate system is defined with the same **right-handed coordinate** convention as the global coordinate frame. The charging antenna extends out of the lander in the positive Y direction. The lander is located at the center of the mapping area. Its origin may be slightly displaced from the exact coordinate origin of the map since it is initialized on uneven terrain. There may also be a rotation due to rotational shifts during initialization. **Therefore the coordinate axes of the lander cannot be expected to exactly align with the coordinate axes of the map. The exact position of the lander can be inferred using information given through the API at mission initialization.**.

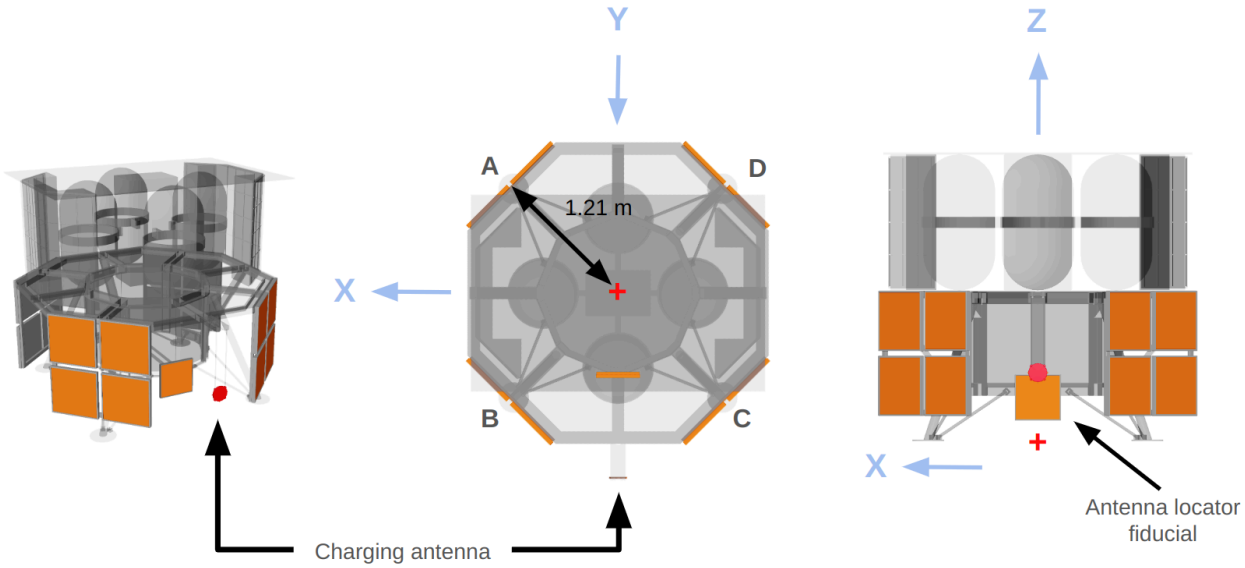


Figure A2.7: Lander geometry from perspective view (left) top-down (middle) and side view (right). Showing the position of the charging antenna and fiducials in its local coordinate system. The red cross indicates the lander's coordinate origin.

Fiducials

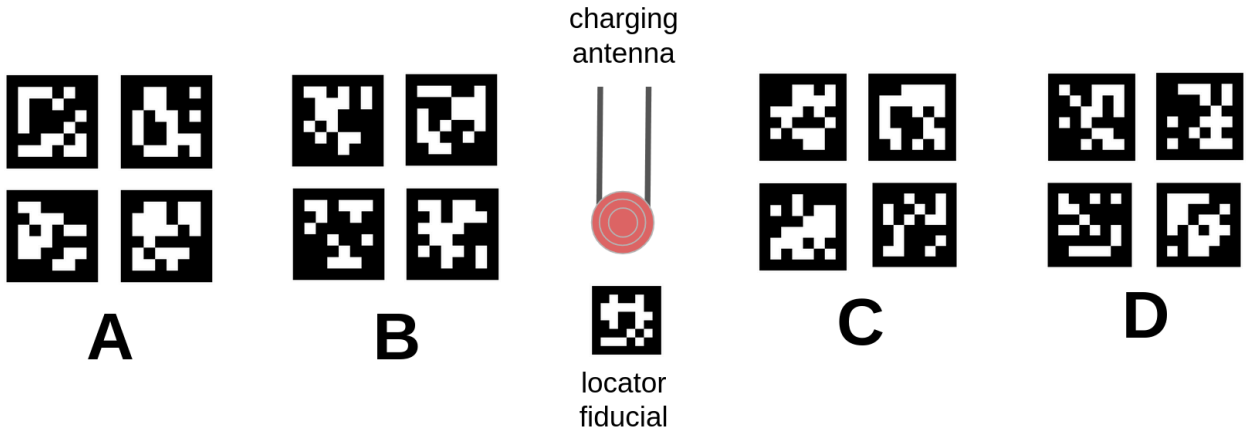


Figure A2.8: Diagram of the AprilTag fiducial groups with the relative position of the charging antenna.

The following tables demonstrate the positions of the fiducial markers (April tags) that are attached to the lander for localization. Each coordinate represents the center of the respective AprilTag. **Each AprilTag is square and the black area of the tag has a dimension of 0.339 meters**. The lateral distance between the center of each fiducial group and the lander's origin is 1.12 meters. All fiducials are drawn from the **36h11 family** of AprilTags, their respective tag ID is given in the tables.

Fiducial group A	X (meters)	Y (meters)	Z (meters)	Tag ID
Top left	0.691	-1.033	0.894	243
Top right	1.033	-0.691	0.894	71
Lower left	0.691	-1.033	0.412	462
Lower right	1.033	-0.691	0.412	37

Fiducial group B	X (meters)	Y (meters)	Z (meters)	Tag ID
Top left	1.033	0.691	0.894	0
Top right	0.691	1.033	0.894	3
Lower left	1.033	0.691	0.412	2
Lower right	0.691	1.033	0.412	1

Fiducial group C	X (meters)	Y (meters)	Z (meters)	Tag ID
Top left	-0.691	1.033	0.894	10
Top right	-1.033	0.691	0.894	11
Lower left	-0.691	1.033	0.412	8
Lower right	-1.033	0.691	0.412	9

Fiducial group D	X (meters)	Y (meters)	Z (meters)	Tag ID
Top left	-1.033	-0.691	0.894	464
Top right	-0.691	-1.033	0.894	459
Lower left	-1.033	-0.691	0.412	258
Lower right	-0.691	-1.033	0.412	5

Lander charging source antenna

The following table gives the coordinates of the charging source antenna of the lander.

	X (meters)	Y (meters)	Z (meters)	Orientation (unit vector)
Charging antenna	0	1.452	0.509	(0, 1, 0)

Lander charging antenna locator fiducial

The following table gives the coordinates of the locator fiducial for the charging antenna of the lander. **The black area of the AprilTag is square and has a dimension of 0.253 meters**. The charging antenna is 0.182 meters higher in Z and 0.79 meters distance in Y from the fiducial.

	X (meters)	Y (meters)	Z (meters)	Tag ID
Antenna locator fiducial	0	0.662	0.325	69

IPEx rover technical details

Updated - 3rd September 2024

IPEx power model

The following table details the power consumption characteristics of the IPEx rover components. These values are estimated and may vary depending on specific operating conditions. They serve as an approximate guide to calculate estimated power consumption. The units are given in Watts (W) and Watt hours (Wh).

Item	Watt hours (Wh)	Note
Total power	283	
	Load in Watts (W)	
Camera	3	For one active camera
Light	9.8	For one active light at full power (linearly scales with intensity)
Wheels	40-60	For all wheels driving straight with linear speed of 0.3 m/s
Excavator arms	200-300	For a single arm while moving (0 while static)
Arm brake	24	For a single arm while moving, 0 while arm is static*
Radiator cover	-	Negligible
Computation load	8	Constant during mission
Non-compute load	2	Constant during mission
IMU	0.15	Constant during mission

- * the arm brake is clamped onto the arm with strong springs to secure the arm in a fixed orientation without using power. To free the arm to move, the motor must work against the force of the spring.

IPEx rover maximum speeds

	Speed	Note
Max linear speed	0.48 m/s	With zero angular speed
Max angular speed	4.13 rad/s	With zero linear speed

Leaderboard evaluator script

Updated - 3rd September 2024

For convenience, we provide the `RunLeaderboard.sh` shell script which wraps the `leaderboard_evaluator.py` Python script, making it easier to configure the parameters. We recommend using this shell script to run the Leaderboard. If you want more control, you may want to run `leaderboard_evaluator.py` script directly with Python. It has a number of parameters explained in the following:

General parameters

- `--host`: host IP of the machine running the lunar simulator (not the machine running the Leaderboard, if different). This can be set to `localhost` if you are running the lunar simulator locally
- `--port`: the port which the lunar simulator server uses for communication with the client, by default this is set to 2000
- `--seed`: random seed for the simulator
- `--timeout`: timeout for the simulator client, i.e. how long the client will wait for a response from the simulator
- `--qualifier`: runs the mission in qualifier mode
- `--evaluation`: runs the mission in evaluation mode
- `--development`: runs the mission in development mode, skips mission statistics calculation

Simulation parameters

- `--missions`: location of the XML file defining the mission(s) chosen to execute
- `--missions-subset`: IDs of the missions you want to run not have to be numbers
- `--repetitions`: how many times to repeat the chosen mission(s)
- `--resume`: resume simulation from the last successfully completed mission

Agent related options

- `-a`, `--agent`: path to the Python file in which your agent is implemented
- `--agent-config`: path to the agent config file

Output options

- `--checkpoint`: Location of the directory where output files from the Leaderboard will be stored (e.g. results files, record logs)
- `--record`: activate the recorder function, to record every movement of your agent in the simulation. You can play the log back to visualize your agent's behavior
- `--record-control`: output JSON telemetry record file, `--record-control True` to output the file, exclude the argument or leave it empty otherwise

Missions file details

This file is an XML file containing details of the missions you want the Leaderboard to execute. Each mission definition specifies a map and a preset.

There are two maps provided with the package:

- Moon_Map_01
- Moon_Map_02

There are also 12 presets to choose from, each preset corresponds to a specific map:

- **Moon_Map_01**: 0-10
- **Moon_Map_01**: 11,13

Please avoid using presets with the wrong map, which will cause an error.

The missions file should look like the following:

```
<missions>
    <mission id="0" map="Moon_Map_01" preset="0"/>
</missions>
```

The `--missions` parameter provided to the Leaderboard should point to this file. The `missions-subset` parameter defines which of the missions in this file the Leaderboard will execute sequentially. For example `--missions-subset 0-2` will run missions between 0 and 2, `--missions-subset 0,2,3` will run missions 0, 2 and 3. IDs are treated as strings and do not need to be integers.

The `--repetitions` parameter sets how many times each mission is repeated. If the simulation crashes before completing all missions and repetitions, re-launch the Leaderboard with the `--resume` parameter to resume the simulation from the point of the last successfully completed mission.

Use the `--resume` argument if a previous run crashed.

Results files

Once the Leaderboard has executed a mission, it will deposit the results files in the directory provided in the `--checkpoint` parameter. The files will be named according to the following rule:

```
{map}_{id}_rep{rep_id}.txt (or .dat, .json, .log)
```

By default, 5 files will be created, for example, if we run a mission with ID=0, using Moon_Map_01, then the files for the first repetition will be named like so:

- `results.json` -> the metrics defining your agent's mission performance
- `Moon_Map_01_0_rep0.txt` -> the ground truth height map, in text format for easy inspection
- `Moon_Map_01_0_rep0.dat` -> the ground truth height map in Numpy format, to be loaded as an array
- `Moon_Map_01_0_rep0_agent.txt` -> the height map estimated by your agent, in text format for easy inspection
- `Moon_Map_01_0_rep0_agent.dat` -> height map estimated by your agent in Numpy format, to be loaded as an array

Recording

If you use the record parameters, the Leaderboard will produce additional output files in the output directory for inspection and debugging.

If you use the parameter `--record`, an additional output file named `Moon_Map_01_0_rep0.log` will be saved which is a log file of the mission. This file contains all the details of the movement of the rover and can be replayed using the simulator playback functionality. This

allows you to retrospectively examine the behavior of your agent during a mission for debugging or iteration.

If you use the parameter `--record-control`, an additional output file named `Moon_Map_01_0_rep0_agent.json` will be saved which is a JSON file containing the recorded telemetry of the rover during a mission, for retrospective examination.