satdatagen: a Python Library for Satellite Sensor Task Scheduler Support

by

Adina H. Golden

B.S. Electrical Engineering and Computer Science, MIT, 2022

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2024

© 2024 Adina H. Golden. This work is licensed under a CC BY-NC-ND 4.0 license.

The author hereby grants to MIT a nonexclusive, worldwide, irrevocable, royalty-free license to exercise any and all rights under copyright, including to reproduce, preserve, distribute and publicly display copies of the thesis, or release the thesis under an open-access license.

Authored by: Adina H. Golden
Department of Electrical Engineering and Computer Science
August 9, 2024

Certified by: Hamsa Balakrishnan
Professor of Aeronautics and Astronautics, Thesis Supervisor

Accepted by: Katrina LaCurts
Chair, Master of Engineering Thesis Committee

This research was sponsored by the Department of the Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000, and by NASA grant #80NSSC23M0220. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Department of the Air Force, NASA or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

satdatagen: a Python Library for Satellite Sensor Task Scheduler Support

by

Adina H. Golden

Submitted to the Department of Electrical Engineering and Computer Science
on August 9, 2024 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

The number of objects in Earth's orbit is increasing rapidly, raising urgency for intensified observations of satellites and other resident space objects (RSOs) to manage space traffic and prevent collisions. Current methods for RSO detection and tracking rely on ground-based and space-based observatories with optical or radar sensors, but these telescopes require complex scheduling to achieve surveillance of all objects. Previous works have implemented scheduling algorithms and machine learning models that optimize the assignment of tasks to the sensors for RSO observations. However, prior methodologies rely on different datasets, making it hard to make comparisons across methods. This paper presents `satdatagen`: a software package that generates datasets that can be used as inputs to sensor task schedulers. The datasets generated from the `satdatagen` library are intended to be used as a baseline input to satellite sensor task schedulers. The datasets contain information about every satellite that passes in view of the sensor such as its angle of altitude and its brightness. Additionally, actual cloud cover data is included for optical telescopes that need to take visibility into account while scheduling observations. `satdatagen` is simple to use, and does not require excess outside knowledge from developers of scheduling tools.

Thesis supervisor: Hamsa Balakrishnan
Title: Professor of Aeronautics and Astronautics

Acknowledgments

I am so grateful to everyone I have crossed paths with during my year back at MIT to complete my master's degree. The following people made this experience especially rewarding.

Thank you to Professor Hamsa Balakrishnan for welcoming me into her lab and supporting me through this project. I could not have completed this thesis without her trust and guidance. I am grateful to have had opportunity to do research outside of the EECS department, and to have been able to learn so much about space domain awareness and space traffic management.

A huge thank you to my graduate student mentor Sydney, who devised this project and allowed me to turn it into a master's thesis. They supported me throughout my research and taught me everything I know about astrodynamics and satellite management. They answered my many questions and kept me accountable for staying on top of writing deadlines. I am very grateful for their patience, guidance, and encouragement during my master's work.

I would also like to thank the rest of the DINaMo Group for welcoming me into the lab and always being friendly faces in Building 31.

I have immense appreciation for my roommates and friends from this year back in Cambridge. They celebrated my accomplishments with me, supported me through rough patches, and were always there to give me valuable advice when I needed it.

The biggest thank you goes to my family. I am grateful to have been raised in a family that values education and encourages me to learn. Thank you to my mother and stepfather for pushing me to challenge myself and for believing in me when I got into MIT. Thank you to my grandparents for supporting me and always being interested in my research. A shout-out to my brother, who gladly bought me lunch every time his pilot's schedule flew him to Boston. A shout-out to my sister, who checks in on me weekly and whose kids are the cutest in the whole world.

Finally, thank you to MIT for accepting me as an undergraduate 6 years ago, and to the EECS department for accepting me again as a graduate student. I will always appreciate that I had the opportunity to receive not one but two degrees from this institution.

Contents

Title page 1

Abstract 3

List of Figures 9

List of Tables 11

1 Introduction 13

2 Related Works 15
- 2.1 Task Schedulers . . . 15
- 2.2 Related Python Libraries . . . 16

3 Methods 19
- 3.1 Determining Overhead Satellites . . . 19
  - 3.1.1 Two-Line Elements . . . 19
  - 3.1.2 Propagation . . . 20
  - 3.1.3 Coordinate Transformations . . . 20
- 3.2 Determining Object Brightness . . . 23
  - 3.2.1 Assumptions . . . 24
  - 3.2.2 Krag Method . . . 26
  - 3.2.3 Hejduk Method . . . 26
  - 3.2.4 Molczan Method . . . 27
  - 3.2.5 Evaluation and Validation . . . 27
- 3.3 Data Collection . . . 28
  - 3.3.1 Retrieving TLEs . . . 28
  - 3.3.2 Retrieving Object Sizes . . . 28
  - 3.3.3 Finding Cloud Cover Information . . . 29

4 Results 31
- 4.1 satdatagen Library Structure . . . 31
  - 4.1.1 Installation and Integration . . . 31
  - 4.1.2 Dependencies . . . 32
  - 4.1.3 Classes . . . 33
- 4.2 Dataset Generation Time . . . 34

4.3 Example Scenario 36

5 Conclusions **39**
5.1 Future Work 39

A Code listing **41**
A.1 Example Dataset Code 41
A.2 generate_dataset 41

B List of Dependencies **45**

List of Figures

2.1 The network of sensors/telescopes employed by the United States Space Force to upkeep their Space-Track database. Dedicated sensors operate only for SDA missions, collateral sensors support other missions in addition to SDA, and contributing sensors operate for SDA missions when needed. Figure from [1]. 16

3.1 The phase angle is defined as the angle between the Earth, satellite, and sun. 24
3.2 Object classes and their quantities as defined by the ESA's Database and Information System Characterising Objects in Space 25

4.1 Flowchart of the satdatagen computation process 32
4.2 Selection of 500 satellites that pass over the Haystack Observatory (lat=42.58, lon=-71.44) during a 30 minute period (06-18-2024 20:00-20:30 UTC). 37

List of Tables

4.1 TimeRange Class Constructor Inputs. Parameters in brackets are optional to instantiate a TimeRange object. 33
4.2 Dataset Generation Times 35
4.3 Individual Process Component Times to Generate a Dataset with 500 Satellites 35
4.4 Parameters to Create TimeRange Object 36
4.5 Parameters to Create GroundLocation Object 36
4.6 Example Overhead Pass of Satellite 28661 37

Chapter 1

Introduction

The population of objects in space is increasing rapidly. The space research community works to maintain space domain awareness (SDA) by surveying these resident space objects (RSOs) orbiting Earth and keeping an updated catalog of their whereabouts in order to prevent collisions and manage space traffic. As of 2022 there were over 25,000 catalogued objects in Earth's orbit, and estimates predict an additional 58,000 satellites will be launched by 2030 [1, 2]. The increase in density of objects in Earth's orbit will result in more collisions, creating debris that further increases collision risk for other objects. SpaceX, manufacturer of the large Starlink satellite constellation, reported that they have had to perform almost 50,000 maneuvers in the past year alone to avoid collisions with debris and other space junk [3]. Just in June 2024, during the writing of this thesis, a conjunction occurred that produced over 100 new RSOs, and countless other debris objects too small to be tracked [4]. Although there are attempts to mitigate debris with satellite re-entry procedures, the European Space Agency (ESA) in their 2024 Space Environment Report noted that the current efforts still lead to an "unsustainable environment in the long term" [5]. The current infrastructure for tracking RSOs and dispatching conjunction warnings, which consists of ground-based and space-based observatories, is overwhelmed by the growing density of objects in space [2].

We focus on the use of ground-based sensors and telescopes for tracking RSOs, although space-based sensors or star trackers already attached to satellites are an area of future work [6]. The ground-based sensors are a mix of optical telescopes, which mostly track objects in low Earth orbit (LEO), and radar telescopes, which can reach deep space and objects in geosynchronous Earth orbit (GEO).

The main catalog of satellite tracking data is maintained by the 18th Space Defense Squadron (18th SDS) of the United States Space Force [7]. The 18th SDS has a network of ground-based sensors, most of which are stationed on the northern hemisphere. Only one radar sensor, capable of observing objects in deep-space, and only two optical telescopes are located on the southern hemisphere. Their lack of geographic diversity makes it difficult to keep track of RSOs that travel primarily over the southern hemisphere in their orbit [1]. The space community schedules the use of the ground-based sensors to account for their limited quantity and geographic placements in order to track and catalog almost every object orbiting the Earth. Sensor tasking and optimization algorithms aim to improve the schedules of the observatories to maximize the surveillance of orbiting RSOs.

This thesis introduces `satdatagen`, a Python library that can generate datasets of satel-

lites for sensor task schedulers. Optimization methods for ground-based sensor tasking have been proposed using reinforcement learning techniques and mixed integer linear programming. When evaluating these methods there is no common input dataset of active satellites available for fair comparison. `satdatagen` is a platform that enables researchers to create these datasets by providing a location on Earth and a time range during which the observations should take place. The dataset generated contains the overhead satellites and their locations during the inputted observation time, as well as their brightness and the cloud cover probability.

This thesis aims to aggregate the relevant data needed for sensor task scheduling and disseminate it in a simple format through the `satdatagen` Python package. First, we describe recent task scheduler methods and other Python software packages made for astronomers. Then, we explain the underlying astrodynamics of our model, how we determine whether objects will be above an observatory, and how we calculate object brightness as well the assumptions made based on previous works thereon. Next, we discuss how we collected our data, and then we present the `satdatagen` library structure and its dependencies. Finally, we evaluate an example dataset created using the `satdatagen` library.

Chapter 2

Related Works

2.1 Task Schedulers

Efficient scheduling of sensors in a satellite-tracking network, particularly that of the 18th SDS, is important to maintain space domain awareness and to monitor space traffic for collision prevention. Tasking algorithms create schedules for sensors and sensor networks to optimize RSO tracking. In this paper we will explore two tasking approaches that optimize the satellite tracking process; the first formulates the optimization as a vehicle routing problem, while the other uses reinforcement learning.

In satellite sensor tasking, the scheduler decides which satellites the sensor shall observe. Optimizing this schedule can be formulated as a vehicle routing problem, as done with the Deep Space Radar Scheduler (DSRS) in [8]. Blanks focuses on automatic, centralized scheduling of the Space Force's network of deep-space radar telescopes, whose schedules are traditionally done independently and by hand. The DSRS minimizes cost, in particular the time needed to compute observations, the time to move between observations, and the expense of utilizing the sensors, while also being able to adapt quickly to sensor outages or other operational scenarios. The DSRS was shown to have reduced slew time (the time needed for the sensor to move between observations) and to have reduced the cost of operations compared to alternative scheduling algorithms across many numbers of tasks. While developed for deep-space sensors, the structure of the DSRS can be translated to other satellite sensor tasking problems by revising the input network of telescopes to account for those that track objects in other orbits.

Other approaches to satellite sensor tasking employ reinforcement learning (RL) to create a schedule [9]. They explore using single- and mulit-agent RL to optimize the task schedules, where each agent is a space-based sensor and they make decisions based on predefined policies and feedback from a reward function tailored to maximize or minimize a certain metric, such as the number of RSOs observed. Their multi-agent experiment utilizes several agents who work independently to assign tasks to their sensors, with a governing scheduler in charge of keeping track of which tasks have already been completed. The data inputted to evaluate the the single- and multi-agent systems and different reward functions are randomly generated RSOs using Monte Carlo simulations. Their RL systems performed better than the baseline myopic, greedy agents that only consider instantaneous reward, and were robust to increasing

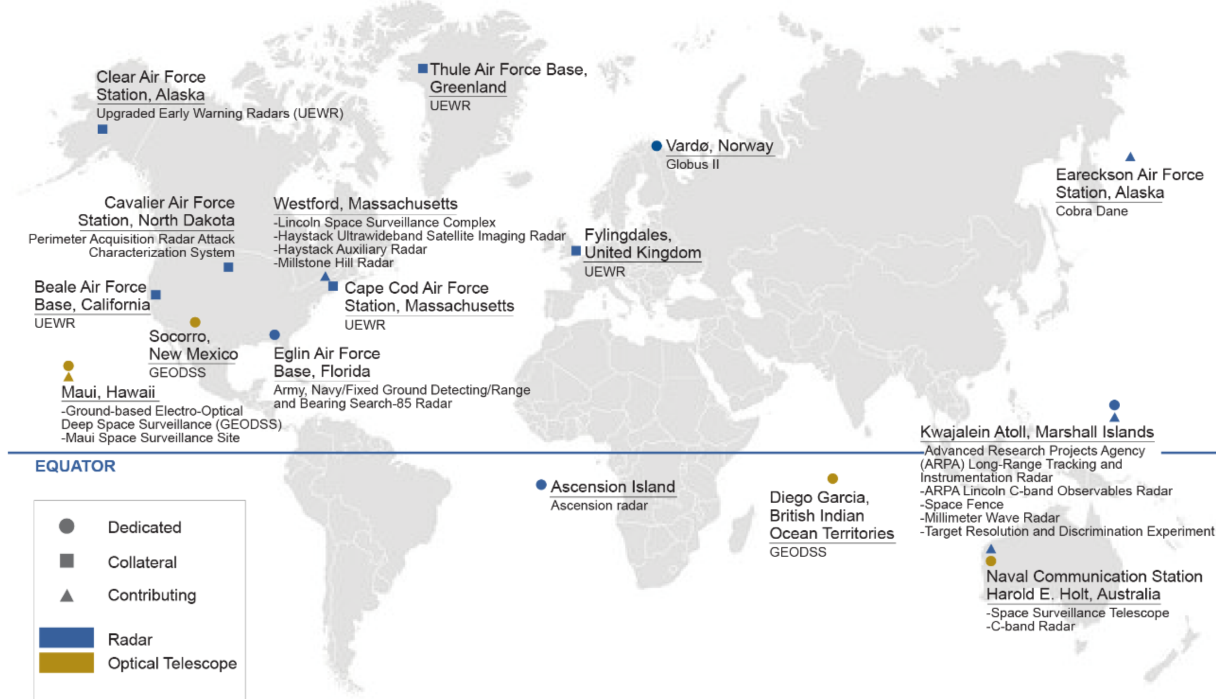

Figure 2.1: The network of sensors/telescopes employed by the United States Space Force to upkeep their Space-Track database. Dedicated sensors operate only for SDA missions, collateral sensors support other missions in addition to SDA, and contributing sensors operate for SDA missions when needed. Figure from [1].

numbers of tasks.

Although the DSRS and the RL approach examine sensor tasking for ground-based and space-based sensors, respectively, their optimization and training can be applied to ground-based networks of telescopes. To directly compare their scheduling capabilities, we can employ `satdatagen` to generate a dataset of satellites that would need to be observed over a period of time. Using this dataset as an input to both the DSRS as designed in [8] and the RL single- and/or multi-agent system as designed in [9], we can then evaluate which method can schedule the most observations of RSOs, as well as their computational performance.

2.2 Related Python Libraries

There are many existing Python libraries to assist with astronomical research, some of which we rely on for `satdatagen`'s functionality. The packages of note are `astropy` [10] and `poliastro` [11], both of which provide straightforward Python application programming interfaces (APIs) for astronomy applications.

The Astropy Project was first developed in 2013 [10]. Inspired by other libraries such as `numpy` or `scipy`, which catered towards scientists, `astropy` is catered towards astronomers. The package provides a structure to easily convert between units, time scales, and coordinate

frames. Users would have to perform tedious calculations and implement complicated algorithms themselves to achieve this common functionality without a dedicated library. The `astropy` library and its associated sublibraries are primarily written in Python, relying on existing packages for certain features, with some additional code written in C. Additionally, `astropy` has integrated `numpy` arrays into its class structure, so most `astropy` objects are easily manipulated using common `numpy` functions.

Another Python library is `poliastro` [11], which focuses on astrodynamics applications and provides an API for visualizing orbits. `poliastro` is intended for a similar audience as `astropy`, but its functionality focuses more on studying the movements of space objects. The `poliastro` library uses `astropy.units`, `astropy.time`, and `astropy.coordinates` modules to control unit conversions, time objects, and coordinate frames in their high-level interface. This allows users to interact with a similar API to what they may be used to with the existing `astropy`, while the lower-level layer – the Core API – of `poliastro` deals with pure Python and `numpy` types in its algorithms as passed from the high level API to improve performance. Some algorithms implemented in the Core API include propagation algorithms, which are used to calculate an object's position in the future based on its orbit and other motion parameters. However, `poliastro` does not have the Simplified General Perturbations #4 algorithm implemented (explained more in Section 3.1.2), which is the propagation algorithm employed by `satdatagen`, and therefore we do not use `poliastro` in our Python library.

Our `satdatagen` library is similar to `poliastro`. We provide a fully Python interface for astronomers which relies on existing data structures from the Astropy Project to perform necessary conversions and transformations. `satdatagen`, however, is a domain-specific package, made for the use-case of satellite sensor task schedulers. While users could also analyze satellite movement, our package's primary purpose is for creating datasets to evaluate these schedulers.

Chapter 3

Methods

This chapter explains our methodology for assembling the datasets generated by `satdatagen`. These datasets include the RSOs that are overhead of the sensor during the period of observation. First we describe how we determine which RSOs are viewable from the sensor during the observation time, starting with how we propagate them along their orbit and then how we transform the coordinates to calculate their angles of elevation. Next we explain three methods to calculate the brightness of an object and the assumptions we make about the physical properties of the satellites. Finally, we present the sources from which we collect data on satellite location and size, and cloud cover.

3.1 Determining Overhead Satellites

The objective of the `satdatagen` library is to generate datasets for use as inputs to satellite task schedulers. The satellites included in these datasets are those that pass overhead of the sensor of interest. We consider any object to be overhead whose angle of altitude with respect to the Earth location of the observatory exceeds 10°, explained further in Section 3.1.3. The following subsections explain how we extract the positions of satellites and how we propagate them in time to the desired time range for scheduling, before calculating their altitude.

3.1.1 Two-Line Elements

In order to determine the locations of RSOs we require information regarding their orbits and recent positions, which is collected in the form of Two-Line Elements (TLEs). From a TLE, we extract coordinates for the objects' positions and velocities, and use these to determine when objects will pass over the observatory.

A TLE is a data structure consisting of two strings, or lines, with 69 characters each. The lines are encoded with multiple data values relating to an object's motion and use spaces as delimiters. The information in a TLE communicates an object's position and direction of motion along its orbit during the time of observation (epoch), which allows us to predict where the object will be in the future [12].

The 18th Space Defense Squadron (18th SDS), our source for object TLEs, creates the

TLEs by observing the objects with their network of sensors and telescopes, and uses the Simplified General Perturbations #4 (SGP4) model to quantify the objects' motion [7]. The SGP4 model defines the algorithms and astrodynamic assumptions to use when formulating an object's position from the values encoded in the TLE and propagating it along its orbit, explained further in Section 3.1.2 [13].

TLEs from the 18th SDS are updated about every 8 hours and uploaded to space-track.org. We retrieve the TLEs used in `satdatagen` from the space-track website, requesting for the TLEs created on the date of interest specified by the user. Because up to three TLEs will be returned in each request, we filter for the TLE that was created closest in time to the start time of the user's time range. Our method of retrieving these TLEs from space-track.org is explained in more detail in Section 3.3.1.

3.1.2 Propagation

The Simplified General Perturbations #4 (SGP4) model is used to create TLEs and predict future positions of satellites by propagating them along their orbits. There are other models for satellite position prediction which handle calculating for variations in the object's motion and orbit such as "incorporation of resonances, third-body forces, atmospheric drag, and other perturbations" differently than SGP4 [13]. However, the 18th SDS recommends SGP4 as the sole propagation model to use on their TLEs to produce the most accurate results [14]. The astrodynamics equations for extracting position and velocity coordinates from data encoded in a TLE, and FORTRAN code to solve these equations, are explained in more detail in [15]. In 2008, Vallado et al. published a followup to the original SGP4 paper which standardizes the SGP4 propagation algorithm and provides the code for it in C++, FORTRAN, MATLAB, and Pascal.

In the `satdatagen` library, we use the `sgp4` package for satellite propagation, which compiles the standard C++ code from [13] into Python. This is the library `poliastro` recommends for propagating satellites from TLEs. The `sgp4` library also has functionality to propagate several RSOs at once through the same time range using its `SatrecArray` class. We utilize this class in `satdatagen` to quickly propagate thousands of RSOs forward in time in parallel.

3.1.3 Coordinate Transformations

Once we know the positions of satellites at the desired observation time, we need to verify that they will be in sight of the observatory. In order to tell whether an object is overhead of a position on Earth, we need to find its coordinates relative to the observatory location on Earth and calculate its angle of altitude (sometimes referred to as elevation). The altitude describes how far above or below the horizon an object is, with an angular value ranging from $-90°$ to $90°$. A positive altitude means the object is above the horizon and possibly visible (depending on brightness and weather conditions), with $90°$ being directly overhead. A negative altitude means the object is hidden below the horizon. We consider any object having an altitude of greater than $10°$ to be sufficiently above the horizon to be visible.

We perform several transformations on the coordinates evaluated from the TLEs to get an RSOs angle of altitude. The SGP4 algorithm extracts position and velocity coordinates

of an object in the True Equator Mean Equniox (TEME) coordinate frame. TEME is an Earth-centered inertial (ECI) frame whose origin lies at the center of the Earth. ECI frames are static and do not rotate along with the Earth. After rotations and translations, we convert the object's TEME coordinates to coordinates in an Earth-fixed topographical frame to perform the altitude calculation with respect to the observatory's ground location. This new frame rotates along with Earth, and the origin lies on the surface of the Earth at the latitude and longitude of the observatory.

The Python library `astropy`, as described in [10], provides functionality to execute these transformations, which we employ in the `satdatagen` source code. The rest of this section will explain the basic rotations and translations required to convert TEME coordinates to an Earth-fixed topographical frame for calculating altitude, while more information on the algorithms and physics behind these calculations can be found in [16, 10].

South-East-Zenith

We convert the RSOs' coordinates to the Earth-fixed, topographical South-East-Zenith (SEZ) frame. From a location on the surface of the Earth, in our case from the observatory, the $\hat{S}$ vector points due south, the $\hat{E}$ vector points due east, and the $\hat{Z}$ vector is orthogonal to the $\hat{S}$-$\hat{E}$ plane, pointing away from the Earth's surface. A simple trigonometric relationship between an object's coordinates and its altitude makes the SEZ frame especially convenient:

$$\sin(alt) = \frac{\rho_Z}{\rho} \tag{3.1}$$

where alt is the angle of altitude, ρ is the vector from the observatory to the object, and ρ_Z is the $\hat{Z}$ component of the vector.

Once we have the objects' coordinates in SEZ, we find their angles of altitude with ease using Equation 3.1 and determine which are visible (assuming brightness and good weather) using our constraint of 10°. However, we first perform several rotations to convert from TEME, an ECI frame, to an intermediate Earth-centered Earth-fixed (ECEF) frame defined by the International Terrestrial Reference System (ITRS), and then finally to SEZ. These rotations account for the wobbly nature of the Earth's spin about its axis and the conversion from an inertial frame to a fixed frame (with respect to the Earth).

We need to know the coordinates of the observatory in ITRS in order to move the origin from the center of the Earth to the surface. Therefore the intermediate transformation of the observatory's and the object's coordinates to ITRS is necessary for translating the origin and then rotating to the SEZ frame.

TEME to ITRS

The Earth's spin about its axis is affected by the other celestial bodies in the Solar System, whose gravitational influences cause the equator and poles to shift over time. We also have to consider where along its rotation the Earth will be at the time for which we convert the coordinates, which is known as sidereal time. We use the Prime Meridian as a reference for determining sidereal time, then rotate again to the observatory's longitude. In total

we perform four transformations by applying rotational matrices to the TEME coordinates to convert them to ITRS. These transformations are time-dependent and account for the following motions, as described in [17] and [16]:

1. **Precession (P)**: the large-scale motion of the polar axis of rotation, caused by gravitational forces of the other planets.
2. **Nutation (N)**: the quasi-periodic variability of the pole with respect to precession, caused by the moon exerting a torque on the Earth due to its bulging at the equator.
3. **Sidereal Rotation (R)**: the amount of rotation the Earth has completed in its period.
4. **Polar Motion (W)**: the small, unpredictable variation in the direction of the polar axis, which is measured empirically.

We apply these rotations as matrix multiplications to the TEME coordinates in Equation 3.2.

$$r_{\vec{ITRS}} = \left[\mathbf{W}(t)\right]\left[\mathbf{R}(t)\right]\left[\mathbf{N}(t)\right]\left[\mathbf{P}(t)\right] r_{\vec{TEME}} \tag{3.2}$$

Where $r_{\vec{TEME}}$ is the 3-dimensional vector of the object's TEME coordinates, $r_{\vec{ITRS}}$ is the final 3-dimensional vector of the object's coordinates in the ITRS frame, and t is the time at which the coordinates are transformed.

The precession and nutation matrices are built by considering the positions of the sun, moon, and other planets of the solar system with respect to the Earth and calculating their gravitational effects on the Earth's rotation at the time of transformation. More detail on this computation can be found in [16]. The sidereal matrix is determined by the Greenwich Mean Sidereal Time, θ_{GMST}, which defines longitudinal angle of the Greenwich (Prime) Meridian with respect to the TEME celestial frame. The $\hat{I}$ vector of ITRS frame points towards the Prime Meridian.

To calculate θ_{GMST} we convert our desired observation time to its Julian Date, defined as the number of days since the Julian period with midnight on January 1, 2000 having a Julian Date of 2451545.0. We can then use the equation developed by Simon Newcomb to find the θ_{GMST} in degrees [16]:

$$\theta_{GMST0^h} = 100.4606184 + (36000.77005361)T + (0.00038793)T^2 - (2.6x10^{-8})T^3 \tag{3.3}$$

Where T is the number of Julian days corresponding to the observation date at midnight, or the truncated Julian Date in integer form. To find the precise Greenwich Mean Sidereal Time for the time of observation we adjust the θ_{GMST0^h} found in Equation 3.3 with the Earth's mean angular rotation in degrees per seconds and the remaining time from the pre-truncated Julian date in seconds. The precise θ_{GMST} is found with Equation 3.4:

$$\theta_{GMST} = \theta_{GMST0^h} + \omega * T \tag{3.4}$$

Finally, the polar motion rotational matrix is determined by the polar motion coordinates x_p and y_p. The International Earth Rotation and Reference Systems Service measures and

publishes these coordinates daily [18]. More detail on the formation of the sidereal and polar motion rotation matrices can be found in [16] and [17].

We use the `astropy.coordinates` module of the `astropy` library to perform the transformation from TEME to ITRS coordinates with the `transform_to` method. We do not re-implement any code or algorithms for building the rotation matrices and solving Equation 3.2 in `satdatagen`, because the Astropy project uses the same methods as detailed in [16] for its computations [10].

ITRS to SEZ using `astropy`

Converting from ITRS to SEZ involves a translation and rotation of coordinates. We translate the ITRS coordinates to the surface of the Earth where the observatory lies, then we rotate the system so the $\hat{I}$ vector points due south, the $\hat{J}$ vector points due east, and the $\hat{K}$ vector points away from Earth's surface. We convert the latitude and longitude of the observatory location to ITRS coordinates by creating a `EarthLocation` object from the `astropy.coordinates` module. Then we call the `get_itrs` method, which takes as an argument the time of observation, to get the observatory's coordinates.

$$\vec{r}_{topo} = \vec{r}_{RSO} - \vec{r}_{observatory} \tag{3.5}$$

Equation 3.5 performs the translation to get the coordinates of the object in a topographical reference frame. The next step would be to rotate the system to SEZ, as we described above, and then solve for the altitude. The `astropy` library allows us to complete both steps at once by applying the `transform_to` method on the topographical coordinates of the object, and specifying the observatory's `EarthLocation` object, the observation time, and the altitude/azimuth destination format as input parameters. This final transformation returns the altitude of the object, which we check against our constraint to see if it could be visible.

The Astropy Project integrated their library with the `numpy` package, which means `astropy` objects can be manipulated the same as `numpy` arrays. `numpy` has the benefit of performing quick, parallel computations on very large arrays. We take advantage of this feature in `satdatagen` by storing the TEME coordinates of all objects in a multi-dimensional array and computing the altitude for all of them in parallel. This technique allows us to avoid `for` loops, which add a lot of computation time for large arrays in Python.

3.2 Determining Object Brightness

`satdatagen` provides users with information on object brightness at the time of observation. Task schedulers for optical sensors rely on whether or not an RSO is visible to the sensor to decide to schedule an observation for that object. We use the apparent visual magnitude (AVM, M_v) as a metric of object brightness. AVM follows a reverse logarithmic scale, with brighter objects having lower magnitudes. We calculate AVM by relating the magnitude of the main light source, the sun ($M_v = -26.78$), to the size and reflectivity of the object and the phase angle. The phase angle is defined as the angle between the sun, the satellite, and the Earth, illustrated in Figure 3.1. Because the phase angle depends on the position of the

RSO, it is necessary to calculate AVM for every satellite that crosses into the field of view of the observer during every observation time-step.

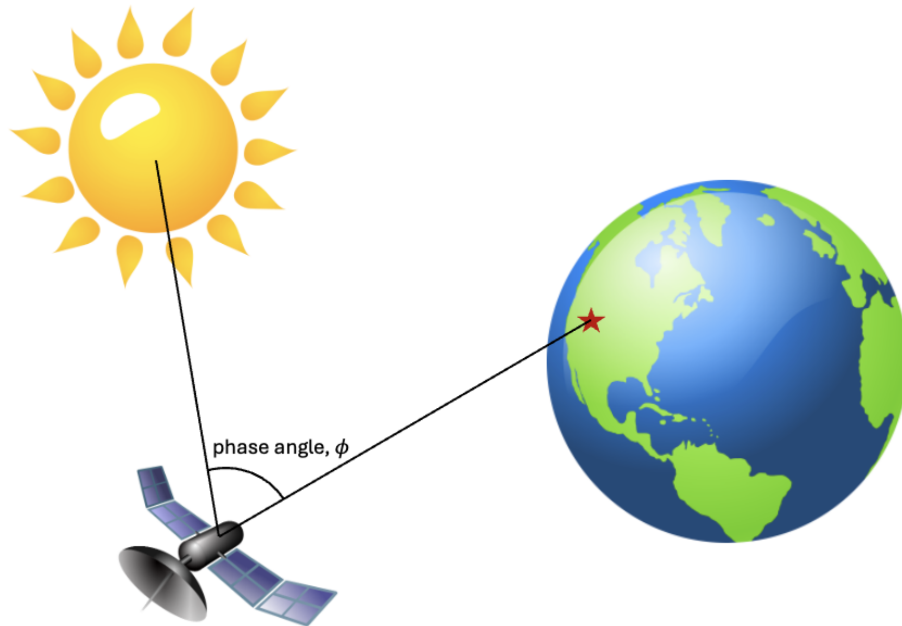

Figure 3.1: The phase angle is defined as the angle between the Earth, satellite, and sun.

In this thesis we discuss three methods for calculating AVM: Krag's [19], Hejduk's [20], and Molczan's [21]. We consider these three methods because they provide simplifications for calculating AVM while still preserving relative accuracy. Users of `satdatagen` can choose which approach to use to calculate AVM for their datasets. The most precise method of determining AVM is by imaging, but it's infeasible at the current time to measure the brightness of every object visually. An object's brightness can differ by up to 4 magnitudes depending on the position of the sun, so we provide AVM calculations for every time step in the time range defined by the user [22]. The user of our Python package is able to choose which of the three methods to employ when determining AVM for the objects in their dataset.

3.2.1 Assumptions

Before discussing the Krag, Hejduk, and Molczan methods for calculating AVM, we will detail the assumptions we make about RSOs and their physical properties.

Spherical Objects

The amount of visible surface area of an object contributes significantly to its brightness. Object dimensions, shape, and orientation, which are factors in determining surface area, are often not published nor easily accessible – especially for debris, which account for a significant portion of RSOs, as notable in Figure 3.2. Therefore, we make the general assumption that all objects take a spherical shape when applying the three AVM calculation methods. This

assumption allows us to ignore uncertainties regarding object orientation. For example, when viewing a cylindrical rocket payload, the observer could see a flat plane or a rounded surface, or some combination of both. The flat plane of the cylinder could exhibit vastly different light reflections than the rounded surface by virtue of its material and orientation with respect to the sun. Conversely, a spherical object would have uniform reflection regardless of its orientation with respect to the sun and Earth. Determining an object's orientation with respect to the Earth is outside the scope of this work.

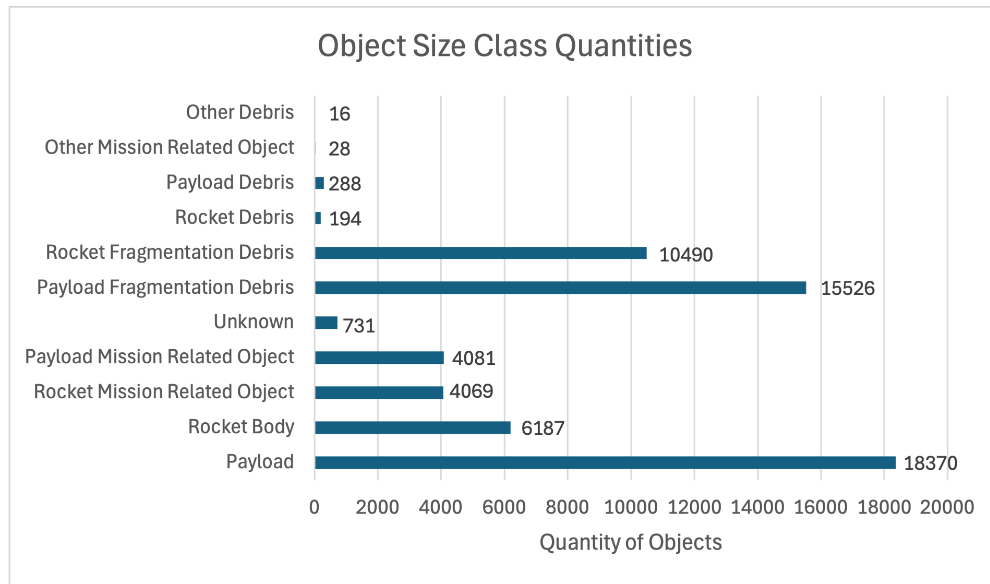

Figure 3.2: Object classes and their quantities as defined by the ESA's Database and Information System Characterising Objects in Space

Radar Cross-Section (RCS)

The approximate size or surface area of an object is determined by analyzing a distribution of radar cross-section (RCS) data. NASA developed the Size Estimation Model (SEM) to map RCS data to an estimated spherical diameter (ESD) for an RSO [23]. The ESD corresponds to the cross-sectional area of the sphere that would have produced the same reflected energy that was measured by the radar [24]. We assume the spherical cross-sectional area from the RCS data is the same as the surface area of the RSO that is visible to the observatory when calculating AVM.

Reflectivity

An object's material affects its reflectivity, or albedo. As with the size and orientation of the object, the reflectivity also requires observations and measurements to precisely determine, which is outside the scope of this project. We assume an albedo coefficient of $\rho = 0.1$ or 10% reflectivity, which accounts for satellite manufacturers' efforts to reduce material brightness and produced accurate AVM values, explained further in Section 3.2.5 [25].

3.2.2 Krag Method

Krag's method of calculating AVM was published in 1974, taking inspiration from similar work done by McCue et al. in 1971 [19, 26]. Krag's approach to determine the AVM of an object uses the magnitude of the sun as a reference value, and then corrects for its shape, size, reflectivity, distance from the observer, and phase angle.

$$M_v = -26.78 - 2.5 * \log\Big(\frac{\rho A * F(\phi)}{R^2}\Big) \tag{3.6}$$

In Equation 3.6 M_v is the AVM value, A represents the surface area of the object, and ρ is the reflectivity coefficient. The phase angle correction is performed by inputting the phase angle, ϕ, into a phase function, $F(\phi)$. The phase function applied in Equation 3.6 models the object as a diffuse sphere. Other phase functions account for object shape and whether the reflection off an object is diffuse or specular. Diffuse reflection means that the light scatters when reflected and the surface appears matte, whereas for specular reflection the light is reflected directly at an angle, and the surface of the object appears reflective like a mirror. For example, a sphere whose surface is totally diffuse has the phase function:

$$F_d(\phi) = \frac{2}{3\pi^2}[(\pi - \phi)\cos\phi + \sin\phi] \tag{3.7}$$

,

while a sphere with totally specular reflection has the constant phase function:

$$F_s(\phi) = \frac{1}{4\pi} \tag{3.8}$$

.

While Krag's method introduces the possibility for objects with different reflective materials, it assumes that an object has either totally diffuse or totally specular reflection. When calculating AVM using Krag's method in the `satdatagen` library, we assume that the reflection off the object is totally diffuse and solve for AVM using Equation 3.6.

3.2.3 Hejduk Method

The second method we have to calculate AVM for an object is Hejduk's [20]. The Hejduk method is similar to Krag's in principle, but it allows for an object's surface to have both specular and diffuse reflection components. Many satellites, such as those in the SpaceX Starlink fleet, have large solar arrays attached to an inner bus [27]. The material of the solar arrays will reflect light specularly, while the material on the bus may scatter light more. Hejduk's equation, shown below in Equation 3.9, allows us to account for the reflections from the many materials on objects in space.

$$M_v = -26.78 - 2.5 * \log\Big(\frac{\rho A * [\eta F_d(\phi) + (1-\eta) F_s(\phi)]}{R^2}\Big) \tag{3.9}$$

Equation 3.9 introduces a mixing coefficient, η, which controls the percentage amount of phase angle correction for diffuse reflection. The remaining percentage is assumed to be the

specular reflection component, so the phase function for specular reflection is also included in the equation.

The exact amount of specular or diffuse reflection for each object is unknown, but Hejduk found that most rocket bodies and debris are more diffuse than specular, so we assume an η of 0.8 or 80% diffuse response [20]. Users of `satdatagen` who choose to use the Hejduk method to calculate AVM of objects can change the default η value as they see fit.

3.2.4 Molczan Method

Finally, we detail Molczan's approach for calculating AVM. This method was developed by amateur astronomer Ted Molczan, who reformatted the apparent magnitude, used to measure brightness of stars, for satellite applications. He assembled a catalog of values he calls the intrinsic magnitudes, M_{90° which represent the average brightness of objects as they travel through their orbits [21, 28]. The intrinsic magnitude is defined as the object's brightness or AVM when it is 1000 km away and is at a phase angle of 90°. To calculate an object's instantaneous AVM, Molczan uses Equation 3.10 which corrects for the object's actual phase angle and distance from the observer at the time of observation. These adjustments are referred to as the phase correction and the distance correction, respectively.

$$M_v = M_{90^\circ} + PhaseCorrection + DistanceCorrection \tag{3.10}$$

The equation used for phase correction is shown below in Equation 3.11, and the distance correction in Equation 3.12.

$$PhaseCorrection = -2.5\log(\sin\phi + [\pi - \phi]\cos\phi) \tag{3.11}$$

$$DistanceCorrection = 5\log(|\vec{r}_{los}|) - 15 \tag{3.12}$$

The intrinsic magnitude is measured when the object is at a 90° phase angle with the observer and the sun, so the maximum orientation correction is 90° because the phase angle will never exceed 180°. More likely only smaller corrections are necessary, because objects with large enough phase angles can be eclipsed by the Earth and completely in shadow, no longer visible.

Molczan's equations produced accurate AVM values (discussed more in Section 3.2.5, but the catalog does not include intrinsic magnitude M_{90° values for many satellites currently in orbit. In the case where we do not have an M_{90° for an object, we use Krag's method to estimate the M_{90°, inputting a phase angle of 90° and a distance of 1000km to Equation 3.6. Then we continue with the phase and distance corrections as in Equation 3.10 to compute an AVM value.

3.2.5 Evaluation and Validation

We compared their results of the Krag, Hejduk, and Molczan equations against each other as an initial evaluation. For satellites whose M_{90° intrinsic magnitude is in Molczan's catalog, we estimated their brightness as they passed overhead of Westford, Massachusetts, where

the Haystack Observatory is located. We also assumed $\eta = 0.8$ for the Hejduk equation. Most AVM values produced by the three methods were within 1 magnitude of each other.

We also validated the three methods to compute AVM against results from the website in-the-sky.org. In-the-sky is an amateur astronomy project developed by Dominic Ford. Users can input their location and in-the-sky will produce a list of the registered satellites that pass overhead of the user along with their brightness during the passes. We compared the results of our AVM calculations from Krag's, Hejduk's, and Molczan's equations against the results listed on in-the-sky (whose AVM determination method is unknown) and found them to be within 2 magnitudes of each other.

Users of `satdatagen` creating datasets for optical telescopes can compare the AVM values with the sensitivity of their sensor to discern whether objects will be visible.

3.3 Data Collection

3.3.1 Retrieving TLEs

We retrieve the TLEs from the 18th Space Defense Squadron by interacting with the application programming interface (API) of their space-track.org (space-track) website via Hypertext Transfer Protocol (HTTP) requests. We use the Python library `requests` to send an HTTP GET request to the space-track.org server, which queries their databases for the desired satellite TLEs. We send the GET request using a URL formatted as specified in the space-track API.

This request returns a list of JavaScript Object Notation (JSON) objects, one for each TLE, sorted in order of the NORAD identification number given to every object at launch/discovery. Because there can be multiple observations of an RSO per day, the query may return multiple TLEs for an RSO. The `satdatagen` user provides a time range for which they will run their sensor task scheduler. We search through the request results to find only the TLEs that were recorded to have been observed closest to and before this time range.

Space-track requires that anyone seeking data through their API create a login to space-track.org. This requirement allows them to limit the amount of requests users can make at one time so as not to overwhelm their servers. Users of `satdatagen` therefore need to create space-track.org login credentials for themselves as a prerequisite to generating datasets with the library.

3.3.2 Retrieving Object Sizes

Knowledge of the sizes of RSOs is needed for accurate AVM calculations. To protect their designs, major commercial manufacturers like SpaceX often do not publish the dimensions of their satellites. Additionally, collision-borne RSOs and other debris may have never been on Earth, so their sizes also need to be determined remotely. As described in Section 3.2.1, we use the average cross-sectional area as computed from Radar Cross Section measurements to assume an object's surface area as seen from Earth.

Average RCS data with units meters squared for most RSOs is published in the ESA's Database and Information System Characterising Objects in Space (DISCOS). Using their

DISCOSweb API, we send HTTP GET requests to retrieve the RCS sizing data and other basic information for every object in their database. This includes the minimum, maximum, average cross-sectional areas as estimated by RCS scanning, and the classification of the object as made by the ESA. The ESA classifies objects based on their purpose, with classes such as "Payload" or "Rocket Fragmentation Debris." The full list of object classes defined by the ESA can be seen in Figure 3.2 in Section 3.2.1. Unfortunately there are some objects in the DICSOS whose approximate sizes are entirely unknown or unmeasured. In order to still provide an approximate AVM value for these unmeasured RSOs, we take the average size of each ESA object class and assign the RSO the corresponding average size. Because object sizes are not expected to change often, nor are they measured often, we only perform this data collection one time and store the results in a Python dictionary that we read from during AVM calculations.

3.3.3 Finding Cloud Cover Information

`satdatagen` provides users with historical cloud cover information so that sensor task schedulers, particularly for optical sensors, can account for scenarios with outages or a sensor being offline due to low visibility. Our cloud cover data is retrieved from the Open-Meteo API, an open-source free weather API [29]. They collect and aggregate weather data from several datasets of the European Centre for Medium-Range Weather Forecasts. The cloud cover information comes from the ERA5 dataset, which provides hourly data and updates daily with a delay of five days, but has data available going back to 1940 [30]. The cloud cover data from the ERA5 dataset represents the "proportion of a grid box covered by cloud," and will be a value between 0 and 1.

If a user desires accurate weather information, they must choose their time range for their generated dataset to be more than five days prior to the current date. The cloud cover data we receive only has resolution to the hour, so each time we propagate the objects to is rounded to the nearest hour to estimate its cloud cover.

Chapter 4

Results

In this chapter we share the structure of `satdatagen` datasets and the information contained therein. We then explain the programming interface and the Object Oriented Programming (OOP) principles we applied to simplify use of `satdatagen` for developers. Next, we discuss the time it takes to generate datasets of varying sizes, and finally we create an example dataset of 500 satellites and their passes over the Haystack Observatory in Westford, Massachusetts.

4.1 satdatagen Library Structure

`satdatagen` builds datasets to be used as inputs to satellite sensor task schedulers. Users of our Python library input information about their sensor and the duration for which they will schedule its tasks, and `satdatagen` outputs a dataset in the form of a Python dictionary or a .JSON file.

The dataset includes information about satellites that pass overhead of the sensor or observatory during the time range of interest. Satellite data includes its name, satellite ID number, AVM, angle of altitude when it passes over, and the TLE used to propagate the satellite to that location. Additionally, the time at which the satellite passes overhead is included as a timestamp, and the probability of cloud cover over the ground location is included in the dataset. We format the dataset as a dictionary so that the information is accessible through simple Python commands, as well as easily translated to JSON to be stored in a .JSON file.

An example scenario of generating a dataset is described in section 4.3, as well as more detail about the contents of the dataset. The remainder of this section describes the usage and structure of the `satdatagen` software library. The entire source code and further usage instructions for `satdatagen` can be found at github.com/ahgolden/satdatagen.

4.1.1 Installation and Integration

The `satdatagen` package is accessible through the Python Package Index (PyPI), Python software repository that hosts over 500,000 projects [31]. Most custom Python libraries,

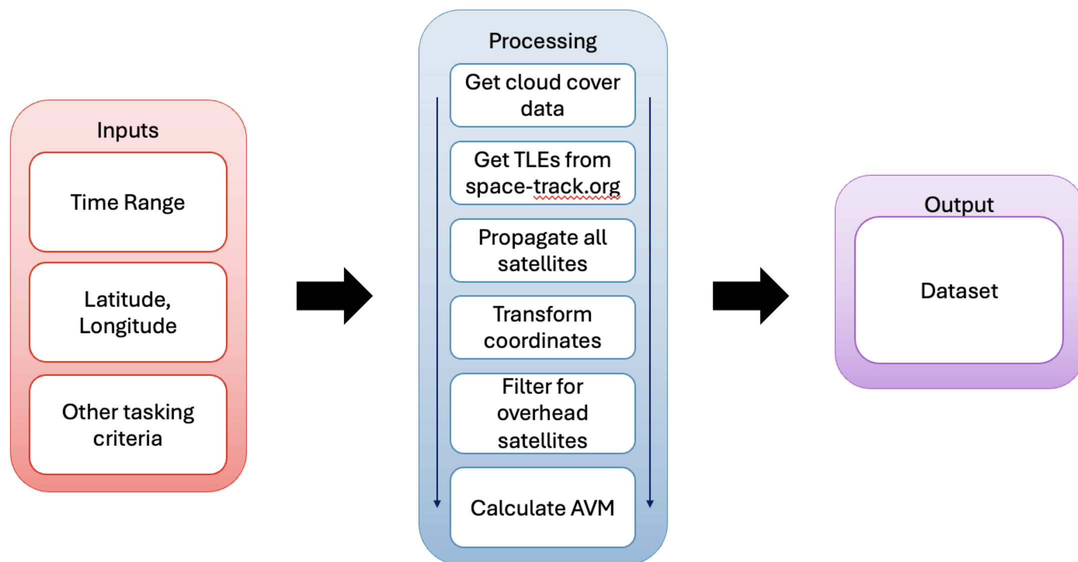

Figure 4.1: Flowchart of the satdatagen computation process

including `astropy` and `poliastro`, are hosted on PyPI. PyPI simplifies software package integration by enabling installations with a single command.

Users install `satdatagen` with the command `python3 -m pip install satdatagen`. After installation, `satdatagen` can be integrated into any Python project by including the command `import satdatagen` at the top of the Python file.

4.1.2 Dependencies

The `satdatagen` library is an open-source package that is distributed through the Python Package Index. The package relies on other Python libraries such as the built-in `pickle` and `json` modules, as well as `numpy`, `requests`, and domain-specific packages like `astropy` and `sgp4.api`. The full list of dependencies can be found in Appendix B. If these packages are not already in the user's current Python environment, they will be installed automatically upon installation of the `satdatagen` library.

The main package `satdatagen` relies upon is the `astropy` library, which contains three main modules: `astropy.coordinates`, `astropy.time`, and `astropy.units`. The module `astropy.coordinates` includes functionality for performing coordinate transformations, most importantly from the TEME frame to the SEZ frame for determining objects' angles of altitude, as explained in 3.1.3. The `astropy.time` module defines data structures for time values. These data structures allow for easy conversions between different formats and scales. For example, Equations 3.3 and 3.4 require a Julian Date. We call `astropy.time` functions in `satdatagen` to compute the Julian Date from a time inputted in the Coordinated Universal Time (UTC) scale. The `astropy.units` module handles assigning units to values, and supports implicit conversions between compatible units.

4.1.3 Classes

Users of `satdatagen` employ object-oriented programming (OOP) principles to generate their datasets. The two primary object types are `TimeRange` and `GroundLocation`.

TimeRange

A `TimeRange` object defines the duration for which the telescopes will be scheduled, holding the time steps from start to end. The time steps are stored as a `numpy` array of `astropy.time Time` objects in the `TimeRange.times` variable. This list later gets passed as an input parameter into `astropy.coordinates` functions.

Table 4.1 shows the input parameters and their data types required to instantiate a `TimeRange` object. Parameters in brackets are optional inputs.

Table 4.1: TimeRange Class Constructor Inputs. Parameters in brackets are optional to instantiate a TimeRange object.

Parameter	Description	Data type
`start_date`	The date and time in UTC to begin the sensor tasking scheduling.	`datetime`, `str`
`periods`	The number of time-steps to include in the time range.	`int`
`[end_date]`	The end date and time in UTC of the sensor tasking scheduling period.	`datetime`, `str`
`[delta]`	The time in minutes between each time-step in the time range	`int`

The `start_date` and `end_date` parameters can either be strings (of type `str` in Python) or `datetime` objects. The strings must be formatted in ISO Date Time Format, which takes the form `yyyy-MM-ddTHH:mm:ss`. The Python `datetime` library is commonly known among developers, and is compatible with `astropy.time Time` objects, so we have made `TimeRange` compatible with its objects as well. The constructor converts the start and end dates to `astropy.Time` objects and creates the intermediate time steps upon instantiation. We constrain the `start_date` and `end_date` input types to be `str` or `datetime` instead of `astropy.time Time` objects to make `satdatagen` more accessible to Python developers who are unfamiliar with the `astropy` library.

GroundLocation

A `GroundLocation` object holds the information for where the sensor/observatory lies on Earth. The constructor accepts the latitude and longitude as type `float` of the desired

location, as well as a `TimeRange` object representing the duration of observation. The latitude and longitude are stored as an `astropy.coordinates EarthLocation` object, which is needed to perform topographical coordinate transformations as described in 3.1.3.

The `GroundLocation` class is the foundation for generating a dataset. Once users instantiate a `GroundLocation` object they can call the class method `generate_dataset` to have `satdatagen` generate a dataset. `generate_dataset` takes several optional parameters to allow users to customize their sensor task scheduling dataset to their needs. These optional parameters are:

- `avm_method`: users input their choice of which method to use to calculate AVM of the objects as described in Section 3.2. Requires a string of either `'krag'`, `'hejduk'`, or `'molczan'`. The default is Krag's method.
- `limit`: users specify the maximum number of RSOs to be included in their dataset. The default is 1000.
- `orbit`: users specify whether they would like to filter their dataset to only include RSOs in a certain orbit. Requires a string of either `'LEO'` for low Earth orbit, `'MEO'` for medium Earth orbit, `'GEO'` for geosynchronous Earth orbit, or `'all'` for no filter. The default is no filter applied.
- `output_file`: users can input the path to a .JSON file to which the dataset will automatically be saved. Requires a string of a file path. The default value is `None`, the dataset will not be saved to any file.

4.2 Dataset Generation Time

In this section, we will evaluate how long it takes `satdatagen` to generate a dataset, and which parts of the process take the most time to compute.

Since users can choose to limit the number of satellites included in their datasets, we ran tests over several sample sizes. We chose the number of satellites to include in the datasets based on the problem sizes tested on the DSRS, a sensor tasking scheduler we described in 2.1. The problem sizes tested on the DSRS represent the number of observation tasks assigned to the sensors per session, in other words: the number of satellites that needed tracking for that period.

As expected, it takes longer to generate larger datasets than it does to create smaller datasets. We recommend that users limit the number of satellites to include in their dataset to reduce the time for generation. If all satellites that pass overhead of the observatory are represented, the data set could take several minutes to generate. While 45 seconds seems like a significant amount of time for a program to run, it adds a negligible amount of time to the sensor task scheduling process. According to [8], the DSRS took at most 168 minutes to generate a schedule for 140 tasks, and at least 34 minutes to generate a schedule for 200 tasks. `satdatagen` allows for an automated process to create input datasets for the DSRS and other schedulers, while adding insignificant time to the overall procedure.

Table 4.2: Dataset Generation Times

Task size	Time (s)
80	13.21
100	14.52
120	15.47
140	15.56
200	17.56
260	20.22
500	30.53
1000	45.38

Furthermore, we can break down the dataset generation into the different parts shown in the flowchart in Figure 4.1, and evaluate the timing of each step in the process. We measured the times on a dataset that was generated with 500 satellites to better emphasize the components that take the most time to compute. The amount of time for each step to create a dataset of 500 satellites is shown below in Table 4.3.

Table 4.3: Individual Process Component Times to Generate a Dataset with 500 Satellites

Step	Time (s)
Get cloud cover	0.58
Get TLEs	8.36
Propagate satellites	0.23
Transform coordinates	1.62
Filter for overhead satellites	0.04
Calculate AVM	17.28
Other processing	1.01
Total	**29.12**

The steps that consume the most time are calculating the AVM and retrieving the TLEs from space-track.org. Other steps, such as propagating the satellites or transforming their coordinates, take significantly less time because their computations are able to be done in parallel for each satellite. This is attributable to `numpy` arrays being especially efficient with element-wise operations, even with large dimensions. The AVM calculations, however, are done in series because we need to retrieve the size of each satellite from the dictionary which stores the RCS data from DISCOS. We must loop over an array of over 500 elements, search through the dictionary for the objects' sizes, and perform the computations to get the AVM.

The database that we query from on space-track.org is their historical General Perturbations database, which contains all recorded TLEs dating back several decades. We request thousands of lines of data from this huge database, so our query takes longer to produce results. However, the response time from space-track is not dependent on the number of satellites requested in the `satdatagen` generated database, so as more satellites are represented this query time becomes less significant.

4.3 Example Scenario

We now present the process of generating an example dataset using `satdatagen`. We will create a dataset to schedule the radar telescopes at Haystack Observatory, located in Westford, Massachusetts. With only limited lines of code we can create a dataset and save it to a .JSON file for easy access from other scripts. We use the following parameters listed in Table 4.4 to create a `TimeRange` object as defined in the `satdatagen` library.

Table 4.4: Parameters to Create TimeRange Object

Parameter	Value
`start_date`	`'2024-06-18T18:00:00'`
`periods`	24
`delta` (minutes)	30

This `TimeRange` is defined to start on June 18th, 2024 at 18:00 UTC, and the satellites are propagated forward in time by 30 minutes 24 times, which means our time range spans 12 hours in total. Objects in low Earth orbit (LEO) have periods as small as 90 minutes, but we expect a 30 minute propagation period to be able to catch the RSOs that pass overhead despite their fast speed. The Haystack Observatory houses the Haystack Ultrawide Satellite Imaging Radar, which can function regardless of cloud cover or night-time visibility. If users are creating datasets for optical sensor tasks, they should ensure that their time range goes through the night for best satellite brightness and visibility.

Next, we input this `TimeRange` object into the constructor of a `GroundLocation` object, along with the other necessary parameters such as latitude and longitude of the Haystack Observatory.

Table 4.5: Parameters to Create GroundLocation Object

Parameter	Value
`space_track_credentials`	`'/path/to/space-track/credentials.JSON'`
`lat`	42.58
`lon`	-71.44
`time_range`	see `TimeRange` object above

Now that we have our `GroundLocation` object instantiated, we can run the method `generate_dataset`. As described in Section 4.1.3, the `generate_dataset` class method has several optional parameters the user can input to tune the generated dataset for their precise scheduling problem. For example, a sensor that has a slow slew rate would not be able to complete an assignment that includes over 200 tasks. Or, as with the DSRS, some radar sensors are equipped to observe only objects in deep-space, and some optical sensors can only observe objects in LEO. Users can also filter to generate datasets that only include objects in LEO or deep-space. These choices to limit or filter the results are made as inputs to the `GroundLocation` class method `generate_dataset`.

We limit our sample dataset to include 500 satellites, with no filtering for orbit. Each pass that a satellite makes overhead of the observatory is documented in the dataset. The

code used to generate this example dataset can be found in Appendix A. An example pass of satellite with ID 28661 and its corresponding data is listed in Table 4.6 below.

Table 4.6: Example Overhead Pass of Satellite 28661

Key	Value
Name	BREEZE-M DEB (TANK)
Time	2024-06-19T01:30:00
Altitude	`dms_tuple(d=17.0, m=2.0, s=39.56494867818378)`
Azimuth	`dms_tuple(d=180.0, m=56.0, s=16.23134670843683)`
TLE Line 1	`1 28661U 05019C   24170....0000-0  11275-2 0  9993`
TLE Line 2	`2 28661  49.3845 233.617....7470  4.06498674263572`
AVM	9.804157640770711
Cloud Cover	30.30

Figure 4.2 illustrates the ground tracks made by the 500 satellites that pass over the Haystack Observatory in our example dataset from 20:00 to 20:30 UTC on June 18, 2024. The lines on the map are a projection of the satellites' orbits onto Earth's surface.

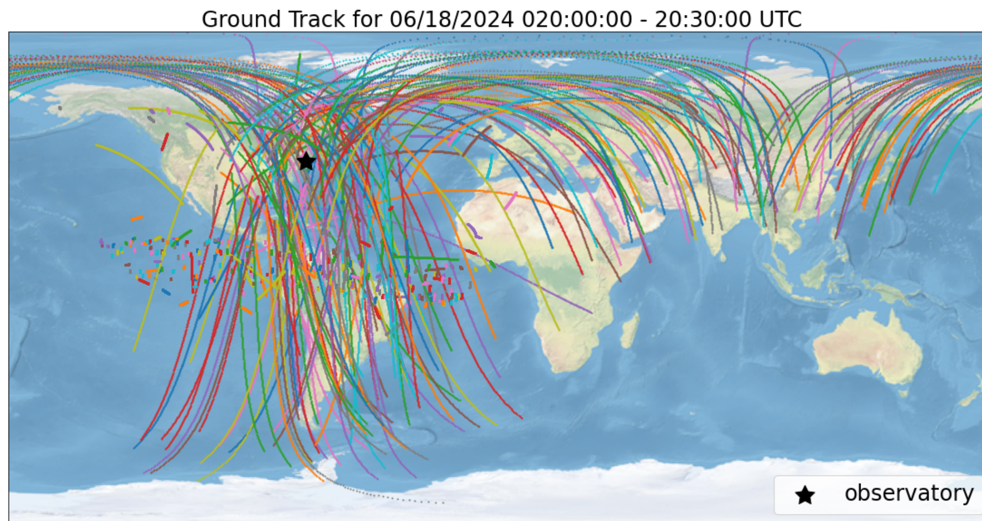

Figure 4.2: Selection of 500 satellites that pass over the Haystack Observatory (lat=42.58, lon=-71.44) during a 30 minute period (06-18-2024 20:00-20:30 UTC).

Chapter 5

Conclusions

The objective of this thesis was to create a software package that would aggregate relevant satellite information to be used in input datasets to tasking schedulers. The population of objects in space is growing rapidly, and we need to keep track of their movements to prevent collisions with other objects and manage space traffic. The United States Space Force uses a network of ground-based sensors to track the RSOs in Earth's orbit. The sensors require scheduling in order to be able to track every object multiple times per day, as is the Space Force's goal. Researchers such as those in [8] and [9] have developed scheduling algorithms and models to optimize the satellite tracking process, but they use different baseline input data to evaluate their methods, some of which is synthetically generated.

We introduced `satdatagen`: a Python library to support dataset generation for satellite sensor task schedulers. `satdatagen` enables researchers to create datasets in less than a minute's time and use them to directly compare the scheduling optimization algorithms. Users of the `satdatagen` library receive information regarding satellite brightness, satellite positions and movement, and the weather conditions for the time and place that they are running their scheduler, all automatically aggregated by the Python package. They can also limit the number of satellites returned in the dataset, and filter for satellites in a certain orbit. The `satdatagen` package is simple to use. Users do not require prior knowledge of dependencies, and a dataset can be generated with fewer than 10 lines of code.

5.1 Future Work

Based on the progress and modeling decisions considered in this thesis, we identify several areas for future work:

- **Add space-based sensor capability.** Currently, `satdatagen` can only generate datasets for ground-based sensor tasks. [6] explores using existing star-trackers mounted on satellites to track RSOs, which often have a wider field-of-view and more mobility than ground-based sensors. Sensor task schedulers are typically designed for only ground-based or only space-based sensors. With additional functionality in `satdatagen` to create datasets for space-based sensors, researchers could explore more how these sensors help to ease the burden of the ground-based network in RSO tracking.

- **Add a `Sensor` class**. Users of `satdatagen` can tailor the datasets to the requirements of the sensors they are scheduling by limiting the number of satellites represented in the dataset or filtering for a certain orbit. With a `Sensor` class, users would specify this customization in the instantiation of an object. The `Sensor` object would then contain the class method `generate_dataset`, rather than a `GroundLocation` object. This also creates space for more options for tailoring a dataset to a sensor's criteria, and de-clutters the optional input parameters of the current `generate_dataset` function.

Appendix A

Code listing

A.1 Example Dataset Code

```
import satdatagen as sdg
from datetime import datetime

start_date = datetime(2024, 6, 18, hour = 18, minute = 0)
periods = 24
haystack_lon = -71.44 #degrees west
haystack_lat = 42.58 #degrees north

credentials = '/path/to/space-track/credentials.json'

tr = sdg.TimeRange(start_date = start_date, periods = periods, delta
    = 30)
gl = sdg.GroundLocation(credentials, haystack_lat, haystack_lon, tr)

dataset = gl.generate_dataset(limit = 500)
```

A.2 generate_dataset

```
def generate_dataset(space_track_credentials, ground_loc, time_list,
    method = 'krag', limit = None, orbit = 'all', mixing_coeff = 0.8,
    output_file = None):
  '''
  generates the dataset as customized by user

  @param space_track_credentials: path to a .json file with your
     space-track.org login information
  @param ground_loc: astropy EarthLocation object
  @param time_list: list of astropy Time objects
```

```
8    @param [method]: string that determines which AVM method to use to
        determine satellite brightness.  options are 'krag', 'molczan',
        'hejduk'. default is the Krag method
9    @param [limit]: integer that limits the number of satellites
        represented in the dataset. default is no limit, all satellites
        that pass over head included
10   @param [orbit]: string that filters for objects in a certain orbit.
          options are 'LEO', 'MEO', 'GEO', 'all'. default is objects at
        all orbits
11   @param [mixing_coeff]: float between 0 and 1 that determines the
        ratio of diffuse/spectral reflection accounted for ONLY when
        method=='hejduk'
12   @param [output_file]: path to a .json file to output dataset to
13
14   @returns a python dictionary of all the satellites overhead with
        keys of satellite NORAD ID as given by space-track and values
        the altitude/azimuth of the satellite at the observation time
15
16   '''
17
18   cloud_cover = get_cloud_cover(ground_loc, time_list[0].datetime,
        time_list[-1].datetime)
19
20   overhead_sats = {}
21   observ_loc = ground_loc
22
23   all_sats_for_obstime = get_all_objects(space_track_credentials,
        time_list[0].datetime)
24
25   sats_in_dataset = []
26   sat_areas = []
27   if orbit == 'LEO':
28     for s in all_sats_for_obstime:
29       if float(s['SEMIMAJOR_AXIS']) < 7178:
30         sats_in_dataset.append(s)
31         sat_areas.append(get_object_area(s['NORAD_CAT_ID']))
32   elif orbit == 'MEO':
33     for s in all_sats_for_obstime:
34       if float(s['SEMIMAJOR_AXIS']) >= 7178 and float(s['
          SEMIMAJOR_AXIS']) < 36378:
35         sats_in_dataset.append(s)
36         sat_areas.append(get_object_area(s['NORAD_CAT_ID']))
37   elif orbit == 'GEO':
38     for s in all_sats_for_obstime:
39       if float(s['SEMIMAJOR_AXIS']) >= 36378:
40         sats_in_dataset.append(s)
41         sat_areas.append(get_object_area(s['NORAD_CAT_ID']))
```

```
42   else:
43     sats_in_dataset = all_sats_for_obstime
44     for s in all_sats_for_obstime:
45       sat_areas.append(get_object_area(s['NORAD_CAT_ID']))
46
47
48   e, r, v = propagate_sats(sats_in_dataset, time_list)
49
50   trans_start = time.time()
51   v_x = v[:,:,0] * u.km / u.s
52   v_y = v[:,:,1] * u.km / u.s
53   v_z = v[:,:,2] * u.km / u.s
54
55   r_x = r[:,:,0] * u.km
56   r_y = r[:,:,1] * u.km
57   r_z = r[:,:,2] * u.km
58
59   r_cartesian = CartesianRepresentation(x=r_x, y = r_y, z = r_z)
60   v_cartesian = CartesianDifferential(d_x = v_x, d_y = v_y, d_z = v_z
       )
61   teme = TEME(r_cartesian.with_differentials(v_cartesian), obstime=
       time_list)
62
63   obstime = Time(time_list, scale = 'utc')
64   itrs_sat = teme.transform_to(ITRS(obstime=time_list))
65
66   topo_itrs_sat = itrs_sat.cartesian.without_differentials() -
       observ_loc.get_itrs(obstime).cartesian
67   sat_itrs_topo = ITRS(topo_itrs_sat, obstime=obstime, location=
       observ_loc)
68   altaz = sat_itrs_topo.transform_to(AltAz(obstime=obstime, location=
       observ_loc))
69   alt_degs = altaz.alt.dms[0]
70
71   over_indices = np.nonzero(alt_degs>10)
72
73   unique_sats = np.unique(over_indices[0])
74   random_sats = unique_sats
75
76   if limit and limit <= len(unique_sats):
77     random_sats = np.random.choice(unique_sats, limit, replace =
         False)
78   prev_s = -1
79   for i in range(len(over_indices[0])):
80     s = over_indices[0][i]
81     t = over_indices[1][i]
82     if s not in random_sats:
```

```
83        continue
84      else:
85
86        if s != prev_s:
87          sat = sats_in_dataset[s]
88        s_area = sat_areas[s]
89        prev_s = s
90        avm = None
91        cc_idx = (24 * (time_list[t].datetime.date() - time_list[0].
             datetime.date()).days) + time_list[t].datetime.hour
92
93        if s_area is not None:
94          avm = str(get_avm(int(sat['NORAD_CAT_ID']), observ_loc,
               itrs_sat[s,t], s_area, time_list[t], method = method,
               mixing_coeff = mixing_coeff))
95        oh_dict = {'name':sat['OBJECT_NAME'],'time': time_list[t].
             datetime.isoformat(), 'alt':str(altaz[s,t].alt.dms), 'az':
             str(altaz[s,t].az.dms), 'TLE_LINE1':sat['TLE_LINE1'], '
             TLE_LINE2':sat['TLE_LINE2'], 'AVM' : avm, 'cloud_cover' :
             str(cloud_cover[cc_idx])}
96        if sat['NORAD_CAT_ID'] in overhead_sats:
97          overhead_sats[sat['NORAD_CAT_ID']].append(oh_dict)
98        else:
99          overhead_sats[sat['NORAD_CAT_ID']] = [oh_dict]
100
101
102   if output_file:
103     out = open(output_file, 'w')
104     json.dump(overhead_sats, out)
105     out.close()
106   return overhead_sats
```

Appendix B

List of Dependencies

- `astropy` version 5.3.4
- `datetime` version 5.5
- `numpy` version 1.26.0
- `openmeteo-requests` version 1.2.0
- `requests` version 2.32.3
- `sgp4` version 2.23

Bibliography

[1] Jon Ludwigson. Space Situational Awareness: DOD Should Evaluate How It Can Use Commercial Data - Report to congressional committees. https://www.gao.gov/products/gao-23-105565, 2023.

[2] Karen L. Howard. Technology Assessment: Large Constellations of Satellites; Mitigating Environmental and Other Effects. https://www.gao.gov/assets/730/723162.pdf, 2022.

[3] Jackson Ryan. "Space is Becoming an 'Unsustainable Environment in the Long Term,' ESA Says". https://www.space.com/european-space-agency-space-environment-report, July 2024.

[4] Elizabeth Howell. "ISS Astronauts Take Shelter in Boeing Starliner and Other Return Spacecraft After June 26 Satellite Breakup". https://www.space.com/iss-astronauts-shelter-return-spacecraft-satellite-breakup, June 2024.

[5] European Space Agency. ESA's Annual Space Environment Report. https://www.sdo.esoc.esa.int/environment_report/Space_Environment_Report_latest.pdf, July 2024.

[6] Samuel Clemens, Regina Lee, Paul Harrison, and Warren Soh. Feasibility of Using Commercial Star Trackers for On-Orbit Resident Space Object Detection. In *2018 Advanced Maui Optical and Space Surveillance Technologies Conference (AMOS)*, 2018.

[7] 18th Space Defense Squadron. space-track.org. https://www.space-track.org/, April 2024.

[8] Lindsey Blanks. Operational Scheduling of Deep Space Radars for Resident Space Object Surveillance. Master's thesis, Massachusetts Institute of Technology, 2022.

[9] Peng Mun Siew, Daniel Jang, Thomas G. Roberts, and Richard Linares. Space-Based Sensor Tasking Using Deep Reinforcement Learning. *The Journal of the Astronautical Sciences*, 69(6):1855–1892, November 2022.

[10] The Astropy Collaboration, Robitaille, Thomas P., Tollerud, Erik J., Greenfield, Perry, Droettboom, Michael, Bray, Erik, Aldcroft, Tom, Davis, Matt, Ginsburg, Adam, Price-Whelan, Adrian M., Kerzendorf, Wolfgang E., Conley, Alexander, Crighton, Neil, Barbary, Kyle, Muna, Demitri, Ferguson, Henry, Grollier, Frédéric, Parikh, Madhura M., Nair, Prasanth H., Günther, Hans M., Deil, Christoph, Woillez, Julien, Conseil, Simon, Kramer, Roban, Turner, James E. H., Singer, Leo, Fox, Ryan, Weaver, Benjamin A., Zabalza, Victor, Edwards, Zachary I., Azalee Bostroem, K., Burke, D. J., Casey, Andrew

R., Crawford, Steven M., Dencheva, Nadia, Ely, Justin, Jenness, Tim, Labrie, Kathleen, Lim, Pey Lian, Pierfederici, Francesco, Pontzen, Andrew, Ptak, Andy, Refsdal, Brian, Servillat, Mathieu, and Streicher, Ole. Astropy: A community python package for astronomy. *Astronomy & Astrophysics*, 558:A33, 2013.

[11] Juan Rodríguez and Jorge Garrido. poliastro: a Python library for interactive astrodynamics. In *Proceedings of the 21st Python in Science Conference (SCIPY 2022)*, pages 136–146, Austin, Texas, 2022.

[12] Marek Ziebart Charles Constant, Santosh Bhattarai. Limitations of current practices in uncooperative space surveillance: Analysis of mega-constellation data time-series. In *2023 Advanced Maui Optical and Space Surveillance Technologies Conference (AMOS)*. Advanced Maui Optical and Space Surveillance Technologies (AMOS), 2023.

[13] David Vallado and Paul Crawford. SGP4 Orbit Determination. In *AIAA/AAS Astrodynamics Specialist Conference and Exhibit*, Honolulu, Hawaii, August 2008. American Institute of Aeronautics and Astronautics.

[14] David A Vallado, Benjamin Bastida Virgili, and Tim Flohrer. Improved SSA Through Orbit Determination of Two-Line Element Sets. In *Proc. '6th European Conference on Space Debris'*, Darmstadt, Germany, 2013.

[15] Felix R. Hoots and Ronald L. Roehrich. Models for Propagation of NORAD Element Sets:. Technical report, Defense Technical Information Center, Fort Belvoir, VA, December 1980.

[16] D.A. Vallado and W.D. McClain. *Fundamentals of Astrodynamics and Applications*. Fundamentals of Astrodynamics and Applications. Microcosm Press, 2001.

[17] John Seago and David Vallado. Coordinate frames of the U.S. Space Object Catalogs. In *Astrodynamics Specialist Conference*, Denver,CO,U.S.A., August 2000. American Institute of Aeronautics and Astronautics.

[18] International Earth Rotation and Reference Systems Service. EOP of the day. https://www.iers.org/IERS/EN/DataProducts/tools/eop_of_today/eop_of_today_tool.html, 2024.

[19] William E. Krag. Visible Magnitude of Typical Satellites in Synchronous Orbits:. Technical report, Defense Technical Information Center, Fort Belvoir, VA, September 1974.

[20] M D Hejduk. Specular and Diffuse Components in Spherical Satellite Photometric Modeling. In *2011 Advanced Maui Optical and Space Surveillance Technologies Conference (AMOS)*, 2011.

[21] Matthew M Schmunk. Initial Determination of Low Earth Orbits Using Commercial Telescopes. Master's thesis, Air Force Institute of Tehchnology, 2008.

[22] Takao Endo, HItomi Ono, Jiro Suzuki, Toshiyuki Ando, and Takashi Takanezawa. Satellite Type Estination from Ground-based Photometric Observation. *2016 Advanced Maui Optical and Space Surveillance Technologies Conference (AMOS)*, 2016.

[23] N Rajan, T Morgan, R Lambour, and I Kupiec. Orbital Debris Size Estimation from Radar Cross Section Measurements. In *Proceedings of the 3rd European Conference on Space Debris, ESOC*, Darmstadt, Germany, 2001.

[24] Carl Reed and David Graham. Volume 5: OGC CDB Radar Cross Section (RCS) Models (Best Practice). https://docs.ogc.org/bp/16-004r5.html#_rcs_data_model, 2021.

[25] J. Tregloan-Reed, A. Otarola, E. Ortiz, V. Molina, J. Anais, R. González, J. P. Colque, and E. Unda-Sanzana. First observations and magnitude measurement of Starlink's Darksat. *Astronomy & Astrophysics*, 637:L1, May 2020.

[26] Gary A. McCue, James G. Williams, and Joan M. Morford. Optical characteristics of artificial satellites. *Planetary and Space Science*, 19(8):851–868, August 1971.

[27] Chance Johnson. Comparing Photometric Behavior of LEO Constellations to SpaceX Starlink using a space- based optical sensor. *2021 Advanced Maui Optical and Space Surveillance Technologies Conference (AMOS)*, 2021.

[28] Mike McCants. Mike McCants' Satellite Tracking Web Pages. https://www.mmccants.org/tles/intrmagdef.html, 2020.

[29] Patrick Zippenfenig. Open-Meteo.com Weather API. https://open-meteo.com/, 2023.

[30] Bell B. Berrisford P. Biavati G. Horányi A. Muñoz Sabater J. Nicolas J. Peubey C. Radu R. Rozum I. Schepers D. Simmons A. Soci C. Dee D. Hersbach, H. and J-N. Thépaut. ERA5 Hourly Data on Single Levels from 1940 to Present. ECMWF, https://cds.climate.copernicus.eu/doi/10.24381/cds.adbb2d47, 2023.

[31] Python Packaging Authority. Python Package Index. https://pypi.org/, 2024.