AST2000SolarSystem Module Documentation (2018)

This documentation is written for the project variant of AST2000 course (formerly known as AST1100) at the University of Oslo. Students following the standard variant of the course should use the `AST2000SolarSystemViewer` module instead.

CONTENTS

I. INTRODUCTION

A. The Purpose Of The `AST2000SolarSystem` Module

The project variant of *AST2000 - Introduction to astrophysics* is a cohesive project in which your later work is dependent on the results of your earlier work, the purpose of the `AST2000SolarSystem` module is to accomplish this unification. Much like the name implies, the module gives you access to your own personal solar system in which all your calculations *"take place"*. All physical parameters concerning your solar system are based on data from this module. Some parameters, such as the surface temperature of the solar system's star, can be accessed directly as class attributes, others, such as the solar system's planetary orbits, must be calculated/simulated using theory from the course Lecture Notes.

B. Terminology In This Document

Below is a list of relevant terminology used throughout this document.

Expression	Description
(N/A)	*"Not Applicable"*.
`static`	A static class attribute or method is one that does not need an instance of the class.
`seed`	The `AST2000SolarSystem seed` is an integer that uniquely defines a solar system.
`np`	Short for `numpy` (or `NumPy`), the scientific library for `Python`.
`default`	The default parameter value in a function or method.
`NumPy` shape `()`	The variable does not have a dimension. This includes `Python types` such as `int` and `float`.

C. Initializing A Solar System

To initialize your personal solar system, you need to generate an `AST2000SolarSystem seed`. There are two ways to do this, you can either use the `static` method `get_seed` from the `AST2000SolarSystem` module directly, or you can run the `myseed.pyc` script from the course page.

If you want to use the `static` method, simply run a script containing these lines:

```
from AST2000SolarSystem import AST2000SolarSystem
AST2000SolarSystem.get_seed(username)
```

Note that you need to use your UiO username! The method both returns and prints your seed.

In case you want to use the `myseed.pyc` script, you can download it from the course pages. Run the program as shown below using your UiO username, the resulting terminal output (displayed as `XXXXX` below) is your `AST2000SolarSystem seed`.

```
[terminal]$python3 myseed.pyc username
XXXXX
```

You can now access your solar system by initializing an instance of the `AST2000SolarSystem` module using your `seed` as the defining parameter:

```
from AST2000SolarSystem import AST2000SolarSystem
seed        = XXXXX
SolarSystem = AST2000SolarSystem(seed)
```

II. LIBRARY

A. Attributes

1. *Attributes in alphabetical order*

The full list of available `AST2000SolarSystem` attributes is seen below, the attributes can be divided into three types: satellite attributes, star attributes and planet attributes. In order to access the attribute `attribute`, simply code:

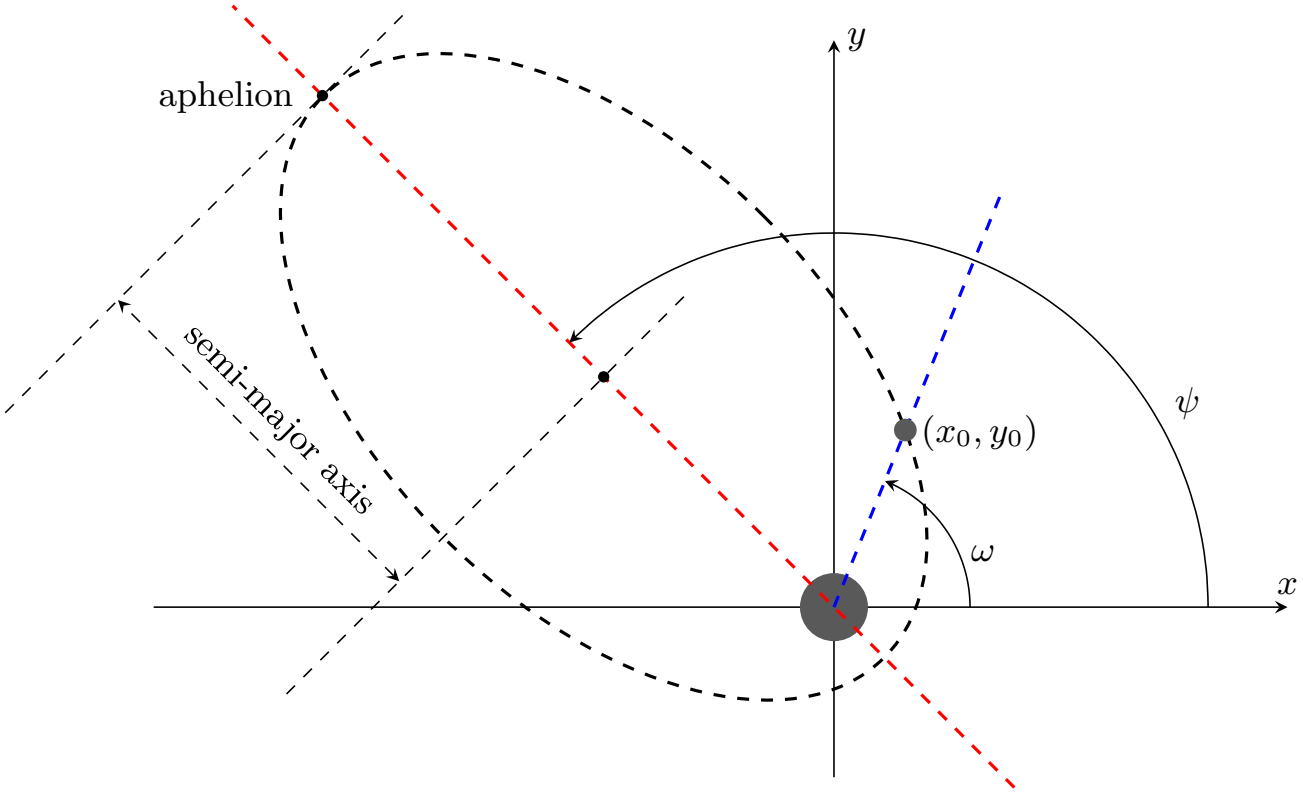
```
from AST2000SolarSystem import AST2000SolarSystem
seed        = AST2000SolarSystem.get_seed(username)
SolarSystem = AST2000SolarSystem(seed)
attribute   = SolarSystem.attribute
```

where `username` is your UiO username. Note that the planet with index 0 is your home planet.

Attribute name	`Python type`	Physical unit	Description
`a`	`np.ndarray`	AU	A `NumPy` array of length `number_of_planets` containing the semi-major axes of the planets.
`area_lander`	`float`	m^2	The surface area of the lander unit.
`area_sat`	`float`	m^2	The surface area of the satellite.
`e`	`np.ndarray`	(N/A)	A `NumPy` array of length `number_of_planets` containing the eccentricities of the planetary orbits.
`mass`	`np.ndarray`	$M_\odot$	A `NumPy` array of length `number_of_planets` containing the masses of the planets.
`mass_lander`	`float`	kg	The mass of the lander unit.
`mass_sat`	`float`	kg	The mass of the satellite.
`number_of_planets`	`int`	(N/A)	The number of planets in the solar system.
`omega`	`np.ndarray`	rad	A `NumPy` array of length `number_of_planets` containing the polar angle of the initial positions of the planets.
`period`	`np.ndarray`	24 hrs	A `NumPy` array of length `number_of_planets` containing the periods of the planets' rotation about the z-axis.
`psi`	`np.ndarray`	rad	A `NumPy` array of length `number_of_planets` containing the polar angle of the aphelion of the planetary orbits.
`radius`	`np.ndarray`	km	A `NumPy` array of length `number_of_planets` containing the radii of the planets.
`rho0`	`np.ndarray`	kg m^{-3}	A `NumPy` array of length `number_of_planets` containing the atmospheric densities at the surface of the planets.
`star_mass`	`np.float64`	$M_\odot$	The mass of the star.
`star_radius`	`np.float64`	km	The radius of the star.
`temperature`	`float`	K	The surface temperature of the star.
`vx0`	`np.ndarray`	AU/yrs	A `NumPy` array of length `number_of_planets` containing the x-components of the initial velocities of the planets as seen from the solar system frame of reference.
`vy0`	`np.ndarray`	AU/yrs	A `NumPy` array of length `number_of_planets` containing the y-components of the initial velocities of the planets as seen from the solar system frame of reference.
`x0`	`np.ndarray`	AU	A `NumPy` array of length `number_of_planets` containing the x-components of the initial positions of the planets as seen from the solar system frame of reference.
`y0`	`np.ndarray`	AU	A `NumPy` array of length `number_of_planets` containing the y-components of the initial positions of the planets as seen from the solar system frame of reference.

2. *An illustration of the planetary orbits*

For convenience, an illustration of the relationship between attributes `a`, `e`, `omega`, `x0`, `y0` and `psi` is shown below. The black, thick and dashed elliptical line is the orbit of the planet. The dashed red line indicates the angle of `psi` (ψ), which is the angle of the aphelion of the orbit. The blue line indicates the angle of `omega` (ω), which is the angle of the initial position of the planet in the orbit `(x0,y0)`. Finally, `a` is the semi-major axis of the elliptical orbit.



B. Methods

There are 2 categories of methods you will encounter during this project: methods related to the satellite and methods related to relativity exercises. The satellite methods are listed in II B 1 and the relativity methods are listed in II B 2. All non-`static` methods are listed using the keyword `System` as a placeholder for your instance of the `AST2000SolarSystem` class. Note that `static` functions do not require an instance of `AST2000SolarSystem` and are therefore not listed with the `System` keyword.

Some of the satellite methods are marked using the tag "Hierarchy", this means that another method must be run (in the same program) before running this method. A complete description of the entire hierarchy is given below in section II B 3.

1. *Satellite methods in alphabetical order*

Shortcuts:


```
distances = System.analyse_distances()
```

Analyses the distances from the satellite to the center of the planets and the star using onboard equipment. The distances are both returned and saved to file `pos.npy`

Returns output	`static`	Prints to terminal	Hierarchy	Other
✓	✗	✓	✓	✓

Function output:

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`distances`	`np.ndarray`	`(number_of_planets+1,)`	AU	A `NumPy` array containing the distances from the satellite to the center of the planets and star. The distance to the star is the final element, the planets are listed according to their planet index in `AST2000SolarSystem`.

`idx = AST2000SolarSystem.ang2pix(theta,phi)`

Transforms the spherical angular direction given by `theta` and `phi` to the corresponding index needed to access the `RGB` data in `himmelkule.npy`.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✓	✓	✗	✗	✗

Function arguments:

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`theta`	`float`	`()`	`radians`	The polar angle of the pixel (the vertical angle).
`phi`	`float`	`()`	`radians`	The azimuthal angle of the pixel (the horizontal angle).

Function output:

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`idx`	`int`	`()`	(N/A)	The index used to access the `RGB` value of the pixel located in the direction of `theta` and `phi`.

About `himmelkule.npy`:
In order for the index `idx` from `ang2pix()` to work, you need to have loaded `himmelkule.npy` using `np.load()`.

`System.check_planet_positions(positions, T_sim, N_yr, writeFile=True)`

Checks the accuracy of the simulated planetary orbits. If the accuracy is sufficient and `writeFile=True`, then the method generates *"perfect orbits"* and saves them as a binary `npy` file named `planet_positions.npy`.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✗	✓

Function arguments:
Let N_{time} denote the number of time steps in the orbit simulation.

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`positions`	`np.ndarray`	`(2,number_of_planets,`N_{time}`)`	AU	A `NumPy` array containing the positions of all of the planets at equidistant consecutive time steps.
`T_sim`	`float`	`()`	yrs	The duration of the orbits simulation.
`N_yr`	`float`	`()`	yrs^{-1}	The number of time steps per year in the orbits simulation.
`writeFile`	`bool`	`()`	(N/A)	`True/False`: Save the perfect orbits to `planet_positions.npy`. `default=False`.

About `planet_positions.npy`:
The `npy` file contains two arrays: `positions` and `times`. The `positions` array is completely analogous to the function argument `positions`, while `times` is the corresponding time array. The arrays have the same resolution as the resolution specified by function arguments `T_sim` and `N_yr`.

`System.engine_settings(dpdt, N_box, dNdt, m_fuel, T_launch, launch_pos, t_launch)`

Installs the engine settings in `System` and prepares the satellite launch.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✗	✗	✗

Function arguments:

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`dpdt`	`float`	`()`	kg m s^{-2}	The thrust force from a single box.
`N_box`	`int/float`	`()`	(N/A)	The number of boxes that comprises the rocket engine.
`dNdt`	`float`	`()`	s^{-1}	The rate of particles that escapes from a single box.
`m_fuel`	`float`	`()`	kg	The initial amount of fuel for the entire mission.
`T_launch`	`float`	`()`	s	The expected amount of time the launch will take.
`launch_pos`	`np.ndarray`	`(2,)`	AU	The initial position of the rocket before launching.
`t_launch`	`float`	`()`	yrs	The time of launch in the solar system frame of reference.

`(ref1, ref2) = System.get_ref_stars()`

Prints and returns star reference data. Each `ref` includes the angular coordinate of the reference star ϕ and the Doppler shift in the H_α spectral line from the reference star as seen from your solar system's frame of reference.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✓	✗	✓	✗	✗

Function output:
Replace `X` by either `1` or `2`.

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`refX`	`tuple`	`(2,)`	(N/A)	`refX = (phiX,lambX)`
`phiX`	`float`	`()`	`degrees`	The angular coordinate $\phi_{\texttt{X}}$ of reference star `X`.
`lambX`	`float`	`()`	nm	The Doppler shift in the H_α spectral line from reference star `X` as seen from your solar system's frame of reference.

`seed = AST2000SolarSystem.get_seed(username)`

Prints and returns `AST2000SolarSystem seed` based on a UiO username.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✓	✓	✓	✗	✗

Function arguments:

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`seed`	`int`	`()`	(N/A)	The `AST2000SolarSystem seed` corresponding to the UiO username `username`.

`System.land_on_planet(p, fileName="landCommands.txt")`

Loads the landing commands in `fileName` and executes them (For Realz).

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✓	✗

Function arguments:

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`p`	`int`	`()`	(N/A)	The index of the target planet you wish to land on.
`fileName`	`str`	`()`	(N/A)	The filename for the landing command file. For available commands see II C 2.

`System.manual_orientation(ang, satVel, satPos)`

Calibrates the orientation software onboard the satellite by comparing calculated values to known ones.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✓	✗

Function arguments:

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`ang`	`float`	`()`	`degrees`	The angular orientation of the satellite.
`satVel`	`np.ndarray`	`(2,)`	AU/yr	The velocity of the satellite in the solar system's frame of reference.
`satPos`	`np.ndarray`	`(2,)`	AU	The position of the satellite in the solar system's frame of reference.

`System.mass_needed_launch(final_launch_pos, test=False)`

Installs the engine settings in `System` and prepares the satellite launch.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✓	✗

Function arguments:

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`final_launch_pos`	`np.ndarray`	`(2,)`	AU	The final position of the rocket after launching.
`test`	`bool`	`()`	(N/A)	`True/False`: Test accuracy of `final_launch_pos`. `default = False`.

`dlambda1, dlambda2 = System.measure_doppler_shifts()`

Measures the Doppler shifts in the H_α spectral line from reference stars using the satellite's onboard equipment.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✓	✗	✓	✓	✗

Function output:
Replace `X` by either `1` or `2`.

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`dlambdaX`	`float`	`()`	nm	The Doppler shift in the H_α spectral line from reference star `X`.

`System.orbit_xml(positions, times)`

Generates an `xml` file for visualizing the planetary orbits in `SolarSystemViewer` (`SSView`).

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✗	✓

Function arguments:
Let N_{time} denote the number of time steps in the orbit simulation.

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`positions`	`np.ndarray`	`(2,number_of_planets,`N_{time}`)`	AU	A `NumPy` array containing the positions of all of the planets at equidistant consecutive time steps.
`times`	`np.ndarray`	(`$N_{\text{time}}$`,)	yrs	A `NumPy` array containing the simulation time that corresponds with `positions`.

`System.send_satellite(fileName="satCommands.txt")`

Loads the satellite commands in `fileName` and executes them (For Realz).

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✓	✗

Function arguments:

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`fileName`	`str`	`()`	(N/A)	The filename for the satellite command file. For available commands see II C 1.

`System.take_picture(filename="find_orient.png")`

Take a picture using the satellite's onboard camera.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✓	✓

Function arguments:

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`filename`	`str`	`()`	(N/A)	The filename of the generated picture.

`System.test_random_number_generator()`

Runs a test to verify that your computer is able to replicate a pseudo-random-generated list of numbers. The results are printed in the terminal.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✗	✗

2. Relativity methods in alphabetical order

Shortcuts:

```
System.part2A_4(chosen_planet, friend_seed=None, increase_height=False,
filename1="part2A_4_frame1.xml", filename2="part2A_4_frame2.xml")
```

Generates the `.xml` files used in Exercise 1 in Part 8.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✗	✓

Function arguments:
Replace `X` by either `1` or `2`.

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`chosen_planet`	`int`	`()`	(N/A)	Index of your chosen target planet.
`friend_seed`	`int`	`()`	(N/A)	In case the relativity exercises take place in another project's solar system, enter their `seed`. `friend_seed=None` indicates your own `seed`.
`increase_height`	`float`	`()`	(N/A)	Indicates the distance above the surface of your target planet the relativity exercises take place. `increase_height=False` indicates a predetermined distance denoted as 1.0. Generally, $0.5 \leq$ `increase_height` ≤ 5.0. Use this option in case the spaceships interfer with the surface of the planet.
`filenameX`	`str`	`()`	(N/A)	The filename for the `.xml` file used by frame `X`.

```
System.part2A_5(chosen_planet,friend_seed=None, filename1="part2A_5_frame1.xml",
filename2="part2A_5_frame2.xml")
```

Generates the `.xml` files used in Exercise 2 in Part 8.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✗	✓

Function arguments:
Replace `X` by either `1` or `2`.

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`chosen_planet`	`int`	`()`	(N/A)	Index of your chosen target planet.
`friend_seed`	`int`	`()`	(N/A)	In case the relativity exercises take place in another project's solar system, enter their `seed`. `friend_seed=None` indicates your own `seed`.
`filenameX`	`str`	`()`	(N/A)	The filename for the `.xml` file used by frame `X`.

```
System.part2B_1(chosen_planet,friend_seed=None, filename1="part2B_1_frame1.xml",
filename2="part2B_1_frame2.xml", filename3="part2B_1_frame3.xml")
```

Generates the `.xml` files used in Exercise 4 in Part 8.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✗	✓

Function arguments:
Replace `X` by either `1`, `2` or `3`.

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`chosen_planet`	`int`	`()`	(N/A)	Index of your chosen target planet.
`friend_seed`	`int`	`()`	(N/A)	In case the relativity exercises take place in another project's solar system, enter their `seed`. `friend_seed=None` indicates your own `seed`.
`filenameX`	`str`	`()`	(N/A)	The filename for the `.xml` file used by frame `X`.

```
System.part2B_3(chosen_planet,friend_seed=None, increase_height=False,
filename1="part2B_3_frame1.xml", filename2="part2B_3_frame2.xml")
```

Generates the `.xml` files used in Exercise 5 in Part 8.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✗	✓

Function arguments:
Replace `X` by either `1` or `2`.

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`chosen_planet`	`int`	`()`	(N/A)	Index of your chosen target planet.
`friend_seed`	`int`	`()`	(N/A)	In case the relativity exercises take place in another project's solar system, enter their `seed`. `friend_seed=None` indicates your own `seed`.
`increase_height`	`float`	`()`	(N/A)	Indicates the distance above the surface of your target planet the relativity exercises take place. `increase_height=False` indicates a predetermined distance denoted as 1.0. Generally, $0.5 \leq$ `increase_height` ≤ 5.0. Use this option in case the spaceships interfer with the surface of the planet.
`filenameX`	`str`	`()`	(N/A)	The filename for the `.xml` file used by frame `X`.

```
System.part2B_4(chosen_planet,friend_seed=None, increase_height=False,
filename1="part2B_4_frame1.xml", filename2="part2B_4_frame2.xml")
```

Generates the `.xml` files used in Exercise 6 in Part 8.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✗	✓

Function arguments:
Replace `X` by either `1` or `2`.

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`chosen_planet`	`int`	`()`	(N/A)	Index of your chosen target planet.
`friend_seed`	`int`	`()`	(N/A)	In case the relativity exercises take place in another project's solar system, enter their `seed`. `friend_seed=None` indicates your own `seed`.
`increase_height`	`float`	`()`	(N/A)	Indicates the distance above the surface of your target planet the relativity exercises take place. `increase_height=False` indicates a predetermined distance denoted as 1.0. Generally, $0.5 \leq$ `increase_height` ≤ 5.0. Use this option in case the spaceships interfer with the surface of the planet.
`filenameX`	`str`	`()`	(N/A)	The filename for the `.xml` file used by frame `X`.

```
System.part2B_5(chosen_planet,friend_seed=None, increase_height=False,
filename1="part2B_5_frame1.xml", filename2="part2B_5_frame2.xml")
```

Generates the `.xml` files used in Exercise 7 in Part 8.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✗	✓

Function arguments:
Replace `X` by either `1` or `2`.

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`chosen_planet`	`int`	`()`	(N/A)	Index of your chosen target planet.
`friend_seed`	`int`	`()`	(N/A)	In case the relativity exercises take place in another project's solar system, enter their `seed`. `friend_seed=None` indicates your own `seed`.
`increase_height`	`float`	`()`	(N/A)	Indicates the distance above the surface of your target planet the relativity exercises take place. `increase_height=False` indicates a predetermined distance denoted as 1.0. Generally, $0.5 \leq$ `increase_height` ≤ 5.0. Use this option in case the spaceships interfer with the surface of the planet.
`filenameX`	`str`	`()`	(N/A)	The filename for the `.xml` file used by frame `X`.

`System.part2C_2(distance, friend_seed=None, filename=None)`

Generates the `.xml` file used in Exercise 2 in Part 9. This method will ask for a series of user-input via the command line, the details are explained in the exercise.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✗	✓

Function arguments:

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`distance`	`str`	`()`	(N/A)	Either `"far"` or `"close"`.
`friend_seed`	`int`	`()`	(N/A)	In case the relativity exercises take place in another project's solar system, enter their `seed`. `friend_seed=None` indicates your own `seed`.
`filename`	`str`	`()`	(N/A)	The filename for the `.xml` file used by frame `distance`. `filename=None` sets the filename to `"part2C_2_distance.xml"`, where `distance` is given by the function argument.

`System.part2C_5(number_of_light_signals=30, friend_seed=None, consider_light_travel=False, write_text=False, filename1="part2C_5_frame1.xml", filename2="part2C_5_frame2.xml")`

Generates the `.xml` files used in Exercise 7 in Part 9.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✗	✓

Function arguments:
Replace `X` by either `1` or `2`.

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`number_of_light_signals`	`int`	`()`	(N/A)	Sets the number of light signals sent out by the falling spaceship. The number of signals is limited to $10 \leq$ `number_of_light_signals` ≤ 100.
`friend_seed`	`int`	`()`	(N/A)	In case the relativity exercises take place in another project's solar system, enter their `seed`. `friend_seed=None` indicates your own `seed`.
`consider_light_travel`	`bool`	`()`	(N/A)	`True/False`: Include the travelling time of light in the videos.
`write_text`	`bool`	`()`	(N/A)	`True/False`: Save the time intervals between successive light signals.
`filenameX`	`str`	`()`	(N/A)	The filename for the `.xml` file used by frame `X`.

```
System.part2C_8(chosen_planet, theta=None, friend_seed=None, increase_height=False,
filename="part2C_8.xml")
```

Generates the `.xml` file used in Exercise 5 in Part 9.

Returns output	`static`	Prints to terminal	Hierarchy	Other
✗	✗	✓	✗	✓

Function arguments:

Variable name	`Python type`	`NumPy` shape	Physical unit	Description
`chosen_planet`	`int`	`()`	(N/A)	Index of your chosen target planet.
`theta`	`float`	`()`	`rad`	Angular position on the planet, which corresponds to rectangular coordinates $\mathbf{r} = (R\cos\theta, R\sin\theta)$. $0 \leq$ `theta` $\leq 2\pi$.
`friend_seed`	`int`	`()`	(N/A)	In case the relativity exercises take place in another project's solar system, enter their `seed`. `friend_seed=None` indicates your own `seed`.
`increase_height`	`list`	`()`	(N/A)	Set this to `True/[0.5,5]` if computer is running slow or there are mountains/clouds blocking your view.
`filename`	`str`	`()`	(N/A)	The filename for the `.xml` file.

3. Hierarchy of methods

With the exception of `ang2pix` and `gen_seed`, all methods require an instance of the `AST2000SolarSystem` class. The methods marked with the tag "Hierarchy" are dependent on data created in previous methods, these include `mass_needed_launch`, `take_picture`, `measure_doppler_shifts`, `analyse_distances`, `manual_orientation`, `send_satellite` and `land_on_planet`. All of these methods are dependent on values that either need to be calculated inbetween the function calls or are predetermined parameters. To avoid having to use random values, I import them using generic functions from `somewhere_else` (meaning you need to define them yourself). If you wish to run the entire satellite project in one script (which isn't required), the following is the required order of operations:

```
from somewhere_else import get_username, engine_simulation, get_launch_parameters, launch_simulation, orientation, get_target_planet
from AST2000SolarSystem import AST2000SolarSystem
username = get_username()
seed     = AST2000SolarSystem.get_seed(username)
System   = AST2000SolarSystem(seed)

dpdt, N_box, dNdt, m_fuel      = engine_simulation()
T_launch, launch_pos, t_launch = get_launch_parameters()
System.engine_settings(dpdt, N_box, dNdt, m_fuel, T_launch, launch_pos, t_launch)
final_launch_pos = launch_simulation(T_launch, launch_pos, t_launch)
System.mass_needed_launch(final_launch_pos, test=True)

System.take_picture(picture_name)
dlambda1, dlamda2   = System.measure_doppler_shifts()
dist                = System.analyse_distances()
ang, satVel, satPos = orientation(picture_name, dlambda1, dlambda 2, dist)
System.manual_orientation(ang, satVel, satPos)

System.send_satellite(filename="satCommands.txt")
p = get_target_planet()
System.land_on_planet(p, filename="landCommands.txt")
```

All of your progress is recorded in a binary file called `s.bin`, you will find it is generated the first time (or if you somehow decide to delete `s.bin` it is generated the next time) you run `mass_needed_launch`. All accomplishments and progress is recorded in `s.bin`, which means you do not need to pass a test twice.

C. Command files

1. `send_satellite` *commands*

The `"satCommands.txt"` command file should be a standard `.txt` file (or `.dat` or whatever) that contains the specific commands necessary for *sending your satellite* to the target planet. All the commands need to be placed on *separate* lines in the command file. Any command parameter should directly follow the command keyword with at least 1 space character, separate command parameters should also be separated with at least 1 space character. Unless otherwise specified, all numbers can be written in any shape or form. For example, the number `1` can also be written as `1.0` and `1e0`. Only two commands can occur simultaneously. Below is an example of a command file.

```
launch
orient 0.6
boost  0.7  5e-1  0.3
video  0.8  1
video  1.2  1
boost  1.5 -1e-1 -0.1
orient 2
```

These are the available satellite commands:

`launch`

The `launch` command initializes the satellite mission. The initial time, position and velocity is returned from `mass_needed_launch`.

`orient`

The `orient` command activates the onboard equipment, running your user-built software, and returns (prints to terminal) the satellite's current position and velocity given a time parameter. The first time the `orient` command is run you need to complete a `manuel orientation` procedure. The syntax is as follows:

```
orient time
```

where `time` is the time in years.

`boost`

The `boost` command activates an instantaneous boost, which alters the momentary velocity of the satellite. The syntax is as follows:

```
boost time vx vy
```

where `time` is the time in years and `(vx,vy)` is the change in the velocity vector $\Delta\mathbf{v} = (\Delta v_x, \Delta v_y)$ in AU/yrs.

`video`

The `video` command create `.xml` videos viewable in MCAst. There are two variants of this command: focusing on a planet and focusing in a particular angular direction. Both variants require two command lines: one for initializing the video creation and one for ending it.

The syntax for the first variant is as follows:

```
video time_start idx
video time_end   idx
```

As the variables imply, `time_start` and `time_end` defines the time span of the video in years, while `idx` is an `int` that specifies the `AST2000SolarSystem` planet index of the planet in focus.

The syntax for the second variant is as follows:

```
video time_start theta1 phi1
video time_end   theta2 phi2
```

The `time` arguments are the same, however, the angular arguments define the initial and final angular direction in which the camera points in `rad`. `theta` and `phi` refer to the same angular coordinates as introduced in the project. During the course of the video, the camera will rotate from `(theta1,phi1)` to `(theta2,phi2)` in a linear fashion.

2. `land_on_planet` *commands*

The `"landCommands.txt"` command file should be a standard `.txt` file (or `.dat` or whatever) that contains the specific commands necessary for *landing your satellite* on the target planet. All the commands need to be placed on *separate* lines in the command file. Any command parameter should directly follow the command keyword with at least 1 space character, separate command parameters should also be separated with at least 1 space character. Unless otherwise specified, all numbers can be written in any shape or form. For example, the number `1` can also be written as `1.0` and `1e0`. Only two commands can occur simultaneously. Note that time in the `land_on_planet` command file is separate from the time in the `send_satellite` command file. The last moment in the `send_satellite` command file is defined as $t = 0$ in the `land_on_planet` command file. Below is an example of a command file.

```
init
boost         5.0e4  0.0  -3.0e3  0.0
launchlander  6.0e5  0.0   4.6e3  0.0
parachute     6.5e5  35
picture       9.4e5  1.5  -3.142  0.0  0.0  1.0
landing       1.0e6  30    3.0e4
```

These are the available satellite commands:

`init`

The `init` command initializes the satellite landing. The initial time, position and velocity is loaded from the `s.bin` file using the final `orient` command in the `send_satellite` command file.

`orient`

The `orient` command activates the onboard equipment, running your user-built software, and returns (prints to terminal) the satellite's current position and velocity given a time parameter. The syntax is as follows:

```
orient time
```

where `time` is the time in seconds.

`boost`

The `boost` command activates an instantaneous boost, which alters the momentary velocity of the satellite. The syntax is as follows:

```
boost time vx vy vz
```

where `time` is the time in seconds and `(vx,vy,vz)` is the change in the velocity vector $\Delta\mathbf{v} = (\Delta v_x, \Delta v_y, \Delta v_z)$ in m/s.

`launchlander`

The `launchlander` command releases the lander unit from the satellite at the specified time. The command requires an additional velocity boost *which is relative to the satellite*. The syntax is as follows:

```
launchlander time Vx Vy Vz
```

where `time` is the time in seconds and `(Vx,Vy,Vz)` is the velocity of the lander unit relative to the satellite $\mathbf{V} = \mathbf{v}_{\text{lander}} - \mathbf{v}_{\text{sat}} = (V_x, V_y, V_z)$ in m/s.

parachute

The `parachute` command deploys the lander unit's parachute whose default surface area is 42 m^2, but can be specified using an optional argument. The syntax is as follows:

```
parachute time area
```

where `time` is the time in seconds and `area` is the optional parameter that specifies the surface area in m^2 of the parachute.

landing

The `landing` command is the final command that (hopefully) completes the landing procedure. The time argument should be chosen such that the landing has already been completed. This command also gives you access to a landing engine that can help smooth the impact. The syntax is as follows:

```
landing time distance thrust
```

where `time` is the final time in seconds, `distance` is the height over ground in meters that the landing engine switches on and `thrust` is the thrust force in Newtons with which the landing engine operates.

video

The `video` command creates `.xml` videos viewable in MCAst. There are two variants of this command: focusing on a planet and focusing in a particular angular direction. Both variants require two command lines: one for initializing the video creation and one for ending it.

The syntax for the first variant is as follows:

```
video time_start idx
video time_end   idx
```

As the variables imply, `time_start` and `time_end` defines the time span of the video in seconds, while `idx` is an `int` that specifies the `AST2000SolarSystem` planet index of the planet in focus.

The syntax for the second variant is as follows:

```
video time_start theta1 phi1
video time_end   theta2 phi2
```

The `time` arguments are the same, however, the angular arguments define the initial and final angular direction in which the camera points in `rad`. `theta` and `phi` refer to the same angular coordinates as introduced in the project. During the course of the video, the camera will rotate from `(theta1,phi1)` to `(theta2,phi2)` in a linear fashion.

picture

The `picture` command creates `.xml` pictures viewable in MCAst. A picture is taken at a specific time and defined using an *up-direction* for the camera and an angular direction in which the camera points. The *up-direction* is given as a rectangular three-dimensional vector. If the vector is set to the satellite's/lander's current position vector, then the picture will be taken with a "natural view" along the horizon of the planet. The angular direction is specified in a similar fashion to the second variant of the `video` command. The syntax is as follows:

```
picture time theta phi x y z
```

where `time` is the time of the picture in seconds, `(theta,phi)` is the angular direction (same angular coordiantes as the ones introduced in the project) in which the camera points in `rad`, and `(x,y,z)` is the rectangular vector that defines the *up-direction* of the camera.