Project 3
Applied Informatics: Two Colonies For Mankind

The deadline for submitting this project is the **25th January, 2024, 11:59 a.m.** (noon).

In the year 2026, the two billionaires Elom Nusk and Jeb Fezos started colonizing the Moon. In compliance with NASA regulations, they agreed to collaborate by sharing a common satellite in a low Moon orbit, which will be responsible for relaying data from both sites back to Earth. Before they agreed, they imposed the following condition: Both parties have equal access to the satellite, i.e. the amount of data relayed via the satellite must be the same per party.

You are responsible for providing connection schedules for both parties with the satellite fulfilling the aforementioned constraint. At first, you will model the situation in STK, extract the possible connection times, and programmatically determine a valid schedule as defined before.

1 Submission Instructions

You are required to work on this project *in groups of two students*, i.e. your *team* that you registered in the dCMS. Should you be unable to do so (for example because your partner drops out), but want to participate in the project, you must contact us via a message to `@Assistants` in our discussion board by the 21st December, 2023, 11:59.

We will provide your team with a Git repository on the chair's GitLab instance. We will consider the latest commit pushed to the `main` branch before the deadline mentioned above as your final submission.

2 Preparations

2.1 Python

During this project, you will use Python to interface with STK [1, 2]. Your code must be executable under Python 3.12, which itself is already installed in our VM. You can download Python here (or using a package manager such as *Chocolatey* or *WinGet*).

After downloading the skeleton (Section 3), install the STK Python bindings by running the following command (provided STK is installed at the default location). Please refer to the manual installation instructions for more details.

```
python -m pip install "C:\Program Files\AGI\STK 12\bin\AgPythonAPI\agi.stk12-12.7.1-py3-none-any.whl"
```

Caution: You can use any functionality provided by the *Python Standard Library*. In general, you **must not** use any third-party library, except these installed above and the one being described and installed in Section 2.2.

2.2 Gurobi

For the scheduling tasks of this project, we use Gurobi Optimizer, a commercial solver for *Mixed Integer Linear Programming* problems. In order to use Gurobi, you need to install it first (we recommend to not use the VM here): Visit the Gurobi download page and register for an *Academic* account using your university e-mail address (or just login). Accept the licence agreement, download and install *Gurobi 11.0.0*.

Configure Gurobi to use the university's academic licence by creating a text file in your home directory (e.g. `~/gurobi.lic` on Linux / macOS or `C:\Users\Astronaut\gurobi.lic` on Windows) with the following content:

```
TOKENSERVER=lizenzserver.hiz-saarland.de
PORT=46325
```

Caution: You need to be connected to the university network to use the Gurobi licence server, either directly or using a VPN connection at home.

Before you can use Gurobi from Python, install the Gurobi Python bindings:

```
python -m pip install gurobipy==11.0.0
```

2.3 Tutorial

Please complete the *Interfacing STK with Python* tutorial before you start:
`https://help.agi.com/stk/index.htm#training/StartPython.htm`

3 Project Skeleton

You should have access to a repository containing the project skeleton by now. Should you think this is not the case, send us a message to **@Assistants** in our discussion board as soon as possible.

3.1 Structure

The skeleton is grouped in five directories:

- `spain` – Contains functions provided by us, e.g. an STK wrapper simplifying the interaction with STK.
- `impl` – Your implementation goes here. More details in the following.
- `docs` – Your documentation goes here. More details in the following.
- `lib` – Some additional files are available here, e.g. default access files for the scheduling.
- `out` – Output files generated during runtime are placed here.

In addition, the following important files are available in the project root:

- `colonize.py` – The main file and therefore the project entry point. See Section 3.2 for information on how to run the project.

Caution: You **must not** change any file outside the `impl` or `docs` directories.

3.2 Command-line Interface

In general, the project is executed with:

```
python colonize.py task{1|2|3}
```

In addition, the skeleton is highly parametrized, allowing it to set various parameters for the scenario itself. However, all parameters come with a default value.

For an overview of the available parameters, run:

```
python colonize.py --help
```

In addition, each task may introduce additional parameters, and an overview can be obtained with:

```
python colonize.py task{1|2|3} --help
```

The command-line arguments are globally available as `namespace`. Just use

```
from spain import namespace
```

Each argument can be obtained by calling the property on `namespace`, e.g. `namespace.base_A`.

3.3 STK Wrapper

The `spain.stk` package contains a wrapper around STK, which takes care of connecting to STK properly. This wrapper is available as `STK`, and can be imported by

```
from spain.stk import STK
```

The wrapper exposes a single property `STK.root`, which corresponds to the `Root` property seen in the tutorial.

Caution: Do not attempt to connect to STK on your own. This can create all kind of trouble.

3.4 Utilities

We provide you with some useful functions in the `spain.util` module. You may or may not use them.

4 Task 1 – Scenario Initialization and Finding Best Orbit

```
python colonize.py task1 [--gui]
```

Before you start, you need to create and initialize a scenario. After adding all required objects, you will determine the best orbit for the Moon satellite.

Caution: You have to name the described objects exactly as stated here.

Hint: The STK UI is deactivated by default. You can enable it using the `--gui` flag.

4.1 Creating a New Scenario

Implement the following in the `initialize_scenario` method in `impl/task1.py`.

At first, you must create a new scenario called *MoonColonisation*. The first property to initialize is the analysis period: Both the start and stop time are passed as parameters, with the following properties: In the code, they are available as `datetime` objects, while on the command line, they can be specified in the ISO format accepted by the datetime package. You have to convert them to a compatible format before passing them to STK.

4.1.1 Adding Colonies

Add two *Targets* on the Moon called *MoonBaseA* and *MoonBaseB*. The geodetic positions of the colonies are passed as parameters and can be accessed as `base_A` and `base_B`, respectively.

4.1.2 Adding Ground Stations

To communicate back with the people left behind on Earth, the satellite uses NASA's *Deep Space Network*. In particular, the three big antennas located in Goldstone, near Madrid, and near Canberra will be used. These are available in the object database provided by STK, which, however, is difficult to access from Python. We therefore provide you with the required data to add the *Facilities* manually: the name and the geodetic position, as given in Table 1.

4.2 Finding Best Orbit

It is well known, however, that only a few inclinations in Moon orbit are stable enough to guarantee a long-time for a satellite. These are 27, 50, 76 and 86 degrees [3]. It is important for the two parties to determine which of these can provide larger data collection throughout the analysis period.

Add four satellites named *MoonSat27*, *MoonSat50*, *MoonSat76*, and *MoonSat86* to the scenario. Do so by implementing the `add_moon_satellite` method in `impl/task1.py`. Apply the following settings:

- The satellite is propagated using a *TwoBody* propagator.
- The initial state configuration is available as parameters (except for the inclination).

Propagate each satellite.

Add a *simple conic* sensor called *MoonSatAntenna* using *fixed* pointing type for each satellite. The *pointing angle*, *yaw*, *pitch*, and *roll* are passed as parameters. Define a *range constraint* for the *maximum* distance, which again is passed as a parameter.

Implement the `find_best_inclination` method in `impl/task1.py`. The method should calculate which of the four satellites can collect the most data and return the inclination which you determine the best given all parameters.

The MoonSat can communicate with the two colonies with a data rate of 100 kbit/s, but it requires 20 s of synchronization to establish the RF channel during which no data can be transmitted. Note that the 20 s of synchronization needs to happen at the beginning of the access window (when the colony is in sight), not before.

Table 1: Locations of DSN Ground Stations.

Name	Latitude	Longitude	Altitude
`DSS_14_Goldstone_STDN_DS14`	35.425 901	−116.889 538	1001.39
`DSS_43_Tidbinbilla_STDN_DS43`	−35.402 424	148.981 267	688.867
`DSS_63_Robledo_STDN_DS63`	40.431 21	−4.248 009	864.816

4.3 Extracting Access Intervals

Once the orbit is decided, extract the access intervals of the satellite with the two ground stations and the Sun.

Implement the `compute_access_intervals` method in `impl/task1.py`. The method should query STK to compute the access intervals of the Moon satellite with the two ground colonies as well as compute the sunlight windows. You should return a triple of three lists containing all access intervals for the respective colony or insolation period. Intervals are represented as tuples holding two `datetime` objects: one representing the start time and one representing the end time of the interval.

Hint: The skeleton renamed the satellite you chose to *MoonSat* and removed the others from the scenario.

Hint: The results you produce here are exported to the disk. This allows you to run the remaining tasks on systems where STK is not available, e.g. outside the VM.

Caution: VM Users

The Windows licence in the virtual machine will **expire on 7th January, 2024** as expected. If you use the VM, we recommend that you complete Task 1 before this date. Tasks 2 and 3 do not require STK and can therefore be solved without the VM on all common operating systems.

Due to the low demand and the high effort of setting up a new VM, we are not currently planning to do so. If you need STK after this date, we kindly ask that you continue to work on the task on your team member's machine. In case you and your team member require the VM, please contact us via a message to **@Assistants** in our discussion board.

Resources for Task 1

[1] Ansys, Inc. *STK API for Python Documentation.* URL: `https://help.agi.com/stkdevkit/index.htm#python/pythonIntro.htm`.

[2] Ansys, Inc. *STK Library Reference.* URL: `https://help.agi.com/stkdevkit/index.htm#ReferenceLibrary.htm`.

[3] A. S. Konopliv et al. 'Recent Gravity Models as a Result of the Lunar Prospector Mission'. In: *Icarus* 150.1 (2001), pp. 1–18. ISSN: 0019-1035. DOI: `10.1006/icar.2000.6573`.

5 Task 2 – Scheduling

As the satellite can only connect to one of the stations at a time, a schedule for the intervals is required.

5.1 LP Formulation

Solve the scheduling problem using linear programming. Try collecting as much data as possible, while not violating the fairness constraint between the two colonies: The collected data per colony should not differ by more than ε kbit (note the unit). Assume that all intervals are non-splittable. This means that you can only decide to schedule the whole interval, not just a fraction of it. Observe that the fairness constraint can always be satisfied, in the worst case by the trivial solution that does not schedule any accesses. As before, assume that 20 s of synchronization are needed at the beginning of each scheduled access interval to establish the RF channel. Only after these 20 s data can flow between the satellite and colony with a data rate of 100 kbit/s.

Define and explain your linear program in `docs/LP2.md`. Give a simple example where a solution that does not consider the synchronization delay would yield a different solution than your solution.

5.2 Implementation

```
python colonize.py task2 [--epsilon <ε>]
```

Implement the scheduling problem in Python using linear programming [4]. Implement the `find_schedule` method in `impl/task2.py`. The method takes a Gurobi model and the access intervals between the satellite and the two ground colonies as parameters. You should return a tuple consisting of two lists, each containing the scheduled intervals for connecting to the respective colony.

Explain your solution by updating `docs/LP2.md` as necessary, i.e. describe your changes and explain in detail why they were necessary.

6 Task 3 – Keeping an Eye on the Battery

Now, consider a simple linear battery model in the *MoonSat*. Adapt the proposed schedule to guarantee that the battery is *always* higher than a given state of charge. Explain how the fairness and data collection metrics are affected by the new constraint.

The satellite can either charge the battery using its solar cells or use the available energy while connecting to one of the ground colonies. For simplicity, we assume that sending data back to Earth is 'free', i.e. does not consume any energy (and thus can be ignored). In fact, the satellite either recharges the battery or uses energy to connect to the colonies (this includes the 20 s of synchronization, which is assumed to have the same energy consumption as during transmission). Otherwise, the charge remains constant (i.e. no background load).

6.1 LP Formulation

Solve the scheduling using linear programming. Again, try collecting as much data as possible, while not violating the fairness constraint between the two colonies. In addition, your schedule must guarantee that the battery charge does not fall below a given threshold at any time.

Define and explain your linear program in `docs/LP3.md`.

6.2 Implementation

```
python colonize.py task3 [--charge <c>] [--epsilon <ε>] [--initial <c>] [--load <i> <o>]
```

Implement the `find_schedule_with_battery` method in `impl/task3.py`. The method takes a Gurobi model and the access intervals of the satellite with the ground stations and the Sun. You should return a triple consisting of three lists: the scheduled intervals for connecting to the respective colony, as well as the selected intervals for charging the battery.

As before, explain your solution by updating `docs/LP3.md` as necessary.

We highly recommend you to test your solution on some small, handcrafted test scenarios.

Resources for Tasks 2 & 3

[4] Gurobi Optimization, LLC. *Gurobi Python Documentation.*
URL: `https://www.gurobi.com/documentation/11.0/refman/py_python_api_overview.html`.

A Command Line Arguments

```
usage: colonize.py [-h] [--access path/to/access/folder] [--start date] [--stop date] taskN ...

options:
  -h, --help            show this help message and exit
  --access path/to/access/folder
                        folder where the access intervals should be stored or loaded from
  --start date          scenario start time
  --stop date           scenario stop time

tasks:
  taskN                 project task to be executed
    task1 (t1)          task 1 - create scenario and determine best orbit
    task2 (t2)          task 2 - schedule fair access intervals
    task3 (t3)          task 3 - keep an eye on the battery
```

```
usage: colonize.py task1 [-h] [--gui] [--scenario path/to/scenario/folder]
                         [--baseA lat lon alt] [--baseB lat lon alt] [--sma a] [--ecc e]
                         [--aop ω] [--raan Ω] [--ta ν] [--ang α] [--rng d] [--ypr y p r]

options:
  -h, --help            show this help message and exit
  --gui                 show STK GUI
  --scenario path/to/scenario/folder
                        location where the scenario should be stored

bases:
  parameters for the moon colonies

  --baseA lat lon alt   geodetic position of the first moon base
  --baseB lat lon alt   geodetic position of the second moon base

satellite:
  parameters for the moon satellite

  --sma a               semi-major axis [km]
  --ecc e               eccentricity
  --aop ω               argument of perigee [deg]
  --raan Ω              right ascension of the ascending node [deg]
  --ta ν                true anomaly [deg]

antenna:
  parameters for the moon satellite's antenna

  --ang α               cone angle [deg]
  --rng d               maximum range [km]
  --ypr y p r           antenna pointing (yaw, pitch, roll) [deg]
```

```
usage: colonize.py task2 [-h] [--epsilon ε]

options:
  -h, --help   show this help message and exit
  --epsilon ε  ensure fairness up to ε kbit
```

```
usage: colonize.py task3 [-h] [--epsilon ε] [--charge c] [--initial c] [--load in out]

options:
  -h, --help     show this help message and exit
  --epsilon ε    ensure fairness up to ε kbit

battery:
  parameters for the satellite's battery

  --charge c     threshold for battery charge [%]
  --initial c    initial battery charge [%]
  --load in out  load for charging and draining the battery [%/s][1]
```

[1] In the code, the load factors are available as `charge_rate` and `discharge_rate`.

B Excursus: Linear Battery Model

As the linear battery model was not presented in too much detail in the lecture, we give a more formal definition here to help you with Task 3.

In a nutshell, the linear battery model views a battery as one well of capacity *cap* that is decreased proportionally to a load l that is imposed on the battery.

Both the capacity *cap* and the current state of charge *soc* are usually expressed in some unit of energy (e.g. mJ). Alternatively, the *soc* can be described as a percentage (%) of the total capacity (for simplicity, this is the case in this project).

The energy consumption of a task is typically given by some load l in mW = mJ/s or, alternatively, in %/s.

Suppose the current state of charge of a battery is soc_0, and it is used to power a task with load l for a duration of t seconds. The state of charge soc_t after t seconds can be computed as follows:

$$soc_t = \max\{0,\ \min\{soc_0 - t \cdot l,\ cap\}\}$$

The min and max operations ensure that the state of charge can never fall below zero or exceed the capacity of the battery.

The same formula can also be used to model the effect of recharging the battery. By convention, energy consuming tasks have a positive load, while a recharging task is modelled with a negative load value.