# PASEOS

*Release v0.2.0*

**Pablo Gómez, Gabriele Meoni, Johan Östman, Vinutha Magal Shre**

**Sep 04, 2023**

# CONTENTS:

CHAPTER

# ONE

# ALL CONTENT

This is the list of all content in *paseos*.

**class** paseos.**ActorBuilder**

Bases: `object`

This class is used to construct actors.

**static add_comm_device**(*actor: BaseActor*, *device_name: str*, *bandwidth_in_kbps: float*)

Creates a communication device.

**Parameters**

- **device_name** (*str*) – device_name of the communication device.
- **bandwidth_in_kbps** (*float*) – device bandwidth in kbps.

**static add_custom_property**(*actor: BaseActor*, *property_name: str*, *initial_value: Any*, *update_function: Callable*)

Adds a custom property to the actor. This e.g. allows tracking any physical the user would like to track.

The update functions needs to take three parameters as input: the actor, the time to advance the state / model and the current_power_consumption_in_W and return the new value of the custom property. The function will be called with (actor,0,0) to check correctness.

**Parameters**

- **actor** (*BaseActor*) – The actor to add the custom property to.
- **property_name** (*str*) – The name of the custom property.
- **initial_value** (*Any*) – The initial value of the custom property.
- **update_function** (*Callable*) – The function to update the custom property.

**static get_actor_scaffold**(*name: str*, *actor_type: object*, *epoch: epoch*)

Initiates an actor with minimal properties.

**Parameters**

- **name** (*str*) – Name of the actor.
- **actor_type** (*object*) – Type of the actor (e.g. SpacecraftActor)
- **epoch** (*pykep.epoch*) – Current local time of the actor.

**Returns**

Created actor

**static set_TLE**(*actor:* SpacecraftActor, *line1: str*, *line2: str*)

Define the orbit of the actor using a TLE. For more information on TLEs see https://en.wikipedia.org/wiki/Two-line_element_set .

TLEs can be obtained from https://www.space-track.org/ or https://celestrak.com/NORAD/elements/

**Parameters**

- **actor** (SpacecraftActor) – Actor to update.
- **line1** (*str*) – First line of the TLE.
- **line2** (*str*) – Second line of the TLE.

**Raises**

**RuntimeError** – If the TLE could not be read.

**static set_central_body**(*actor: ~paseos.actors.spacecraft_actor.SpacecraftActor*, *pykep_planet: <module 'pykep.planet' from '/home/docs/checkouts/readthedocs.org/user_builds/paseos/conda/latest/lib/python3.11/site-packages/pykep/planet/__init__.py'>*, *mesh: tuple = None*, *radius: float = None*, *rotation_declination: float = None*, *rotation_right_ascension: float = None*, *rotation_period: float = None*)

Define the central body of the actor. This is the body the actor is orbiting around.

If a mesh is provided, it will be used to compute visibility and eclipse checks. Otherwise, a sphere with the provided radius will be used. One of the two has to be provided.

Note the specification here will not affect the actor orbit. For that, use set_orbit, set_TLE or set_custom_orbit.

**Parameters**

- **actor** (SpacecraftActor) – Actor to update.
- **pykep_planet** (*pk.planet*) – Central body as a pykep planet in heliocentric frame.
- **mesh** (*tuple*) – A tuple of vertices and triangles defining a mesh.
- **radius** (*float*) – Radius of the central body in meters. Only used if no mesh is provided.
- **rotation_declination** (*float*) – Declination of the rotation axis in degrees in the
- **0.** (*the central body's inertial frame. Rotation at current actor local time is presumed to be*) –
- **rotation_right_ascension** (*float*) – Right ascension of the rotation axis in degrees in
- **0.** –
- **rotation_period** (*float*) – Rotation period in seconds. Rotation at current actor local time is presumed to be 0.

**static set_custom_orbit**(*actor:* SpacecraftActor, *propagator_func: Callable*, *epoch: epoch*)

Define the orbit of the actor using a custom propagator function. The custom function has to return position and velocity in meters and meters per second respectively. The function will be called with the current epoch as the only parameter.

**Parameters**

- **actor** (SpacecraftActor) – Actor to update.
- **propagator_func** (*Callable*) – Function to propagate the orbit.

- **epoch** (*pk.epoch*) – Current epoch.

**static set_ground_station_location**(*actor: GroundstationActor*, *latitude: float*, *longitude: float*, *elevation: float = 0*, *minimum_altitude_angle: float = 30*)

Define the position of a ground station actor.

**Parameters**

- **actor** (*GroundstationActor*) – Actor to update.
- **latitude** (*float*) – Latitude of the ground station in degrees.
- **longitude** (*float*) – Longitude of the ground station in degrees.
- **elevation** (*float*) – A distance specifying elevation above (positive)
- **below** (*or*) –
- **0.** (*ellipsoid specified by the WSG84 model in meters. Defaults to*) –
- **minimum_altitude_angle** (*float*) – Minimum angle above the horizon that
- **with.** (*this station can communicate*) –

**static set_orbit**(*actor: ~paseos.actors.spacecraft_actor.SpacecraftActor*, *position*, *velocity*, *epoch: ~pykep.core.core.epoch*, *central_body: <module 'pykep.planet' from '/home/docs/checkouts/readthedocs.org/user_builds/paseos/conda/latest/lib/python3.11/site-packages/pykep/planet/__init__.py'>*)

Define the orbit of the actor

**Parameters**

- **actor** (*BaseActor*) – The actor to define on
- **position** (*list of floats*) – [x,y,z].
- **velocity** (*list of floats*) – [vx,vy,vz].
- **epoch** (*pk.epoch*) – Time of position / velocity.
- **central_body** (*pk.planet*) – Central body around which the actor is orbiting as a pykep planet.

**static set_position**(*actor: BaseActor*, *position: list*)

Sets the actors position. Use this if you do *not* want the actor to have a keplerian orbit around a central body.

**Parameters**

- **actor** (*BaseActor*) – Actor set the position on.
- **position** (*list*) – [x,y,z] position for SpacecraftActor.

**static set_power_devices**(*actor: SpacecraftActor*, *battery_level_in_Ws: float*, *max_battery_level_in_Ws: float*, *charging_rate_in_W: float*, *power_device_type: PowerDeviceType = PowerDeviceType.SolarPanel*)

Add a power device (battery + some charging mechanism (e.g. solar power)) to the actor. This will allow constraints related to power consumption.

**Parameters**

- **actor** (*SpacecraftActor*) – The actor to add to.
- **battery_level_in_Ws** (*float*) – Current battery level in Watt seconds / Joule
- **max_battery_level_in_Ws** (*float*) – Maximum battery level in Watt seconds / Joule

- **charging_rate_in_W** (*float*) – Charging rate of the battery in Watt
- **power_device_type** (PowerDeviceType) – Type of power device.
- **SolarPanel.** (*Either "SolarPanel" or "RTG" at the moment. Defaults to*) –

**static set_radiation_model**(*actor: SpacecraftActor*, *data_corruption_events_per_s: float*, *restart_events_per_s: float*, *failure_events_per_s: float*)

Enables the radiation model allowing data corruption, activities being interrupted by restarts and potentially critical device failures. Set any of the passed rates to 0 to disable that particular model.

**Parameters**

- **actor** (SpacecraftActor) – The actor to add to.
- **data_corruption_events_per_s** (*float*) – Single bit of data being corrupted, events per second,
- **Upset** (*i.e. a Single Event*) –
- **restart_events_per_s** (*float*) – Device restart being triggered, events per second.
- **failure_events_per_s** (*float*) – Complete device failure, events per second, i.e. a Single Event Latch-Up (SEL).

**static set_thermal_model**(*actor: SpacecraftActor*, *actor_mass: float*, *actor_initial_temperature_in_K: float*, *actor_sun_absorptance: float*, *actor_infrared_absorptance: float*, *actor_sun_facing_area: float*, *actor_central_body_facing_area: float*, *actor_emissive_area: float*, *actor_thermal_capacity: float*, *body_solar_irradiance: float = 1360*, *body_surface_temperature_in_K: float = 288*, *body_emissivity: float = 0.6*, *body_reflectance: float = 0.3*, *power_consumption_to_heat_ratio: float = 0.5*)

Add a thermal model to the actor to model temperature based on heat flux from sun, central body albedo, central body IR, actor IR emission and due to actor activities. For the moment, it is a slightly simplified version of the single node model from "Spacecraft Thermal Control" by Prof. Isidoro Martínez available at http://imartinez.etsiae.upm.es/~isidoro/tc3/Spacecraft%20Thermal%20Modelling%20and%20Testing.pdf

**Parameters**

- **actor** (SpacecraftActor) – Actor to model.
- **actor_mass** (*float*) – Actor's mass in kg.
- **actor_initial_temperature_in_K** (*float*) – Actor's initial temperature in K.
- **actor_sun_absorptance** (*float*) – Actor's absorptance ([0,1]) of solar light
- **actor_infrared_absorptance** (*float*) – Actor's absportance ([0,1]) of IR.
- **actor_sun_facing_area** (*float*) – Actor area facing the sun in m^2.
- **actor_central_body_facing_area** (*float*) – Actor area facing central body in m^2.
- **actor_emissive_area** (*float*) – Actor area emitting (radiating) heat.
- **actor_thermal_capacity** (*float*) – Actor's thermal capacity in J / (kg * K).
- **body_solar_irradiance** (*float, optional*) – Irradiance from the sun in W. Defaults to 1360.
- **body_surface_temperature_in_K** (*float, optional*) – Central body surface temperature. Defaults to 288.

- **body_emissivity** (*float, optional*) – Centrla body emissivity [0,1] in IR. Defaults to 0.6.
- **body_reflectance** (*float, optional*) – Central body reflectance of sun light. Defaults to 0.3.
- **power_consumption_to_heat_ratio** (*float, optional*) – Conversion ratio for activities.
- **0.5.** (*0 leads to know heat-up due to activity. Defaults to*) –

**class** paseos.**BaseActor**(*name: str*, *epoch: epoch*)

Bases: ABC

This (abstract) class is the baseline implementation of an actor (e.g. spacecraft, ground station) in the simulation. The abstraction allows some actors to have e.g. limited power (spacecraft) and others not to.

**property central_body:** *CentralBody*

Returns the central body this actor is orbiting.

**charge**(*duration_in_s: float*)

Charges the actor from now for that period. Note that it is only verified the actor is neither at start nor end of the period in eclipse, thus short periods are preferable.

**Parameters**
**duration_in_s** (*float*) – How long the activity is performed in seconds

**property communication_devices: DotMap**

Returns the communications devices.

**Returns**
Dictionary (DotMap) of communication devices.

**Return type**
DotMap

**property current_activity: str**

Returns the name of the activity the actor is currently performing.

**Returns**
Activity name. None if no activity being performed.

**Return type**
str

**property custom_properties**

Returns a dictionary of custom properties for this actor.

**discharge**(*consumption_rate_in_W: float*, *duration_in_s: float*)

Discharge battery depending on power consumption. Not implemented by default.

**Parameters**

- **consumption_rate_in_W** (*float*) – Consumption rate of the activity in Watt
- **duration_in_s** (*float*) – How long the activity is performed in seconds

**get_altitude**(*t0: epoch = None*) → float

Returns altitude above [0,0,0]. Will only compute if not computed for this timestep.

**Parameters**
**t0** (*pk.epoch*) – Epoch to get altitude at. Defaults to local time.

**Returns**
Altitude in meters.

**Return type**
float

`get_custom_property`(*property_name: str*) → Any

Returns the value of the specified custom property.

**Parameters**
**property_name** (*str*) – The name of the custom property.

**Returns**
The value of the custom property.

**Return type**
Any

`get_custom_property_update_function`(*property_name: str*) → Callable

Returns the update function for the specified custom property.

**Parameters**
**property_name** (*str*) – The name of the custom property.

**Returns**
The update function for the custom property.

**Return type**
Callable

`get_position`(*epoch: epoch*)

Compute the position of this actor at a specific time. Requires orbital parameters or position set.

**Parameters**
**epoch** (*pk.epoch*) – Time as pykep epoch

**Returns**
[x,y,z] in meters

**Return type**
np.array

`get_position_velocity`(*epoch: epoch*)

Compute the position / velocity of this actor at a specific time. Requires orbital parameters set.

**Parameters**
**epoch** (*pk.epoch*) – Time as pykep epoch.

**Returns**
[x,y,z] in meters

**Return type**
np.array

`property has_central_body: bool`

Returns true if actor has a central body, else false.

**Returns**
bool indicating presence.

**Return type**
bool

property has_power_model: bool

Returns true if actor's battery is modeled, else false.

**Returns**
bool indicating presence.

**Return type**
bool

property has_radiation_model: bool

Returns true if actor's temperature is modeled, else false.

**Returns**
bool indicating presence.

**Return type**
bool

property has_thermal_model: bool

Returns true if actor's temperature is modeled, else false.

**Returns**
bool indicating presence.

**Return type**
bool

is_in_eclipse(*t: epoch = None*)

Checks if the actors is in eclipse at the specified time.

**Parameters**
**t** (*pk.epoch, optional*) – Time to check, if None will use current local actor time.

is_in_line_of_sight(*other_actor:* BaseActor, *epoch: epoch*, *minimum_altitude_angle: float = None*, *plot=False*)

Determines whether a position is in line of sight of this actor

**Parameters**

- **other_actor** (BaseActor) – The actor to check line of sight with
- **epoch** (*pk.epoch*) – Epoch at which to check the line of sight
- **minimum_altitude_angle** (*float*) – The altitude angle (in degree) at which the actor has
- **90.** (*to be in relation to the ground station position to be visible. It has to be between 0 and*) –
- **station.** (*Only relevant if one of the actors is a ground*) –
- **plot** (*bool*) – Whether to plot a diagram illustrating the positions.

**Returns**
true if in line-of-sight.

**Return type**
bool

property local_time: epoch

Returns local time of the actor as pykep epoch. Use e.g. epoch.mjd2000 to get time in days.

**Returns**
local time of the actor

**Return type**
pk.epoch

property mass: float

Returns actor's mass in kg.

**Returns**
Mass

**Return type**
float

name = None

set_custom_property(*property_name: str*, *value: Any*) → None

Sets the value of the specified custom property.

**Parameters**

- **property_name** (*str*) – The name of the custom property.
- **value** (*Any*) – The value to set for the custom property.

set_time(*t: epoch*)

Updates the local time of the actor.

**Parameters**
**t** (*pk.epoch*) – Local time to set to.

class paseos.CentralBody(*planet: <module 'pykep.planet' from '/home/docs/checkouts/readthedocs.org/user_builds/paseos/conda/latest/lib/python3.11/site-packages/pykep/planet/__init__.py'>*, *initial_epoch: ~pykep.core.core.epoch*, *mesh: tuple = None*, *encompassing_sphere_radius: float = None*, *rotation_declination: float = None*, *rotation_right_ascension: float = None*, *rotation_period: float = None*)

Bases: object

Class representing a central body. This can be the Earth but also any other user-defined body in the solar system.

blocks_sun(*actor*, *t: epoch*, *plot=False*) → bool

Checks whether the central body blocks the sun for the given actor.

**Parameters**

- **actor** (SpacecraftActor) – The actor to check
- **t** (*pk.epoch*) – Epoch at which to check
- **plot** (*bool*) – Whether to plot a diagram illustrating the positions.

**Returns**
True if the central body blocks the sun

**Return type**
bool

is_between_actors(*actor_1*, *actor_2*, *t: epoch*, *plot=False*) → bool

Checks whether the central body is between the two actors.

**Parameters**

- **actor_1** (SpacecraftActor) – First actor

- **actor_2** (SpacecraftActor) – Second actor
- **t** (*pk.epoch*) – Epoch at which to check
- **plot** (*bool*) – Whether to plot a diagram illustrating the positions.

**Returns**
True if the central body is between the two actors

**Return type**
bool

**is_between_points**(*point_1*, *point_2*, *t: epoch*, *reference_frame:* ReferenceFrame = *ReferenceFrame.CentralBodyInertial*, *plot: bool = False*) → bool

Checks whether the central body is between the two points.

**Parameters**

- **point_1** (*np.array*) – First point
- **point_2** (*np.array*) – Second point
- **t** (*pk.epoch*) – Epoch at which to check
- **reference_frame** (ReferenceFrame, *optional*) – Reference frame of the points.
- **ReferenceFrame.CentralBodyInertial.** (*Defaults to*) –
- **plot** (*bool*) – Whether to plot a diagram illustrating the positions.

**Returns**
True if the central body is between the two points

**Return type**
bool

**property planet**

**class** paseos.**GroundstationActor**(*name: str*, *epoch: epoch*)

Bases: *BaseActor*

This class models a groundstation actor.

**get_position**(*epoch: epoch*)

Compute the position of this ground station at a specific time. Positions are in J2000 geocentric reference frame, same reference frame as for the spacecraft actors. Since the Earth is rotating, ground stations have a non-constant position depending on time.

**Parameters**
**epoch** (*pk.epoch*) – Time as pykep epoch

**Returns**
[x,y,z] in meters

**Return type**
np.array

**class** paseos.**PASEOS**(*local_actor:* BaseActor, *cfg*)

Bases: `object`

This class serves as the main interface with the user.

add_known_actor(*actor:* BaseActor)

Adds an actor to the simulation.

**Parameters**
**actor** (BaseActor) – Actor to add

advance_time(*time_to_advance: float*, *current_power_consumption_in_W: float*, *constraint_function: function = None*)

Advances the simulation by a specified amount of time

**Parameters**

- **time_to_advance** (*float*) – Time to advance in seconds.
- **current_power_consumption_in_W** (*float*) – Current power consumed per second in Watt.
- **constraint_function** (*FunctionType*) – Constraint function which will be evaluated
- **False.** (*every cfg.sim.activity_timestep seconds. Aborts the advancement if*) –

**Returns**
Time remaining to advance (or 0 if done)

**Return type**
float

empty_known_actors()

Clears the list of known actors.

get_cfg() → DotMap

Returns the current cfg of the simulation

**Returns**
cfg

**Return type**
DotMap

property is_running_activity

Allows checking whether there is currently an activity running.

**Returns**
Yes if running an activity.

**Return type**
bool

property known_actor_names: list

Returns names of known actors.

**Returns**
List of names of known actors.

**Return type**
list

property known_actors: dict

Returns known actors.

**Returns**
Dictionary of the known actors.

**Return type**
dict of *BaseActor*

**property local_actor: *BaseActor***

Returns the local actor.

**Returns**
Local actor

**Return type**
*BaseActor*

**property local_time: epoch**

Returns local time of the actor as pykep epoch. Use e.g. epoch.mjd2000 to get time in days.

**Returns**
local time of the actor

**Return type**
pk.epoch

**log_status()**

Updates the status log.

**model_data_corruption(***data_shape: list*, *exposure_period_in_s: float***)**

Computes a boolean mask for each data element that has been corrupted.

**Parameters**

- **data_shape** (*list*) – Shape of the data to corrupt.
- **exposure_period_in_s** (*float*) – Period of radiation exposure.

**Returns**
Boolean mask which is True if an entry was corrupted.

**Return type**
np.array

**property monitor**

Access paseos operations monitor which tracks local actor attributes such as temperature or state of charge.

**Returns**
Monitor object.

**Return type**
OperationsMonitor

**perform_activity(***name: str*, *activity_func_args: list = None*, *termination_func_args: list = None*, *constraint_func_args: list = None***)**

Perform the specified activity. Will advance the simulation if automatic clock is not disabled.

**Parameters**

- **name** (*str*) – Name of the activity
- **power_consumption_in_watt** (*float, optional*) – Power consumption of the
- **None.** (*activity in seconds if not specified. Defaults to*) –
- **duration_in_s** (*float, optional*) – Time to perform this activity. Defaults to 1.0.

**Returns**
Whether the activity was performed successfully.

**Return type**
bool

**register_activity**(*name: str*, *activity_function: coroutine*, *power_consumption_in_watt: float*, *on_termination_function: coroutine = None*, *constraint_function: coroutine = None*)

Registers an activity that can then be performed on the local actor.

**Parameters**

- **name** (*str*) – Name of the activity.
- **activity_function** (*types.CoroutineType*) – Function to execute during the activity.
- **later.** (*Can accept a list of arguments to be specified*) –
- **power_consumption_in_watt** (*float*) – Power consumption of the activity in W (per second).
- **on_termination_function** (*types.CoroutineType*) – Function to execute when the activities stops
- **async.** (*False if they aren't. Needs to be*) –
- **later.** –
- **constraint_function** (*types.CoroutineType*) – Function to evaluate if constraints are still valid.
- **valid** (*Should return True if constraints are*) –
- **async.** –
- **later.** –

**remove_activity**(*name: str*)

Removes a registered activity

**Parameters**
**name** (*str*) – Name of the activity.

**remove_known_actor**(*actor_name: str*)

Remove an actor from the list of known actors.

**Parameters**
**actor_name** (*str*) – name of the actor to remove.

**save_status_log_csv**(*filename*) → None

Saves the status log incl. all kinds of information such as battery charge, running activtiy, etc.

**Parameters**
**filename** (*str*) – File to save the log in.

**property simulation_time: float**

Get the current simulation time of this paseos instance in seconds since start.

**Returns**
Time since start in seconds.

**Return type**
float

**async wait_for_activity**()

This functions allows waiting for the currently running activity to finish.

class paseos.**PlotType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Describes the different plot types 1 - SpacePlot

**SpacePlot = 1**

class paseos.**PowerDeviceType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Describes the different power device types 1 - Solar Panels 2 - Radioisotope Thermoelectric Generator (RTG)

**RTG = 2**

**SolarPanel = 1**

class paseos.**ReferenceFrame**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

Enum for used reference frames.

**CentralBodyInertial = 1**

**Heliocentric = 2**

class paseos.**SpacecraftActor**(*name: str*, *epoch: epoch*)

Bases: *BaseActor*

This class models a spacecraft actor which in addition to pos, velocity also has additional constraints such as power/battery.

**property battery_level_in_Ws**

Get the current battery level.

**Returns**
current battery level in wattseconds.

**Return type**
float

**charge**(*duration_in_s: float*)

Charges the actor from now for that period. Note that it is only verified the actor is neither at start nor end of the period in eclipse, thus short periods are preferable.

**Parameters**
**duration_in_s** (*float*) – How long the activity is performed in seconds

**property charging_rate_in_W**

Get the current charging rate.

**Returns**
current charging rate in W.

**Return type**
float

**discharge**(*consumption_rate_in_W: float*, *duration_in_s: float*)

Discharge battery depending on power consumption.

**Parameters**

- **consumption_rate_in_W** (*float*) – Consumption rate of the activity in Watt
- **duration_in_s** (*float*) – How long the activity is performed in seconds

**property is_dead: bool**

Returns whether the device experienced fatal radiation failure.

**Returns**
True if device is dead.

**Return type**
bool

**property mass: float**

Returns actor's mass in kg.

**Returns**
Mass

**Return type**
float

**property power_device_type**

Get the power device type

**Returns**
Type of power device.

**Return type**
*PowerDeviceType*

**set_is_dead**()

Sets this device to "is_dead=True" indicating permanent damage.

**set_was_interrupted**()

Sets this device to "was_interrupted=True" indicating current activities were interrupted.

**property state_of_charge**

Get the current battery level as ratio of maximum.

**Returns**
current battery level ratio in [0,1].

**Return type**
float

**property temperature_in_C: float**

Returns the current temperature of the actor in C.

**Returns**
Actor temperature in Celsius.

**Return type**
float

**property temperature_in_K: float**

Returns the current temperature of the actor in K.

**Returns**
Actor temperature in Kelvin.

**Return type**
float

**property was_interrupted: bool**

Returns whether the actor was interrupted in its activity.

**Returns**
True if device is interrupted.

**Return type**
bool

paseos.**find_next_window**(*local_actor:* BaseActor, *local_actor_communication_link_name: str*, *target_actor:* BaseActor, *search_window_in_s: float*, *t0: epoch*, *search_step_size: float = 10*)

Returns the start time of the next window in the given timespan (if any).

**Parameters**

- **local_actor** (BaseActor) – Actor to find window from.
- **local_actor_communication_link_name** (*str*) – Name of the comm device.
- **target_actor** (BaseActor) – Actor find window with.
- **search_window_in_s** (*float*) – Size of the search window in s.
- **t0** (*pk.epoch*) – Start time of the search.
- **search_step_size** (*float*) – Size of steps in the search. Defaults to 10.

**Returns**
Window start (pk.epoch), window length (float [s]), data (int [b]). None,0,0 if none found.

paseos.**get_communication_window**(*local_actor:* BaseActor, *local_actor_communication_link_name: str*, *target_actor:* BaseActor, *dt: float = 10.0*, *t0: epoch = None*, *data_to_send_in_b: int = 9223372036854775807*, *window_timeout_value_in_s=7200*)

Returning the communication window and the amount of data that can be transmitted from the local to the target actor.

**Parameters**

- **local_actor** (BaseActor) – Local actor.
- **local_actor_communication_link_name** (*str*) – Name of the local_actor's communication link to use.
- **target_actor** (BaseActor) – other actor.
- **dt** (*float*) – Simulation timestep [s]. Defaults to 10.
- **t0** (*pk.epoch*) – Current simulation time. Defaults to local time.
- **data_to_send_in_b** (*int*) – Amount of data to transmit [b]. Defaults to maxint.
- **window_timeout_value_in_s** (*float, optional*) – Timeout for estimating the communication window. Defaults to 7200.0.

**Returns**
Communication window start time. pk.epoch: Communication window end time. int: Data that can be transmitted in the communication window [b].

**Return type**
pk.epoch

paseos.**load_default_cfg**()

Loads the default toml config file from the cfg folder.

paseos.**plot**(*sim: PASEOS*, *plot_type: PlotType*, *filename: str = None*)

Creates the animation object

**Parameters**

- **sim** (PASEOS) – simulation object
- **plot_type** (PlotType) – enum deciding what kind of plot object to be made
- **filename** (*str, optional*) – filename to save the animation to. Defaults to None.

**Raises**
**ValueError** – supplied plot type not supported

**Returns**
Animation object

**Return type**
Animation

paseos.**set_log_level**(*log_level: str*)

Set the log level for the logger.

**Parameters**
**log_level** (*str*) – The log level to set. Options are 'TRACE','DEBUG', 'INFO', 'SUCCESS', 'WARNING', 'ERROR', 'CRITICAL'.

CHAPTER

TWO

# PASEOS - PASEOS SIMULATES THE ENVIRONMENT FOR OPERATING MULTIPLE SPACECRAFT

Disclaimer: This project is currently under development. Use at your own risk.

CHAPTER

THREE

# ABOUT THE PROJECT

PASEOS is a `Python` module that simulates the environment to operate multiple spacecraft. In particular, PASEOS offers the user some utilities to run their own *activities* by taking into account both operational and onboard (e.g. limited-power-budget, radiation, and thermal effects) constraints. PASEOS is designed to be:

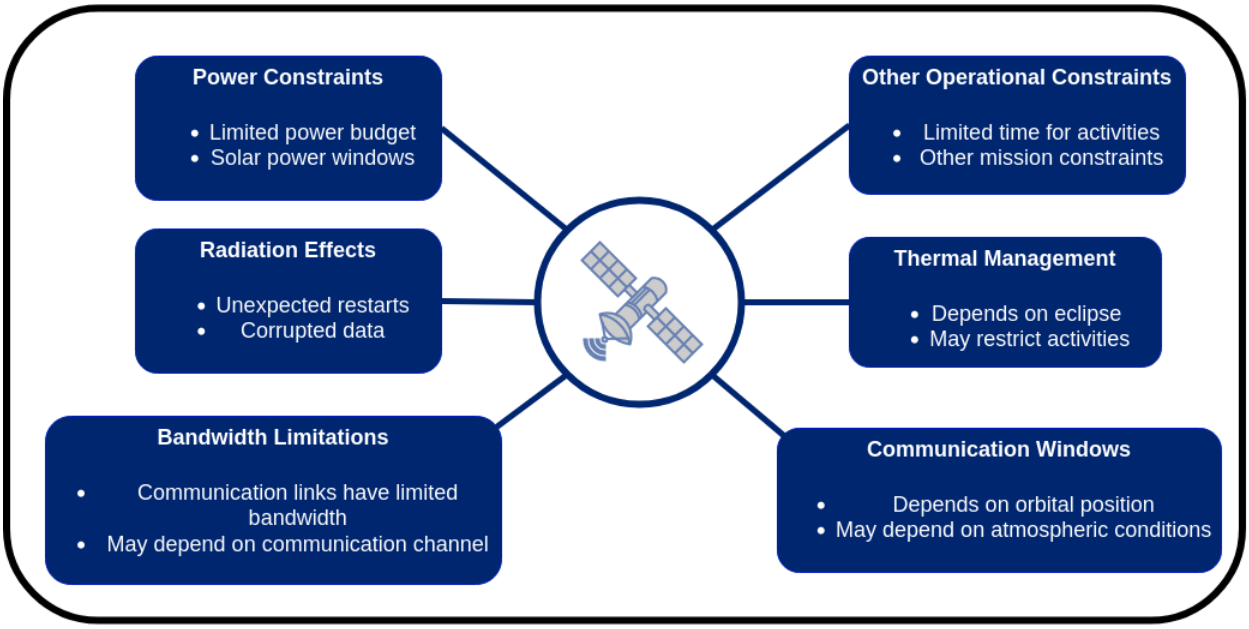
- **open-source**: the source code of PASEOS is available under a GPL license.
- **fully decentralised**: one instance of PASEOS shall be executed in every node, i.e. individual spacecraft (actor), of the emulated spacecraft. Each instance of PASEOS is responsible for handling the user *activities* executed on that node (the local actor) while keeping track of the status of the other nodes. In this way, the design of PASEOS is completely decentralised and independent of the number of nodes of the constellation. Because of that, both single-node and multi-node scenarios are possible.
- **application-agnostic**: each user operation that has to be executed on a node is modelled as an *activity*. The user is only required to provide the code to run and some parameters (e.g., power consumption) for each *activity*. Thus, activities can be any code the user wants to simulate running on a spacecraft and thereby PASEOS is completely application-agnostic. Conceivable applications range from modelling constellations to training machine learning methods.

The project is being developed by $\Phi$-lab@Sweden in the frame of a collaboration between AI Sweden and the European Space Agency to explore distributed edge learning for space applications. For more information on PASEOS and $\Phi$-lab@Sweden, please take a look at the recording of the $\Phi$-lab@Sweden kick-off event.

CHAPTER

FOUR

# PASEOS SPACE ENVIRONMENT SIMULATION




PASEOS allows simulating the effect of onboard and operational constraints on user-registered *activities*. The image above showcases the different phenomena considered (or to be implemented) in PASEOS.

CHAPTER

FIVE

# INSTALLATION

## 5.1 pip / conda

The recommended way to install PASEOS is via conda / mamba using

```
conda install paseos -c conda-forge
```

Alternatively, on Linux you can install via `pip` using

```
pip install paseos
```

The pip version requires `Python 3.8.16` due to pykep's limited support of pip.

## 5.2 Building from source

To build from source, first of all clone the GitHub repository as follows (Git required):

```
git clone https://github.com/aidotse/PASEOS.git
```

To install PASEOS you can use conda as follows:

```
cd PASEOS
conda env create -f environment.yml
```

This will create a new conda environment called `PASEOS` and install the required software packages. To activate the new environment, you can use:

```
conda activate paseos
```

Alternatively, you can install PASEOS by using pip as follows:

```
cd PASEOS
pip install -e .
```

## 5.3 Using Docker

Two Docker images are available:

- paseos: corresponding to the latest release.
- paseos-nightly: based on the latest commit on the branch `main`.

If you want to install PASEOS using Docker, access the desired repository and follow the provided instructions.

CHAPTER

SIX

# EXAMPLES

The next examples will introduce you to the use of PASEOS.

Comprehensive, self-contained examples can also be found in the `examples` folder where you can find an example on:

- Modelling and analysing a large constellation with PASEOS
- Modelling distributed learning on heterogeneous data in a constellation
- Using PASEOS with MPI to run PASEOS on supercomputers
- Using PASEOS to model the task of onboard satellite volcanic eruptions detection
- An example showing how total ionizing dose could be considered using a PASEOS *custom property*

The following are small snippets on specific topics.

## 6.1 Actors

### 6.1.1 Create a PASEOS actor

The code snippet below shows how to create a PASEOS *actor* named **mySat** of type *SpacecraftActor*. pykep is used to define the satellite epoch in format mjd2000 format. *actors* are created by using an `ActorBuilder`. The latter is used to define the *actor* `scaffold` that includes the *actor* minimal properties. In this way, *actors* are built in a modular fashion that enables their use also for non-space applications.

```
import pykep as pk
from paseos import ActorBuilder, SpacecraftActor

# Define an actor of type SpacecraftActor of name mySat
sat_actor = ActorBuilder.get_actor_scaffold(name="mySat",
                                            actor_type=SpacecraftActor,
                                            epoch=pk.epoch(0))
```

### 6.1.2 Local and Known Actors

Once you have instantiated a *PASEOS simulation* you can add other PASEOS *actors* (*Known actors*) to the simulation. You can use this, e.g., to study communications between actors and to automatically monitor communication windows. The next code snippet will add both a *SpacecraftActor* and a *GroundstationActor* (`other_sat`). An orbit is set for `other_sat`, which is placed around Earth at position `(x,y,z)=(-10000,0,0)` and velocity `(vx,vy,vz)=(0, -8000,0)` at epoch `epoch=pk.epoch(0)`. The latter (`grndStation`) will be placed at coordinates `(lat,lon)=(79.002723, 14.642972)` and elevation of 0 m. You cannot add a power device and an orbit to a `GroundstationActor`.

```
import pykep as pk
import paseos
from paseos import ActorBuilder, SpacecraftActor, GroundstationActor
# Define the local actor as a SpacecraftActor of name mySat and its orbit
local_actor = ActorBuilder.get_actor_scaffold(name="mySat",
                                      actor_type=SpacecraftActor,
                                      epoch=pk.epoch(0))

ActorBuilder.set_orbit(
    actor=local_actor,
    position=[10000000, 0, 0],
    velocity=[0, 8000.0, 0],
    epoch=pk.epoch(0),
    central_body=pk.planet.jpl_lp("earth"),  # use Earth from pykep
)

# Initialize PASEOS simulation
sim = paseos.init_sim(local_actor)

# Create another SpacecraftActor
other_spacraft_actor = ActorBuilder.get_actor_scaffold(name="other_sat",
                                     actor_type=SpacecraftActor,
                                     epoch=pk.epoch(0))

# Let's set the orbit of other_spacraft_actor.
ActorBuilder.set_orbit(actor=other_spacraft_actor,
                       position=[-10000000, 0, 0],
                       velocity=[0, -8000.0, 0],
                       epoch=pk.epoch(0), central_body=earth)

#Create GroundstationActor
grndStation = GroundstationActor(name="grndStation", epoch=pk.epoch(0))

#Set the ground station at lat lon 79.002723 / 14.642972
# and its elevation 0m
ActorBuilder.set_ground_station_location(grndStation,
                                         latitude=79.002723,
                                         longitude=14.642972,
                                         elevation=0)

# Adding other_spacraft_actor to PASEOS.
sim.add_known_actor(other_spacraft_actor)

# Adding grndStation to PASEOS.
sim.add_known_actor(grndStation)
```

## 6.2 Physical Models

### 6.2.1 Set an orbit for a PASEOS SpacecraftActor

Once you have defined a *SpacecraftActor*, you can assign a Keplerian orbit or use SGP4 (Earth orbit only).

#### Keplerian Orbit

To this aim, you need to define the central body the *SpacecraftActor* is orbiting around and specify its position and velocity (in the central body's inertial frame) and an epoch. In this case, we will use `Earth` as a central body.

```
import pykep as pk
from paseos import ActorBuilder, SpacecraftActor
# Define an actor of type SpacecraftActor of name mySat
sat_actor = ActorBuilder.get_actor_scaffold(name="mySat",
                                            actor_type=SpacecraftActor,
                                            epoch=pk.epoch(0))

# Define the central body as Earth by using pykep APIs.
earth = pk.planet.jpl_lp("earth")

# Let's set the orbit of sat_actor.
ActorBuilder.set_orbit(actor=sat_actor,
                       position=[10000000, 0, 0],
                       velocity=[0, 8000.0, 0],
                       epoch=pk.epoch(0), central_body=earth)
```

#### SGP4 / Two-line element (TLE)

For using SGP4 / Two-line element (TLE) you need to specify the TLE of the *SpacecraftActor*. In this case, we will use the TLE of the Sentinel-2A satellite from celestrak.

```
from paseos import ActorBuilder, SpacecraftActor
# Define an actor of type SpacecraftActor
sat_actor = ActorBuilder.get_actor_scaffold(name="Sentinel-2A",
                                            actor_type=SpacecraftActor,
                                            epoch=pk.epoch(0))

# Specify your TLE
line1 = "1 40697U 15028A   23188.15862373  .00000171  00000+0  81941-4 0  9994"
line2 = "2 40697  98.5695 262.3977 0001349  91.8221 268.3116 14.30817084419867"

# Set the orbit of the actor
ActorBuilder.set_TLE(sat_actor, line1, line2)
```

#### Custom Propagators

You can define any kind of function you would like to determine actor positions and velocities. This allows integrating more sophisticated propagators such as orekit. A dedicated example on this topic can be found in the `examples` folder.

In short, you need to define a propagator function that returns the position and velocity of the actor at a given time. The function shall take the current epoch as arguments. You can then set the propagator function with

```
import pykep as pk
from paseos import ActorBuilder, SpacecraftActor
# Create a SpacecraftActor
starting_epoch = pk.epoch(42)
my_sat = ActorBuilder.get_actor_scaffold(
    name="my_sat", actor_type=SpacecraftActor, epoch=starting_epoch
)

# Define a custom propagator function that just returns a sinus position
def my_propagator(epoch: pk.epoch):
  position,velocity = your_external_propagator(epoch)
  return position,velocity

# Set the custom propagator
ActorBuilder.set_custom_orbit(my_sat, my_propagator, starting_epoch)
```

#### Accessing the orbit

You can access the orbit of a *SpacecraftActor* with

```
# Position, velocity and altitude can be accessed like this
t0 = pk.epoch("2022-06-16 00:00:00.000") # Define the time (epoch)
print(sat_actor.get_position(t0))
print(sat_actor.get_position_velocity(t0))
print(sat_actor.get_altitude(t0))
```

### 6.2.2 How to add a communication device

The following code snippet shows how to add a communication device to a [SpacecraftActors] (#spacecraftactor). A communication device is needed to model the communication between [SpacecraftActors] (#spacecraftactor) or a *SpacecraftActor* and *GroundstationActor*. Currently, given the maximum transmission data rate of a communication device, PASEOS calculates the maximum data that can be transmitted by multiplying the transmission data rate by the length of the communication window. The latter is calculated by taking the period for which two actors are in line-of-sight into account.

```
import pykep as pk
from paseos import ActorBuilder, SpacecraftActor
# Define an actor of type SpacecraftActor of name mySat
sat_actor = ActorBuilder.get_actor_scaffold(name="mySat",
                                            actor_type=SpacecraftActor,
                                            epoch=pk.epoch(0))
# Add a communication device
ActorBuilder.add_comm_device(actor=sat_actor,
                             # Communication device name
```

(continues on next page)

(continued from previous page)

```
                                device_name="my_communication_device",
                                # Bandwidth in kbps.
                                bandwidth_in_kbps=100000)
```

### 6.2.3 How to add a power device

The following code snippet shows how to add a power device to a *SpacecraftActor*. Moreover, PASEOS assumes that the battery will be charged by solar panels, which will provide energy thanks to the incoming solar radiation when the spacecraft is not eclipsed. Charging and discharging happens automatically during *activities*.

```
import pykep as pk
import paseos
from paseos import ActorBuilder, SpacecraftActor
# Define an actor of type SpacecraftActor of name mySat
sat_actor = ActorBuilder.get_actor_scaffold(name="mySat",
                                            actor_type=SpacecraftActor,
                                            epoch=pk.epoch(0))
# Add a power device
ActorBuilder.set_power_devices(actor=sat_actor,
                               battery_level_in_Ws=100, # current level
                               max_battery_level_in_Ws=2000,
                               charging_rate_in_W=10,
                               power_device_type=paseos.PowerDeviceType.SolarPanel)
```

Alternatively to the default `paseos.PowerDeviceType.SolarPanel` you can also use `paseos.PowerDeviceType.RTG`. The only difference at the moment is that RTGs also charge in eclipse.

Note that at the moment only one power device is supported. Adding another will override the existing one.

You can check the battery's state of charge and level in Ws with:

```
print(my_actor.state_of_charge)
print(my_actor.battery_level_in_Ws)
```

### 6.2.4 Thermal Modelling

To model thermal constraints on spacecraft we utilize a model inspired by the one-node model described in Martínez - Spacecraft Thermal Modelling and Test. Thus, we model the change in temperature as

$$mc , \frac{dT}{dt} = \dot{Q}*{solar} + *dot{Q}*{albedo} + \dot{Q}*{central_body_IR} - *dot{Q}*{dissipated} + \dot{Q}_{activity}.$$

This means your spacecraft will heat up due to being in sunlight, albedo reflections, infrared radiation emitted by the central body as well as due to power consumption of activities. It will cool down due to heat dissipation.

The model is only available for a *SpacecraftActor* and (like all the physical models) only evaluated for the *local actor*.

The following parameters have to be specified for this:

- Spacecraft mass [kg], initial temperature [K], emissive area (for heat dissipation) and thermal capacity [J / (kg * K)]
- Spacecraft absorptance of Sun light, infrared light. [0 to 1]
- Spacecraft area [m^2] facing Sun and central body, respectively

- Solar irradiance in this orbit [W] (defaults to 1360W)
- Central body surface temperature [k] (defaults to 288K)
- Central body emissivity and reflectance [0 to 1] (defaults to 0.6 and 0.3)
- Ratio of power converted to heat (defaults to 0.5)

To use it, simply equip your *SpacecraftActor* with a thermal model with:

```
from paseos import SpacecraftActor, ActorBuilder
my_actor = ActorBuilder.get_actor_scaffold("my_actor", SpacecraftActor, pk.epoch(0))
ActorBuilder.set_thermal_model(
    actor=my_actor,
    actor_mass=50.0, # Setting mass to 50kg
    actor_initial_temperature_in_K=273.15, # Setting initial temperature to 0°C
    actor_sun_absorptance=1.0, # Depending on material, define absorptance
    actor_infrared_absorptance=1.0, # Depending on material, define absorptance
    actor_sun_facing_area=1.0, # Area in m2
    actor_central_body_facing_area=1.0, # Area in m2
    actor_emissive_area=1.0, # Area in m2
    actor_thermal_capacity=1000, # Capacity in J / (kg * K)
    # ... leaving out default valued parameters, see docs for details
)
```

The model is evaluated automatically during *activities*. You can check the spacecraft temperature with:

```
print(my_actor.temperature_in_K)
```

At the moment, only one thermal model per actor is supported. Setting a second will override the old one.

### 6.2.5 Radiation Modelling

PASEOS models three types of radiation effects.

1. Data corruption due to single event upsets which a event rate $r_d$.
2. Unexpected software faults leading to a random interruption of *activities* with a Poisson-distributed event rate $r_i$ per second
3. Device failures with a Poisson-distributed event rate $r_f$ per second, which can be imputed mostly to single event latch-ups

You can add a radiation model affecting the operations of the devices you are interested in with

```
    from paseos import SpacecraftActor, ActorBuilder
    my_actor = ActorBuilder.get_actor_scaffold("my_actor", SpacecraftActor, pk.epoch(0))
    ActorBuilder.set_radiation_model(
        actor=my_actor,
        data_corruption_events_per_s=r_d,
        restart_events_per_s=r_i,
        failure_events_per_s=r_f,
    )
```

You can set any of the event rates to 0 to disable that part. Only *SpacecraftActors* support radiation models. You can find out if your actor has failed with

```
my_actor.is_dead
```

Interrupted *activities* will return as if a *constraint function* was no longer satisfied.

To get a binary mask to model data corruption on the *local actor* you can call

```
mask = paseos_instance.model_data_corruption(data_shape=your_data_shape,
                                             exposure_time_in_s=your_time)
```

### 6.2.6 Custom Modelling

Beyond the default supported physical quantities (power, thermal, etc.) it possible to model any type of parameter by using custom properties. These are defined by a name, an update function and an initial value. The initial value is used to initialize the property. As for the other physical models, you can specify an update rate via the `cfg.sim.dt` *cfg parameter*.

Custom properties are automatically logged in the *operations monitor*. Below is a simple example tracking actor altitude.

```
import pykep as pk
from paseos import ActorBuilder, SpacecraftActor

# Define the local actor as a SpacecraftActor of name mySat and some orbit
local_actor = ActorBuilder.get_actor_scaffold(
    name="mySat", actor_type=SpacecraftActor, epoch=pk.epoch(0)
)

ActorBuilder.set_orbit(
    actor=local_actor,
    position=[10000000, 0, 0],
    velocity=[0, 8000.0, 0],
    epoch=pk.epoch(0),
    central_body=pk.planet.jpl_lp("earth"),  # use Earth from pykep
)


# Define the update function for the custom property
# PASEOS will always pass you the actor, the time step and the current power consumption
# The function shall return the new value of the custom property
def update_function(actor, dt, power_consumption):
    return actor.get_altitude()  # get current altitude


# Add the custom property to the actor, defining name, update fn and initial value
ActorBuilder.add_custom_property(
    actor=local_actor,
    property_name="altitude",
    update_function=update_function,
    initial_value=local_actor.get_altitude(),
)

# One can easily access the property at any point with
print(local_actor.get_custom_property("altitude"))
```

### 6.2.7 Custom Central Bodies

In most examples here you will see Earth via the pykep API being used as a spherical, central body for Keplerian orbits. For Keplerian orbits around spherical bodies, you can simply use pykep with an type of pykep planet just as the above examples used Earth. E.g.

```
import pykep as pk
from paseos import ActorBuilder, SpacecraftActor
# Define an actor of type SpacecraftActor of name mySat
sat_actor = ActorBuilder.get_actor_scaffold(name="mySat",
                                            actor_type=SpacecraftActor,
                                            epoch=pk.epoch(0))

# Define the central body as Mars by using pykep APIs.
mars = pk.planet.jpl_lp("mars")

# Let's set the orbit of sat_actor.
ActorBuilder.set_orbit(actor=sat_actor,
                       position=[10000000, 1, 1],
                       velocity=[1, 1000.0, 1],
                       epoch=pk.epoch(0),
                       central_body=mars)
```

However, you can also use any other central body defined via a mesh. This is especially useful in conjunction with *custom propagators*. To use a custom central body, you need to define a mesh and add it to the simulation configuration. The following example shows how to do this for the comet 67P/Churyumov–Gerasimenko.

We assume `polyhedral_propagator` to be a custom propagator as explained in *Custom Propagators*.

To correctly compute eclipses, we also need to know the orbit of the custom central body around the Sun. In this case we use the orbital elements one can find online for 67P/Churyumov–Gerasimenko.

```
import pykep as pk
from paseos import ActorBuilder, SpacecraftActor

# Define the epoch and orbital elements
epoch = pk.epoch(2460000.5, "jd")
elements = (3.457 * pk.AU, 0.64989, 3.8719 * pk.DEG2RAD, 36.33 * pk.DEG2RAD, 22.15 * pk.
↪DEG2RAD, 73.57 * pk.DEG2RAD)

# Create a planet object from pykep for 67P
comet = pk.planet.keplerian(epoch, elements, pk.MU_SUN, 666.19868, 2000, 2000, "67P")

# Load the 67P mesh with pickle
with open(mesh_path, "rb") as f:
    mesh_points, mesh_triangles = pickle.load(f)
    mesh_points = np.array(mesh_points)
    mesh_triangles = np.array(mesh_triangles)

# Define local actor
my_sat = ActorBuilder.get_actor_scaffold("my_sat", SpacecraftActor, epoch=epoch)

# Set the custom propagator
ActorBuilder.set_custom_orbit(my_sat, polyhedral_propagator, epoch)
```

(continues on next page)

(continued from previous page)

```
# Set the mesh
ActorBuilder.set_central_body(my_sat, comet, (mesh_points, mesh_triangles))

# Below computations will now use the mesh instead spherical approximations
print(my_sat.is_in_eclipse())
print(my_sat.is_in_line_of_sight(some_other_actor))

# You could even specify a rotation of the central body.
# Set a rotation period of 1 second around the z axis
ActorBuilder.set_central_body(
    my_sat,
    comet,
    (mesh_points, mesh_triangles),
    rotation_declination=90,
    rotation_right_ascension=0,
    rotation_period=1,
)
```

This is particularly useful if you want to use a central body that is not included in pykep or if you want to use a central body that is not a planet (e.g. an asteroid).

N.B. `get_altitude` computes the altitude above [0,0,0] in the central body's frame, thus is not affected by the central body's rotation or mesh. N.B. #2 Any custom central body still has to orbit the Sun for PASEOS to function correctly.

## 6.3 Simulation Settings

### 6.3.1 Initializing PASEOS

We will now show how to create an instance of PASEOS. An instance of PASEOS shall be bounded to one PASEOS *actor* that we call *local actor*. Please, notice that an orbit shall be placed for a *SpacecraftActor* before being added to a PASEOS instance.

### 6.3.2 How to instantiate PASEOS

```
import pykep as pk
import paseos
from paseos import ActorBuilder, SpacecraftActor
# Define the local actor as a SpacecraftActor of name mySat and its orbit
local_actor = ActorBuilder.get_actor_scaffold(name="mySat",
                                              actor_type=SpacecraftActor,
                                              epoch=pk.epoch(0))
ActorBuilder.set_orbit(
    actor=local_actor,
    position=[10000000, 0, 0],
    velocity=[0, 8000.0, 0],
    epoch=pk.epoch(0),
    central_body=pk.planet.jpl_lp("earth"),  # use Earth from pykep
)
```

(continues on next page)

(continued from previous page)

```
# initialize PASEOS simulation
sim = paseos.init_sim(local_actor)
```

For each actor you wish to model, you can create a PASEOS instance. Running multiple instances on the same machine / thread is supported.

### 6.3.3 Using the cfg

When you instantiate PASEOS as shown in *Initializing PASEOS*, a PASEOS instance is created by using the default configuration. However, sometimes it is useful to use a custom configuration.

The next code snippet will show how to start the PASEOS simulation with a time different from `pk.epoch(0)` (MJD2000) by loading a custom configuration.

```
import pykep as pk
import paseos
from paseos import ActorBuilder, SpacecraftActor

#Define today as pykep epoch (16-06-22)
#please, refer to https://esa.github.io/pykep/documentation/core.html#pykep.epoch
today = pk.epoch_from_string('2022-06-16 00:00:00.000')

# Define the local actor as a SpacecraftActor of name mySat
# pk.epoch is set to today
local_actor = ActorBuilder.get_actor_scaffold(name="mySat",
                                              actor_type=SpacecraftActor,
                                              epoch=today)

# Let's set the orbit of local_actor.
# pk.epoch is set to today
ActorBuilder.set_orbit(
    actor=local_actor,
    position=[10000000, 0, 0],
    velocity=[0, 8000.0, 0],
    epoch=pk.epoch(0),
    central_body=pk.planet.jpl_lp("earth"),  # use Earth from pykep
)

# Loading cfg to modify defaults
cfg=load_default_cfg()
# Set simulation starting time by converting epoch to seconds
cfg.sim.start_time=today.mjd2000 * pk.DAY2SEC
# initialize PASEOS simulation
sim = paseos.init_sim(local_actor)
```

You can access the current simulation time (seconds since the start) and the current epoch like this:

```
time_since_start_in_s = sim.simulation_time
current_epoch = sim.local_time
```

### 6.3.4 Faster than real-time execution

In some cases, you may be interested to simulate your spacecraft operating for an extended period. By default, PASEOS operates in real-time, thus this would take a lot of time. However, you can increase the rate of time passing (i.e. the spacecraft moving, power being charged / consumed etc.) using the `time_multiplier` parameter. Set it as follows when initializing PASEOS.

```
cfg = load_default_cfg() # loading cfg to modify defaults
cfg.sim.time_multiplier = 10 # setting the parameter so that in 1s real time, paseos␣
↪models 10s having passed
paseos_instance = paseos.init_sim(my_local_actor, cfg) # initialize paseos instance
```

### 6.3.5 Event-based mode

Alternatively, you can rely on an event-based mode where PASEOS will simulate the physical constraints for an amount of time. The below code shows how to run PASEOS for a fixed amount of time or until an event interrupts it.

```
    import pykep as pk
    import paseos
    from paseos import ActorBuilder, SpacecraftActor

    # Define the central body as Earth by using pykep APIs.
    earth = pk.planet.jpl_lp("earth")

    # Define a satellite with some orbit and simple power model
    local_actor = ActorBuilder.get_actor_scaffold("MySat", SpacecraftActor, pk.epoch(0))
    ActorBuilder.set_orbit(local_actor, [10000000, 0, 0], [0, 8000.0, 0], pk.epoch(0),␣
↪earth)
    ActorBuilder.set_power_devices(local_actor, 500, 1000, 1)

    # Abort when sat is at 10% battery
    def constraint_func():
        return local_actor.state_of_charge > 0.1

    # Set some settings to control evaluation of the constraint
    cfg = load_default_cfg()  # loading cfg to modify defaults
    cfg.sim.dt = 0.1  # setting timestep of physical models (power, thermal, ...)
    cfg.sim.activity_timestep = 1.0  # how often constraint func is evaluated
    sim = paseos.init_sim(local_actor, cfg) # Init simulation

    # Advance for a long time, will interrupt much sooner due to constraint function
    sim.advance_time(3600, 10, constraint_function=constraint_func)
```

## 6.4 Activities

### 6.4.1 Simple activity

PASEOS enables the user to register their *activities* that will be executed on the `local actor`. This is an alternative to the *event-based mode*

To register an activity, it is first necessary to define an asynchronous *activity function*. The following code snippet shows how to create a simple *activity function* `activity_function_A` that prints "Hello Universe!". Then, it waits for 0.1 s before concluding the activity. When you register an *activity*, you need to specify the power consumption associated to the activity.

```
#Activity function
async def activity_function_A(args):
  print("Hello Universe!")
  await asyncio.sleep(0.1) #Await is needed inside an async function.
```

Once an activity is registered, the user shall call `perform_activity(...)` to run the registered activity. The next snippet will showcase how to register and perform the activity `activity_A`.

```
import pykep as pk
import paseos
from paseos import ActorBuilder, SpacecraftActor
import asyncio
# Define the local actor as a SpacecraftActor of name mySat and its orbit
local_actor = ActorBuilder.get_actor_scaffold(name="mySat",
                                      actor_type=SpacecraftActor,
                                      epoch=pk.epoch(0))

ActorBuilder.set_orbit(
    actor=local_actor,
    position=[10000000, 0, 0],
    velocity=[0, 8000.0, 0],
    epoch=pk.epoch(0),
    central_body=pk.planet.jpl_lp("earth"),  # use Earth from pykep
)

# Add a power device
ActorBuilder.set_power_devices(actor=local_actor,
                               # Battery level at the start of the simulation in Ws
                               battery_level_in_Ws=100,
                               # Max battery level in Ws
                               max_battery_level_in_Ws=2000,
                               # Charging rate in W
                               charging_rate_in_W=10)

# initialize PASEOS simulation
sim = paseos.init_sim(local_actor)

#Activity function
async def activity_function_A(args):
  print("Hello Universe!")
  await asyncio.sleep(0.1) #Await is needed inside an async function.
```

(continues on next page)

(continued from previous page)

```
# Register an activity that emulate event detection
sim.register_activity(
    "activity_A",
    activity_function=activity_function_A,
    power_consumption_in_watt=10
)

#Run the activity
sim.perform_activity("activity_A")
```

#### Waiting for Activities to Finish

At the moment, parallel running of multiple activities is not supported. However, if you want to run multiple activities in a row or just wait for the existing one to finish, you can use

```
await sim.wait_for_activity()
```

to wait until the running activity has finished.

### 6.4.2 Activities with Inputs and Outputs

The next code snippet will show how to register and perform activities with inputs and outputs. In particular, we will register an *activity function* `activity_function_with_in_and_outs` that takes an input argument and returns its value multiplied by two. Then, it waits for 0.1 s before concluding the activity. Please, notice that the output value is placed in `args[1][0]`, which is returned as reference.

```
import pykep as pk
import paseos
from paseos import ActorBuilder, SpacecraftActor
import asyncio
# Define the local actor as a SpacecraftActor of name mySat and its orbit
local_actor = ActorBuilder.get_actor_scaffold(name="mySat",
                                              actor_type=SpacecraftActor,
                                              epoch=pk.epoch(0))

ActorBuilder.set_orbit(
    actor=local_actor,
    position=[10000000, 0, 0],
    velocity=[0, 8000.0, 0],
    epoch=pk.epoch(0),
    central_body=pk.planet.jpl_lp("earth"),  # use Earth from pykep
)

# Add a power device
ActorBuilder.set_power_devices(actor=local_actor,
                               # Battery level at the start of the simulation in Ws
                               battery_level_in_Ws=100,
                               # Max battery level in Ws
                               max_battery_level_in_Ws=2000,
```

(continues on next page)

(continued from previous page)

```
                                # Charging rate in W
                                charging_rate_in_W=10)

# initialize PASEOS simulation
sim = paseos.init_sim(local_actor)

#Activity function
async def activity_function_with_in_and_outs(args):
  activity_in=args[0]
  activity_out=activity_in * 2
  args[1][0]=activity_out
  await asyncio.sleep(0.1) #Await is needed inside an async function.

# Register an activity that emulate event detection
sim.register_activity(
    "my_activity",
    activity_function=activity_function_with_in_and_outs,
    power_consumption_in_watt=10,
)

#Creatie an input variable for activity
activity_in=1

#Create a placeholder variable to contain the output of the activity function.
#It is created as a list so its first value is edited
# as reference by the activity function.
activity_out=[None]

#Run the activity
sim.perform_activity("my_activity",
                     activity_func_args=[activity_in, activity_out],
                    )
#Print return value
print("The output of the activity function is: ", activity_out[0])
```

### 6.4.3 Constraint Function

It is possible to associate a *constraint function* with each *activity* to ensure that some particular constraints are met during the *activity* execution. When constraints are not met, the activity is interrupted. Constraints can be used, e.g., to impose power requirements, communication windows or maximum operational temperatures. The next code snippet shows how to:

- create a *constraint function* (`constraint_function_A`) which returns `True` when the local actor's temperature is below ~86°C and `False` otherwise (this requires a thermal model on the actor)
- how use `constraint_function_A` to constraint our *Simple Activity*.

```
import pykep as pk
import paseos
from paseos import ActorBuilder, SpacecraftActor
import asyncio
# Define the local actor as a SpacecraftActor of name mySat and its orbit
```

(continues on next page)

(continued from previous page)

```
local_actor = ActorBuilder.get_actor_scaffold(name="mySat",
                                              actor_type=SpacecraftActor,
                                              epoch=pk.epoch(0))

ActorBuilder.set_orbit(
    actor=local_actor,
    position=[10000000, 0, 0],
    velocity=[0, 8000.0, 0],
    epoch=pk.epoch(0),
    central_body=pk.planet.jpl_lp("earth"),  # use Earth from pykep
)

# Add a power device
ActorBuilder.set_power_devices(actor=local_actor,
                               # Battery level at the start of the simulation in Ws
                               battery_level_in_Ws=100,
                               # Max battery level in Ws
                               max_battery_level_in_Ws=2000,
                               # Charging rate in W
                               charging_rate_in_W=10)

# initialize PASEOS simulation
sim = paseos.init_sim(local_actor)

#Activity function
async def activity_function_A(args):
  print("Hello Universe!")
  await asyncio.sleep(0.1) #Await is needed inside an async function.

#Constraint function
async def constraint_function_A(args):
  local_actor_temperature=args[0]
  return (local_actor_temperature < 350)

# Register an activity that emulate event detection
sim.register_activity(
    "activity_A_with_constraint",
    activity_function=activity_function_A,
    power_consumption_in_watt=10,
    constraint_function=constraint_function_A
)

#The constraint function is related to the operational temperature of the local actor.
operational_temperature_in_K=local_actor.temperature_in_K

#Run the activity
sim.perform_activity("activity_A_with_constraint",
                     constraint_func_args=[operational_temperature_in_K],
                    )
```

### 6.4.4 On-termination Function

It is also possible to define an *on-termination function* to perform some specific operations when on termination of the *activity*. The next code snippet shows:

- how to create an *on-termination function* that prints "activity (activity_A_with_termination_function) ended.".
- How to associate our *on-termination function* to our *Simple Activity*.

The name of the *activity* is passed as input to the *on-termination function* to showcase to handle *on-termination function* inputs.

```
import pykep as pk
import paseos
from paseos import ActorBuilder, SpacecraftActor
import asyncio
# Define the local actor as a SpacecraftActor of name mySat and its orbit
local_actor = ActorBuilder.get_actor_scaffold(name="mySat",
                                        actor_type=SpacecraftActor,
                                        epoch=pk.epoch(0))

ActorBuilder.set_orbit(
    actor=local_actor,
    position=[10000000, 0, 0],
    velocity=[0, 8000.0, 0],
    epoch=pk.epoch(0),
    central_body=pk.planet.jpl_lp("earth"),  # use Earth from pykep
)

# Add a power device
ActorBuilder.set_power_devices(actor=local_actor,
                               # Battery level at the start of the simulation in Ws
                               battery_level_in_Ws=100,
                               # Max battery level in Ws
                               max_battery_level_in_Ws=2000,
                               # Charging rate in W
                               charging_rate_in_W=10)

# initialize PASEOS simulation
sim = paseos.init_sim(local_actor)

#Activity function
async def activity_function_A(args):
  print("Hello Universe!")
  await asyncio.sleep(0.1) #Await is needed inside an async function.

#On-termination function
async def on_termination_function_A(args):
  #Fetching input
  activity_name=args[0]
  print("Activity ("+str(activity_name)+") ended.")

# Register an activity that emulate event detection
sim.register_activity(
    "activity_A_with_termination_function",
```

(continues on next page)

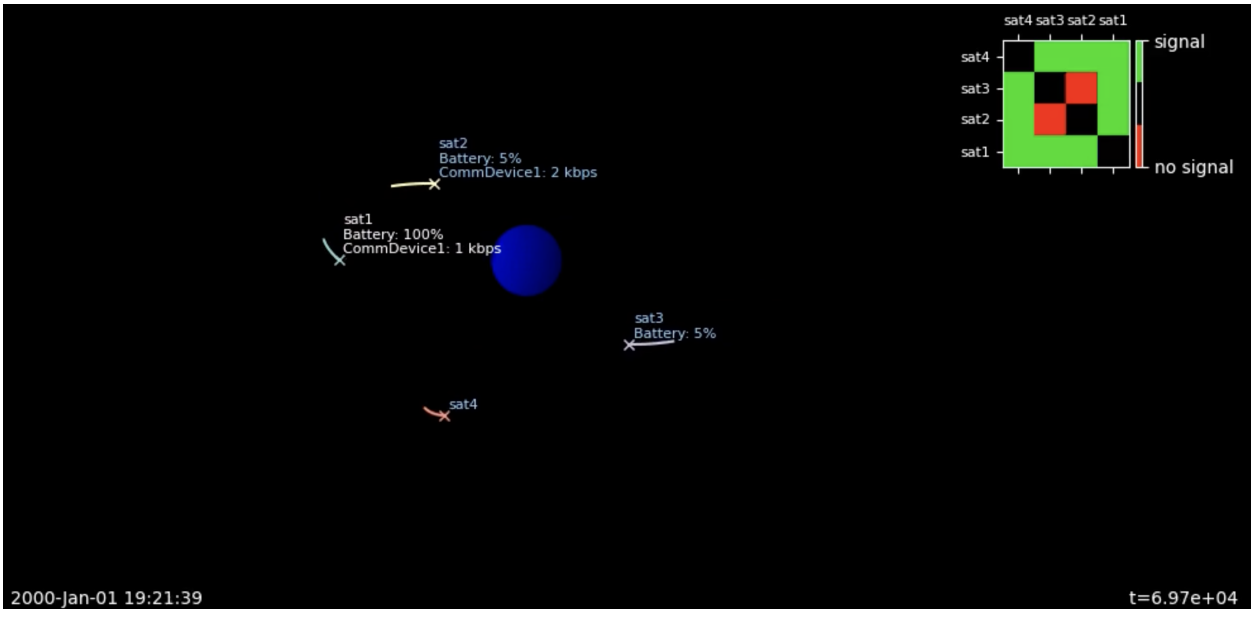
(continued from previous page)

```
    activity_function=activity_function_A,
    power_consumption_in_watt=10,
    on_termination_function=on_termination_function_A
)

#The termination function input is the activity name
activity_name="activity_A_with_termination_function"

#Run the activity
sim.perform_activity("activity_A_with_termination_function",
                     termination_func_args=[activity_name],
                    )
```

## 6.5 Utilities

### 6.5.1 Visualization

Navigate to paseos/visualization to find a jupyter notebook containing examples of how to visualize PASEOS. Visualization can be done in interactive mode or as an animation that is saved to your disc. In the figure below, Earth is visualized in the centre as a blue sphere with different spacecraft in orbit. Each spacecraft has a name and if provided, a battery level and a communications device. The local device is illustrated with white text. In the upper-right corner, the status of the communication link between each spacecraft is shown. Finally, the time in the lower left and lower right corners corresponds to the epoch and the PASEOS local simulation time.



### 6.5.2 Monitoring Simulation Status

You can easily track the status of a PASEOS simulation via the `monitor` which keeps track of actor status.

It allows access like this

```
(...) # actor definition etc., see above
instance = paseos.init_sim(local_actor=my_local_actor)

(...) # running the simulation

# access tracked parameters
timesteps = instance.monitor["timesteps"]
state_of_charge = instance.monitor["state_of_charge"]
```

### 6.5.3 Writing Simulation Results to a File

To evaluate your results, you will likely want to track the operational parameters, such as actor battery status, currently running activity etc. of actors over the course of your simulation. By default, PASEOS will log the current actor status every 10 seconds, however you can change that rate by editing the default configuration, as explained in *How to use the cfg*. You can save the current log to a *.csv file at any point.

```
cfg = load_default_cfg() # loading cfg to modify defaults
cfg.io.logging_interval = 0.25  # log every 0.25 seconds
paseos_instance = paseos.init_sim(my_local_actor, cfg) # initialize paseos instance

# Performing activities, running the simulation (...)

paseos_instance.save_status_log_csv("output.csv")
```

## 6.6 Wrapping Other Software and Tools

PASEOS is designed to allow easily wrapping other software and tools to, e.g., use more sophisticated models for specific aspects of interest to the user. There are three ways to do this:

- *Via Activities* - An *activity* using an external software is registered and executed as any other *activity*, e.g. to perform some computations while tracking runtime of that operation.
- *Via Constraint Functions* - A *constraint function* using an external software. This is useful to use a more sophisticated model to check whether, e.g., a physical constraint modelled outside of PASEOS is met.
- *Via Custom Properties* - A *custom property* using an external software. This is useful to, e.g., use a more sophisticated model for a physical quantity such as total ionization dose or current channel bandwidth.

### 6.6.1 Via Activities

The wrapping via activities is quite straight forward. Follow the *instructions on registering and performing activities* and make use of your external software inside the activity function.

```
import my_external_software
#Activity function
async def activity_function_A(args):
  my_external_software.complex_task_to_model()
  await asyncio.sleep(0.01)
```

### 6.6.2 Via Constraint Functions

Inside constraint functions, external software can be used to check whether a constraint is met or not. This works both for *activity constraints* and for *constraints in event-based mode*.

The constraint function should return `True` if the constraint is met and `False` otherwise.

```
import pykep as pk
from paseos import ActorBuilder, SpacecraftActor

import my_complex_radiation_model

# Defining a local actor
local_actor = ActorBuilder.get_actor_scaffold("MySat", SpacecraftActor, pk.epoch(0))

def constraint_func():
  t = local_actor.local_time
  device_has_failed = my_complex_radiation_model.check_for_device_failure(t)
  return not device_has_failed

# Can be passed either with event-based mode, will run until constraint is not met
sim.advance_time(3600, 10, constraint_function=constraint_func)

# (...)

# or via activity constraints, will run until constraint is not met
# N.B: this is an excerpt follow the #constraint-function link for more details
sim.register_activity(
    "activity_A_with_constraint_function",
    activity_function=activity_function_A,
    power_consumption_in_watt=10,
    constraint_function=constraint_func
)
```

### 6.6.3 Via Custom Properties

Finally, *custom properties* can be used to wrap external software. This is useful to use a more sophisticated model for a physical quantity, e.g. one could use a simulator like ns-3 to model the current channel bandwidth.

For more details see *custom properties*.

```
import my_channel_model

# Will be automatically called during PASEOS simulation
def update_function(actor, dt, power_consumption):
    # Get the current channel bandwidth from the external model
    channel_bandwidth = my_channel_model.get_channel_bandwidth(actor)
    return channel_bandwidth

# Add the custom property to the actor, defining name, update fn and initial value
ActorBuilder.add_custom_property(
    actor=local_actor,
    property_name="channel_bandwidth",
    update_function=update_function,
    initial_value=1000,
)

# (... run simulation)

# One can easily access the property at any point with
print(local_actor.get_custom_property("channel_bandwidth"))
```

CHAPTER

SEVEN

# GLOSSARY

- **Activity**

  Activity is the abstraction that PASEOS uses to keep track of specific actions performed by an *actor* upon a request from the user. >PASEOS is responsible for the execution of the activity and for updating the system status depending on the effects of the activity (e.g., by discharging the satellite battery). When registering an activity, the user can specify a *constraint function* to specify constraints to be met during the execution of the activity and an *on-termination* function to specify additional operations to be performed by PASEOS on termination of the activity function.

- **Activity function**

  User-defined function emulating any operation to be executed in a PASEOS by an *actor*. Activity functions are necessary to register *activities*. Activity functions might include data transmission, housekeeping operations, onboard data acquisition and processing, and others.

- **Actor**

  Since PASEOS is fully-decentralised, each node of a PASEOS constellation shall run an instance of PASEOS modelling all the nodes of that constellation. The abstraction of a constellation node inside a PASEOS instance is a PASEOS `actor`.

- **Constraint function**

  A constraint function is an asynchronous function that can be used by the PASEOS user to specify some constraints that shall be met during the execution of an activity.

- **Custom Property**

  Users can define their own physical quantity to track parameters not natively simulated by PASEOS. This is described in detail *above* and in a dedicated example notebook on modelling total ionizing dose.

- **GroundstationActor**

  `PASEOS actor` emulating a ground station.

- **Local actor**

  The `local actor` is the `actor` whose behaviour is modelled by the locally running PASEOS instance.

- **Known actors**

  In a PASEOS instance, `known actors` are all the other actors that are known to the *local actor*.

- **On-termination function**

  An on-termination function is an asynchronous function that can be used by the PASEOS user to specify some operations to be executed on termination of the predefined PASEOS user's activity.

- **SpacecraftActor**

  PASEOS *actor* emulating a spacecraft or a satellite.

## 7.1 Physical Model Parameters

Description of the physical model parameters and default values in PASEOS with indications on sensitivity of parameters and suggested ranges.

| Name | Datatype | Description | Default | Suggested Range | Sensitivity |
|---|---|---|---|---|---|
| Battery Level [Ws] | float | Current battery level | - | > 0 | high |
| Maximum Battery Level [Ws] | float | Maximum battery level | - | > 0 | high |
| Charging Rate [W] | float | Charging rate of the battery | - | > 0 | high |
| Power Device Type | enum | Type of power device. Can be either "SolarPanel" or "RTG" | SolarPanel | - | medium |
| Data Corruption Events [Hz] | float | Rate of single bit of data being corrupted, i.e. a Single Event Upset (SEU) | - | >= 0 | low |
| Restart Events [Hz] | float | Rate of device restart being triggered | - | >= 0 | medium |
| Failure Events [Hz] | float | Rate of complete device failure due to a Single Event Latch-Up (SEL) | - | >= 0 | high |
| Mass [kg] | float | Actor's mass | - | > 0 | low |
| Initial Temperature [K] | float | Actor's initial temperature | - | >= 0 | medium |
| Sun Absorptance | float | Actor's absorptance of solar light | - | [0,1] | high |
| Infrared Absorptance | float | Actor's absportance of infrared light | - | [0,1] | medium |
| Sun-Facing Area [$m^2$] | float | Actor's area facing the sun | - | >= 0 | high |
| Central Body-Facing Area [$m^2$] | float | Actor's area facing central body | - | >= 0 | medium |
| Emissive Area [$m^2$] | float | Actor's area emitting (radiating) heat | - | >= 0 | high |
| Thermal Capacity [$J / (kg * K)$] | float | Actor's thermal capacity | - | >= 0 | low |
| Body Solar Irradiance [W] | float | Irradiance from the sun | 1360 | >= 0 | medium |
| Body Surface Temperature [K] | float | Central body surface temperature | 288 | >= 0 | low |
| Body Emissivity | float | Central body emissivity in infrared | 0.6 | [0,1] | medium |
| Body Reflectance | float | Central body reflectance of sunlight | 0.3 | [0,1] | medium |
| Heat Conversion Ratio [-] | float | Conversion ratio for activities, 0 leads to know heat-up due to activity | 0.5 | [0,1] | high |

CHAPTER

EIGHT

# CONTRIBUTING

The PASEOS project is open to contributions. To contribute, you can open an issue to report a bug or to request a new feature. If you prefer discussing new ideas and applications, you can contact us via email (please, refer to *Contact*). To contribute, please proceed as follow:

1. Fork the Project
2. Create your Feature Branch (`git checkout -b feature/AmazingFeature`)
3. Commit your Changes (`git commit -m 'Add some AmazingFeature'`)
4. Push to the Branch (`git push origin feature/AmazingFeature`)
5. Open a Pull Request

CHAPTER

NINE

# LICENSE

Distributed under the GPL-3.0 License.

CHAPTER

TEN

# CONTACT

Created by $\Phi$-lab@Sweden.

- Pablo Gómez - pablo.gomez at esa.int, pablo.gomez at ai.se
- Gabriele Meoni - gabriele.meoni at esa.int, g.meoni at tudelft.nl
- Johan Östman - johan.ostman at ai.se
- Vinutha Magal Shreenath - vinutha at ai.se

CHAPTER

# ELEVEN

# REFERENCE

If you have used PASEOS, please cite the following paper:

```
@article{gomez23paseos,
  author = {Gómez, Pablo and Östman, Johan and Shreenath, Vinutha Magal and Meoni,␣
↪Gabriele},
  title = {{PA}seos {S}imulates the {E}nvironment for {O}perating multiple {S}pacecraft},
  journal = {arXiv:2302.02659 [cs.DC]},
  year = {2023},
}
```

CHAPTER

TWELVE

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

# INDEX

## M



## N



## P



## R



## S



## T



## W