A SCALABLE HARDWARE-IN-THE-LOOP SIMULATION FOR SATELLITE CONSTELLATIONS AND OTHER MULTI-AGENT NETWORKS

A Thesis
Presented to
The Academic Faculty

By

Christopher F. DeGraw

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Aerospace Engineering

Georgia Institute of Technology

May 2018

A SCALABLE HARDWARE-IN-THE-LOOP SIMULATION FOR SATELLITE CONSTELLATIONS AND OTHER MULTI-AGENT NETWORKS

To Mary Louise,

The best partner I could have asked for this and everything else in our lives.

To my daughter (yet to be named)

I can't wait to meet you in just a few weeks!

ACKNOWLEDGEMENTS

I would like to thank Dr. Marcus Holzinger for inviting me to join his group and giving me the opportunity to develop this project at the intersection of his and my interests. I would also like to thank Sandia National Laboratories for funding my time here at Georgia Tech through the Critical Skills Master's Program. Most importantly, I would like to thank my wife, Mary Louise Gucik-DeGraw, who was supportive of me undertaking this degree even though it meant living apart for nearly two years and the sheer madness of the last three weeks.

Also, a special thank you to Dr. Andris Jaunzemis for providing me with an emergency Python 3 orbit propagator at the 11th hour.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 Motivation

The recent miniaturization and standardization of satellite components has made smallsat development an economically viable route for commercial space development on a massive scale[1, 2]. Entrepreneurs have capitalized on this cost reduction to find more uses for such constellations and are now able to find financial backing to implement them. Additionally, public sector entities continue to operate and design large constellations for a variety of purposes both civilian and military. Current examples include large constellations of large satellites such as the GPS system, large constellations of small satellites like the Planet's Earth observation constellation, and small constellations of large satellites including NASA's A-Train[3, 4, 5]. Other paradigms include the Sirius XM constellation, which uses a mixture of cheap smallsats in LEO coordinating with large, expensive assets in GEO[6]. Companies, including SpaceX and OneWeb, are planning "mega-constellations" containing thousands of satellites, pushing the field an order of magnitude beyond current operating paradigms[7, 8].

Low Earth orbit (LEO) is already a crowded environment, and the number of proposed constellations increases the chances that orbital debris and malfunctioning or dead satellites will pose a risk to operational systems[9, 10]. Additionally, satellite ground control is already an intense endeavor and the effort required to manage conjunctions with other satellites and debris will only increase[11]. The US Air Force's planned Space Fence will increase the size of the LEO resident space object (RSO) catalog and therefore increase the number of conjunctions which must be managed[12]. The confluence of these plans has created four general motivations for the project proposed in this thesis.

1.1.1 Motivation 1: Development of Mega-Constellations and Swarms

Some of the planned mega-constellations are being designed using a “swarm” paradigm. There appears to be no globally agreed upon definition separating a “swarm” from a constellation, but a useful definition used by Verhoeven et. al. specifies that the individual elements of a swarm be functionally identical[13]. The swarm paradigm provides reliability and performance through the quantity of swarm members, and it explicitly does not require the survival or availability of any particular member of the swarm. Swarms also rely on emergent behaviors generated by interactions between swarm members[13, 14, 15]. This reliance on emergent behaviors and distributed control systems can make analytical evaluation of swarm behavior difficult, thus requiring a robust simulation environment to evaluate swarm behavior prior to operational testing. For simplicity, this paper will include swarms as part of the more general definition of a “constellation”.

It is important to note that swarm concepts expect the failure of individual swarm members. In a swarm of locusts, predators can consume thousands of individual grasshoppers, but the risk to any particular grasshopper remains nearly zero, and so the swarm survives. In the case of satellites, a failed member becomes dangerous hypervelocity space debris and poses a much greater risk to the LEO environment than a dead swarm member would in a terrestrial project [10].

1.1.2 Motivation 2: Increasing Number of Resident Space Objects in Low Earth Orbit

NASA and DARPA have presented models showing that, assuming no further satellite launches, the amount of space debris will nearly double over the next 50 years[16]. However, SpaceWorks Enterprises reports that there are 803 nano- and micro-satellite launches planned for the 2017-2019 period. They also project that there will be 320 to 460 similar launches yearly by 2023[17]. These satellites include a large number of university missions, which are a great opportunity for students but present a problem given the lack of experience and resources for such missions.

1.1.3 Motivation 3: Ground Station Proliferation and Workload

Most current paradigms require ground stations and operators to monitor each satellite's health, instrumentation, and handle conjunctions manually. Planet has already discovered a need to increase the automation of its ground operations with only a few hundred satellites on orbit[18]. Mega-constellations will require an exponential increase in the capability and reliability of automated control systems. They will also require the development of guidelines for how humans interact with constellations and how automated mega-constellations interact with other space environment residents, especially other automated mega-constellations.

1.2 Near-Term Challenges

We have derived a set of six expected challenges, or "blind spots" in the Johari Window framework, from these three motivations[19]. These challenges often overlap the three motivations and have led us to the work being proposed for this thesis.

1.2.1 Need to Increase Technology Readiness Levels Quickly and Affordably

Legacy developers such as Lockheed, Boeing, NASA, and the ESA have large reservoirs of operational experience and tribal knowledge to draw from. Newer entrants to the field, such as start-up companies and universities, need to develop that knowledge base through trial and error or by poaching experienced operators. In the case of university-class missions, this is further complicated by a 2-5 year turnover period for all their "employees". Many of these programs develop their testing and simulation capacity in-house, as there does not appear to be a general-use framework for evaluating new satellite concepts. The development of such a framework could lower the barriers to entry for university-class and entrepreneurial satellite development while increasing the safety and reliability of such projects.

1.2.2 Need to Explore Constellation Operation Procedures

The only two private entities with large constellation operation experience are Planet and Iridium with 196 and 66 satellites respectively. Below that are the government and military-run navigation constellations with only 30-40 satellites each. This means that the procedures for operating mega-constellations of 500-2000, or more, satellites likely do not exist and certainly have not been tested on operational systems. The companies planning mega-constellations are most likely developing procedures and ways to test them, but there is no transparent and publicly evaluable framework for determining if these procedures are safe, effective, or optimal.

1.2.3 Need to Mitigate High Operational Tempo and Operator Overload

Planet operates the largest satellite constellation on orbit and its satellites also most closely match a swarm paradigm. However, until very recently, each satellite was controlled individually, although some guidance decisions were made based on the constellation state[20]. Furthermore, the number of potential conjunctions involving Planet satellites is growing rapidly[11]. Manging these conjunction events required substantial effort on the part of Planet's ground controllers. Because of that, they are developing automated operations systems to handle this load, but their satellite constellation is still an order of magnitude smaller than the proposed next generation systems[18]. The activation of the USAF's Space Fence will also increase the number conjunction events that must be managed by a constellation operator even before the injection of mega-constellations into LEO[12].

The GPS constellation is four times smaller than the Planet constellation, but its much greater complexity has several instructive lessons for constellation management. Currently, the GPS Control Segment manages each satellite individually, requiring at least once-per-day contacts with each vehicle. For only 31 satellites, GPS communication and control requires 24-7 staffing with up-links occurring at least once every 45 minutes[21]. Fortunately for the operators, the GPS constellation exists in an isolated orbit with few, if any, conjunction events. This already punishing operation tempo could become unmanageable if combined with the need for conjunction awareness and avoidance in LEO.

1.2.4 Need to Develop Autonomous Satellite and Constellation Systems

Planet is already developing automation procedures to reduce the workload associated with conjunctions for their relatively modest 196 satellite constellation[11, 18]. Their differential drag control scheme was implemented through ground systems with commands transmitted to the satellites during ground communication windows[20]. While extremely effective in deploying the Flock-2b, it still involved ground-in-the-loop decision making and human-in-the-loop evaluation of the developed solution. A 1000-member mega-constellation would expect a dramatically increased catalog of guidance, navigation, and control (GNC) decisions.

Planet's automated control systems are showing promise in managing their constellation. However, we can see from the development of self-driving automobiles that larger scale automation solutions in more complex environments are not easily implemented or tested. An independent framework capable of providing verified and validation simulation assessment could reduce the risk associated with deploying autonomous satellite constellations.

1.2.5 Need to Evaluate Constellation Interactions in the Crowded LEO Environment

The previous challenge considers the needs of a single automated mega-constellation. However, current plans include the simultaneous deployment of multiple mega-constellations, each most likely requiring some level of automation. The interactions of multiple mega-constellations will likely create emergent behaviors between constellations as they may not be designed to cooperate with each other's needs in mind. An independent simulation framework becomes more necessary as we consider multiple autonomous mega-constellations being deployed by strategic competitors into the LEO environment.

1.2.6 Need to Develop Space Traffic Control

A 2005 article co-authored by Secretary of Defense William J. Perry, Air Force General Brent Scowcroft, and others discussed the need to develop a system for "Space Traffic Control" (STC)[22]. Since then, many other voices have added to the literature surrounding STC, but very little policy has been implemented. Satellite operators and mega-constellation entrepreneurs are pushing for voluntary, bottom-up frameworks as opposed to governmental or international body regulations[23]. Whether STC is implemented by a governing body or as a collaboration between industry players, it will need highly developed control and visualization tools to understand the increasingly crowded and complicated space environment.

1.3 Proposed Contributions

Addressing all of these problems is an enormous task. However, it should be possible to create a framework from which to approach many of these challenges. We believe that the development of a robust, modular, scalable, and high-fidelity hardware-in-the-loop (HWIL) simulator for massive-scale constellations will address many of the listed challenges. The particular work for this thesis is divided into two primary contributions.

1.3.1 Contribution 1: Development of a Scalable Satellite Constellation Simulator

The first goal of this research is the creation of a system capable of providing high-fidelity and transparent HWIL simulation capability for satellite constellations and swarms. There appears to be no general purpose mutli-satellite simulator in the current literature, although one proposed system suggested using cloud computing to scale single satellite simulations up to a constellation simulation[24]. Unfortunately, the latency associated with cloud computing does not permit the integration of flight hardware into a real-time simulation.

This project is called Constellation Simulation on a Massive Scale, or COSMoS. The initial operational goal of COSMoS is to test different control algorithms on a cohesive 93-satellite constellation and determine the efficacy and efficiency of those methods.

As a fortunate side effect, the system on which COSMoS runs will not be application specific. The system is separately called the Multi-Agent Distributed Network Simulator, or MADNS. MADNS will be capable of simulating swarm systems other than satellite constellations and should provide value outside the spaceflight community. For the sake of brevity most references to COSMoS in this paper will implicitly include MADNS as it is only current application.

The specific development targets for this contribution are:

1. Demonstrate COSMoS's ability to operate at 100 Hz or faster with more than 24 agents.

2. Create visualizations of COSMoS data in post-processing. A stretch goal would be data display in real-time.

3. Test at least one algorithm for controlling a large satellite constellation.

1.3.2 Contribution 2: Hardware Integration with COSMoS

Hardware-in-the-loop (HWIL) simulation is a common industry practice for retiring risk in experimental and production technologies. Marko Bacic quantifies the quality of an HWIL simulation in terms of “transparency” and robustness of prediction. Transparency is a measure of how closely a simulation mimics the expected interactions between hardware and its operating environment. Robustness of prediction is the measure of how well simulation results match experimental and operational data[25].

The testing of flight and flight-like hardware in experimentally validated simulations is a key part of developing the technology readiness level (TRL) of untested technologies. A goal of this simulator is be to increase a component or system’s TRL to 4 or 5, which are defined as “Component and/or breadboard validation in a laboratory environment” and “Component and or breadboard validation in a relevant environment” respectively. If COSMoS can be validated against flight missions, it may be possible to increase TRL to level 6: “System/subsystem model or prototype demonstration in a relevant environment”[26, 27, 28].

The specific development targets for this contribution are:

1. Modularity: Demonstrate the ability of COSMoS to interface with independently developed code modules to develop a simulation.

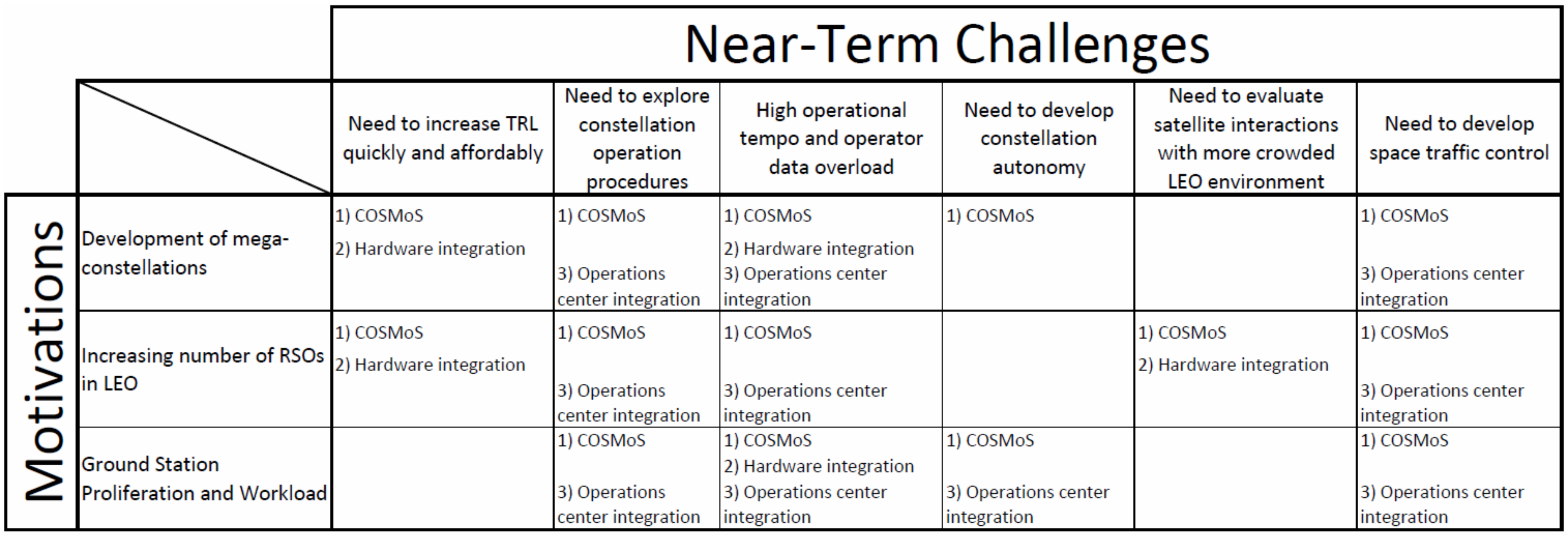
2. HWIL: Evaluate COSMoS’s transparency by comparing simulations where agents are provided with truth data to simulations using modeled actuator and sensor data.

3. HWIL: Demonstrate COSMoS’s ability to communicate with external hardware devices.

1.4 Summary of Planned Contributions and Relevant Literature

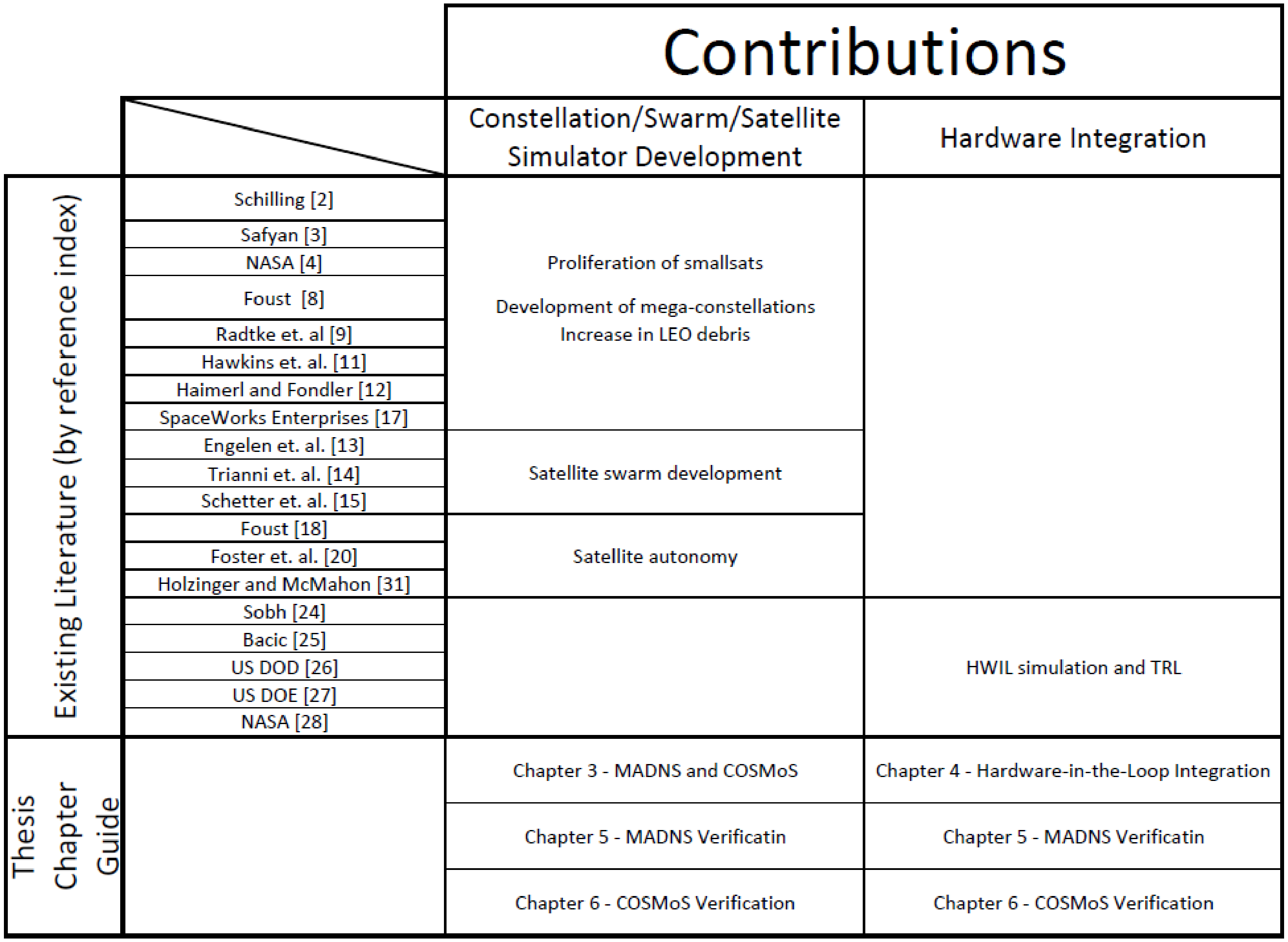
Figure 1.1 shows how the proposed contributions match the problems and motivations describe above. Figure 1.2 shows how the relevant existing literature backs the proposed contributions. Additionally, this project will provide the following specific benefits to the SSDL group:

- It will provide a software-in-the-loop (SWIL) and HWIL simulation capability for SSDL flight missions.
- It will create a platform capable of training SSDL mission controllers.



		Near-Term Challenges					
		Need to increase TRL quickly and affordably	Need to explore constellation operation procedures	High operational tempo and operator data overload	Need to develop constellation autonomy	Need to evaluate satellite interactions with more crowded LEO environment	Need to develop space traffic control
Motivations	Development of mega-constellations	1) COSMoS 2) Hardware integration	1) COSMoS 3) Operations center integration	1) COSMoS 2) Hardware integration 3) Operations center integration	1) COSMoS		1) COSMoS 3) Operations center integration
	Increasing number of RSOs in LEO	1) COSMoS 2) Hardware integration	1) COSMoS 3) Operations center integration	1) COSMoS 3) Operations center integration		1) COSMoS 2) Hardware integration	1) COSMoS 3) Operations center integration
	Ground Station Proliferation and Workload		1) COSMoS 3) Operations center integration	1) COSMoS 2) Hardware integration 3) Operations center integration	1) COSMoS 3) Operations center integration		1) COSMoS 3) Operations center integration

Figure 1.1: Motivations, near-term problems, and contributions.



		Contributions	
		Constellation/Swarm/Satellite Simulator Development	Hardware Integration
Existing Literature (by reference index)	Schilling [2]	Proliferation of smallsats Development of mega-constellations Increase in LEO debris	
	Safyan [3]		
	NASA [4]		
	Foust [8]		
	Radtke et. al [9]		
	Hawkins et. al. [11]		
	Haimerl and Fondler [12]		
	SpaceWorks Enterprises [17]		
	Engelen et. al. [13]	Satellite swarm development	
	Trianni et. al. [14]		
	Schetter et. al. [15]		
	Foust [18]	Satellite autonomy	
	Foster et. al. [20]		
	Holzinger and McMahon [31]		
	Sobh [24]		HWIL simulation and TRL
	Bacic [25]		
	US DOD [26]		
	US DOE [27]		
	NASA [28]		
Thesis Chapter Guide		Chapter 3 - MADNS and COSMoS	Chapter 4 - Hardware-in-the-Loop Integration
		Chapter 5 - MADNS Verificatin	Chapter 5 - MADNS Verificatin
		Chapter 6 - COSMoS Verification	Chapter 6 - COSMoS Verification

Figure 1.2: Contributions and related literature.

CHAPTER 2
DESIGN PHILOSOPHY AND SYSTEM PLANNING

A substantial amount of planning and design has gone into the development of COSMoS. It is necessary to carefully separate simulation truth and agent "reality" in the system. Additionally, the system has many of same problems found in supercomputing such as I/O and network bottlenecks, data access rates, node communication, thermal management, and power management. The solutions to these problems developed in this thesis are described below.

2.1 Conceptual Design

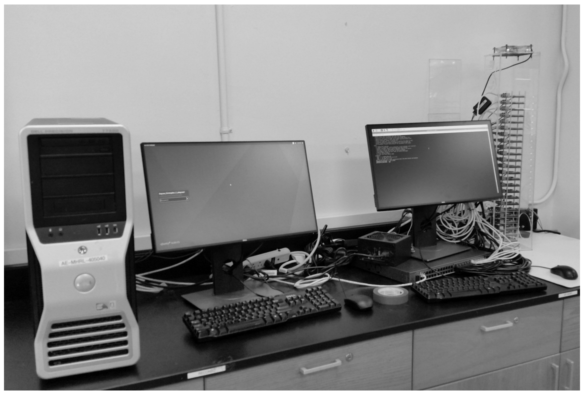


Figure 2.1: Current COSMoS System.

HWIL simulation quality is the guiding design principle behind COSMoS. Therefore, there is an emphasis on simulation fidelity, modularity, and the strict separation of noumenological and phenomenological processes. The terms noumenological and phenomenological are borrowed from Kantian philosophy where a noumenon is a "thing as it is in itself", or in this case simulation truth. Phenomenons are "things as they appear to be", which are instrument measurements such as magnetometer and star tracker outputs[29]. The system architecture will be designed to use plug-in modules developed by subject-matter experts to improve simulation fidelity, although there will be locally developed modules to fill gaps where these are unavailable.

The heart of COSMoS will be a message passing interface (MPI) designed to provide maximum transparency between externally developed simulation environment and the hardware and/or software being tested. A common application programming interface (API) will make the integration of external simulation code as simple and modular as possible. The fidelity of physical models developed by subject matter experts and transparency of the MPI should translate into a high robustness of prediction.

Modularity is accomplished through both hardware and software design. The current design includes the host computer, one managed gigabit network switch, and 24 single-board computers (SBCs). In this case, each SBC is be a Raspberry Pi 3B micro-computer. Each SBC will host a single simulated satellite and will referred to as an "agent". The COSMoS environment will include a wrapper designed to provide maximum transparency for the flight software (FSW) running on the agents, while the host computer runs the simulated environment and handles all data communication across the network.

The strict separation between noumenological and phenomenological processes is maintained by logically isolating agents from the simulation processes on the host. A design diagram showing a dual-loop FSW routine operating under this regime can be seen in Figure 2.2. The result is a closed-loop control system which presents flight hardware and software with an environment that is as realistic, from the perspective of the flight system, as possible.

2.2 System Hardware Configuration

The system hardware design also follows the principles of modularity, simulation fidelity and noumenon/phenomenon separation. The SBC micro-computers are divided

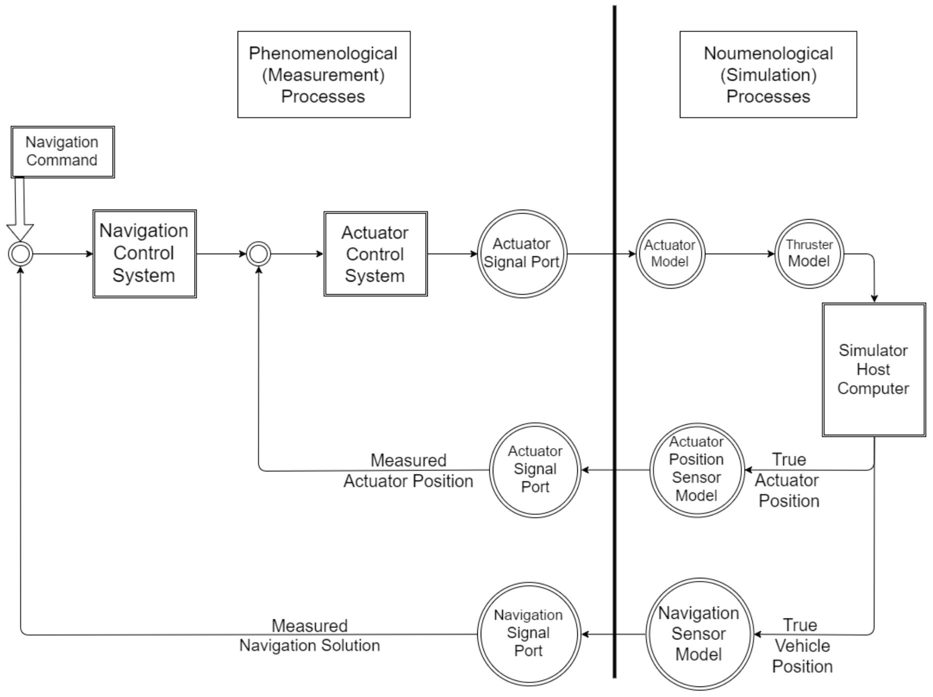


Figure 2.2: Control scheme with strict noumenonological and phenomenonological separation.

into towers, each holding 23 or 24 SBCs encased in an acrylic enclosure. Currently, one tower has been constructed and is operational with 24 SBCs. Each tower is a separate unit with access ports for power, Ethernet, and other mission-specific communication pathways, and can therefore be added or removed from the system at need. One 48-port managed network switch provides connectivity between the host computer and the SBCs. Figure 2.3 is a CAD model of an individual tower and the first tower prototype can be seen in Figure 2.4.

A hardware connection diagram for the entire system is shown in Figure 2.5

2.2.1 Current Hardware

The current system supporting the MADNS simulator and COSMoS simulation consists of the following elements:

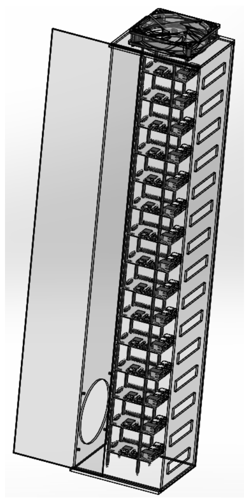


Figure 2.3: Tower CAD model.

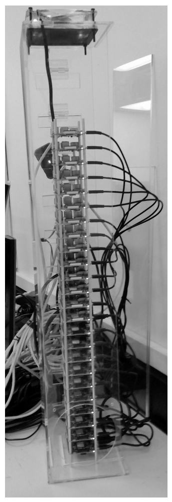


Figure 2.4: Tower prototype photograph.

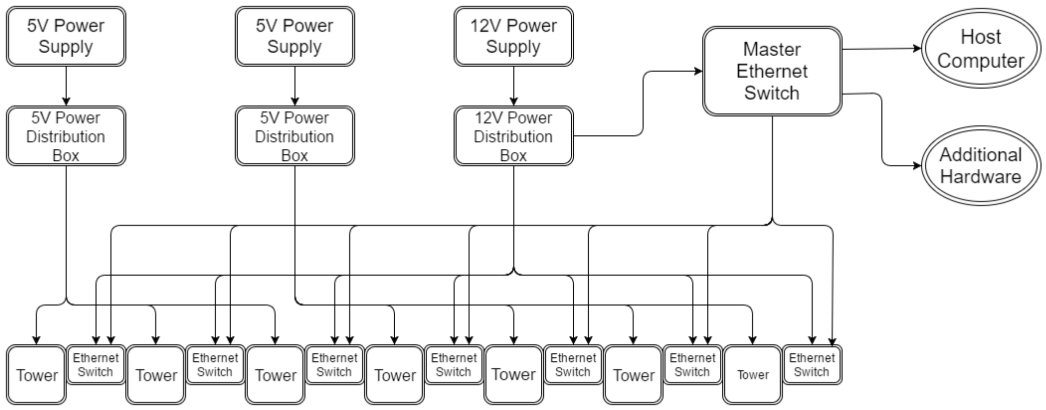


Figure 2.5: Hardware layout for the system.

Element	Description
Host Computer	8-Core Intel Xenon CPU E5520 @ 2.27 GHz
SBC	24 Raspberry Pi 3-B micro-computers
Network backbone	NETGEAR 48-Port Gigabit Managed Switch

2.3 Real-Time Operation

We are assuming that flight software and hardware operate at 1 Hz for general operations and 10 Hz for maneuvers. The goal is to have the simulator operate at 100 Hz, or better, so that the dynamics appear to be continuous to the flight computer. A rate of 200 Hz would be preferable to provide a minimum 20:1 ratio of simulated dynamics to control loop rate, but benchmarking tests need to be completed to determine if this is achievable.

Neither the host computer nor the SBCs run real-time operating systems (RTOS). This creates the potential for time management issues and the equivalent of real-time overflows during simulation operation. Currently, the SBCs have not suffered from processor overflow during intentionally stressful benchmark tests. Because the SBCs are more powerful than current generation flight computers, there should not be issues during COSMoS simulations. However, it is possible to apply a real-time operating system on the SBCs to enforce task prioritization and protect FSW routines from deprioritization.

2.4 Message Passing Interface

MADNS and COSMoS use a custom designed message passing interface (MPI) for their host to agent communications. There several high-quality and well tested message passing interfaces in common usage on clusters and supercomputers such MPICH[30] and Open MPI[31]. However, none of these options are specifically applicable to the MADNS real-time HWIL (RT-HWIL) framework.

The underlying architectures of MPICH and Open MPI assume that most processing units are interchangeable, as would be expected in a supercomputer with many identical nodes. These MPI systems allow for the separation the system into "ranks" using a rankfile to assign certain processes to specific nodes or processors, but generally seeks to limit this behavior to the lowest number of ranks needed for optimal execution. By contrast, the MADNS system assumes that each agent, or node in the MPI sense, is a unique system and therefore cannot be treated as interchangeable by the MPI. Implementing Open MPI or MPICH would require providing each agent with a unique rank, potentially resulting in hundreds or thousands of ranks in a massive-scale simulation. Most implementations of standard MPIs seek to minimize the number of ranks within the system to avoid overhead and optimize load distribution. Therefore, the MADNS system is fundamentally different in its approach to processing nodes than standard MPI libraries and MADNS requires an MPI designed for its structure.

2.5 Scalability of HWIL Simulators

In general, hardware-in-the-loop simulators run on fully real-time systems, i.e. the simulation runs in real time as do the hardware interfaces. In the case of a distributed network simulator, this requirement can be relaxed as only the distributed agents running active element software need to run in true real-time. The simulation itself can run asynchronously so long as there is no interruption in the availability of real-time state information at the agent level. To that end, the following framework can be used to determine the available trade-off space for the development of a simulator in the model of MADNS and COSMOS.

The scalability of such a system is governed by the ability to pass data round-trip between the host and agents. Much of this data transfer can be asynchronous, but sometimes data transfer will need to be near or truly synchronous and that will be the most restrictive limitation on the system. In both cases, the system can be divided into four elements placed on three hardware elements and each with a unique limiting factor: The two processes on the agent represent a quasi-noumenonological process

Table 2.1: Real-Time Data Management Elements

Process	Location	Limiting Factor
Data generation	Host	Parallel processing capacity
Data transfer	Network	Transmission bandwidth
Data buffering	Agent	Storage capability
Data consumption	Agent	Agent processing speed

run in proximity to the phenomenon, but it still maintains the distinction between data provided to an agent executable, such as flight software, and data available to the MADNS or COSMoS framework.

The host is responsible for generating all agent state data and must be able to provide this data to the agents as they need to consume it. As hardware analogs, agents operate at a set clock rate and only take updates on their clock ticks. Therefore, if an agent's main process is running at 10 Hz, it only needs to have the next state data packet available at its next time step, e.g. the agent ticks at $t = 0.0$ and will not check again until $t = 0.1$ seconds. If the host were attempting to provide single state updates to every agent on every tick, it is unlikely that the system could scale to an arbitrary number of real-time agents. However, if it is possible to buffer the data on the agents, this restriction becomes substantially more relaxed. This situation will be discussed in the contexts of synchronous and asynchronous communication.

2.5.1 Asynchronous Data Communication

The most general case for such a real-time HWIL simulator is a situation in which the agents and system dynamics respond relatively slowly to state changes and maneuvers can be planned for in advance. In this situation, it is possible for the host computer to propagate large stretches of state data based on expected system behavior and store those states in buffers. The agents can then draw on these buffers as needed to provide appropriate real-time state. An example of this process is shown in Figs. 2.5.1 and 2.5.1.

In this case there is actually no need for the host or the agent buffer to operate in real time. Instead, the governing limitation is that the host be able to replenish the agent buffer before it runs out of data. So long as the agent buffer remains populated, the simulation software (such as satellite FSW), will operate as if its receiving measurements and will act as though it were operating in a true hardware environment.

The restriction can be defined in terms of the following variables:

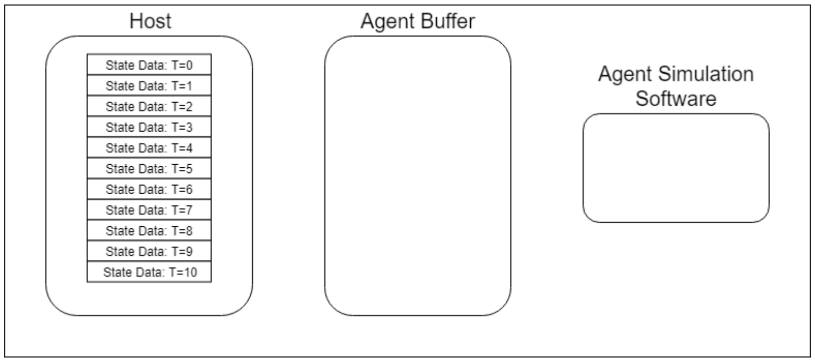


Figure 2.6: Asynchronous host to agent to software state data diagram before simulation start.

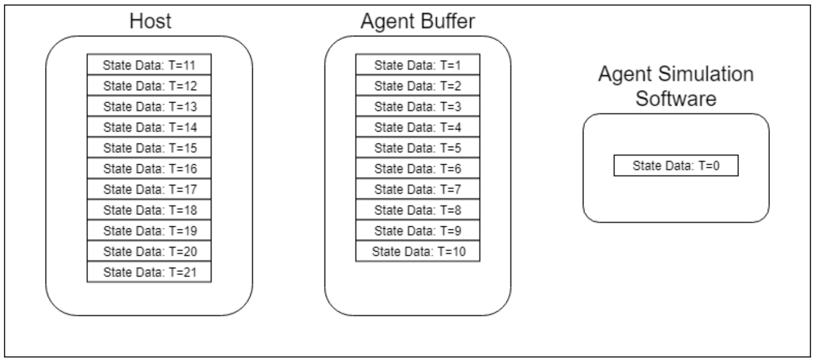


Figure 2.7: Asynchronous host to agent to software state data diagram after first transmission.

- N_a - Number of agents.
- N_p - Number of processing units available on the host.
- R_a - Operational rate of the agent software.
- S_a - Storage size of the agent buffer.
- S_b - Size of a data batch.
- S_{sync} - Size of a smaller synchronous data batch.
- T_{bi} - Time duration of the buffer on agent i which is a function of S_a and R_a.
- T_d - Time required to generate a batch of state updates, which is a function of S_b.

- T_n - Time required for the batch to pass through the network.

So long as the host is able to provide all agents with new batches of state data on time, the real-time nature of the simulation will be preserved. This gives the following governing inequality for the timing of asynchronous RT-HWIL communication:

$$T_p(S_b)\frac{N_a}{N_p} + T_n \leq T_{bi}(S_a, R_a) \tag{2.1}$$

Where the time required to create a data batch T_d is necessarily a function of the number of states in a batch S_b. Similarly, the amount of state data that can be stored in an agent buffer S_a and the rate at which the software consumes those states R_a determine the amount of time an agent can provide real-time data to its simulation without requiring an update from the host. In practice, this restriction has proven easy to accommodate for the relatively slow dynamics of a satellite constellation.

2.5.2 Synchronous Data Communication

The more restrictive scenario for an RT-HWIL simulator if the agents are maneuvering rapidly. Because the host is not "aware" of the agents' phenomenological decision making processes, it cannot necessarily anticipate their maneuvers and the effects those actions will have on the state propagation. In the event of a vehicle maneuver, some data must pass from the agent to the host, which must propagate a new state based on that data, and pass the data back to the agent. An example of this process can be seen in Fig. 2.5.2 which occurs after the example in Fig. 2.5.1.

This situation introduces the following new variables:

- T_s - Current simulation time.
- T_m - Time at which a maneuver will be executed.

In this case, the agent plans a maneuver at a future time T_m, but which can be the

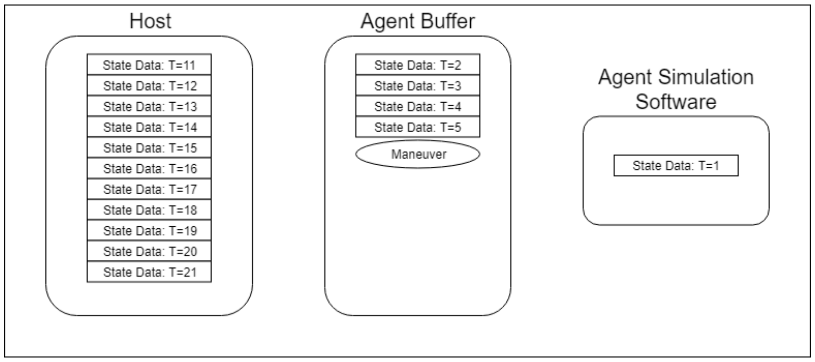


Figure 2.8: Agent plans a maneuver.

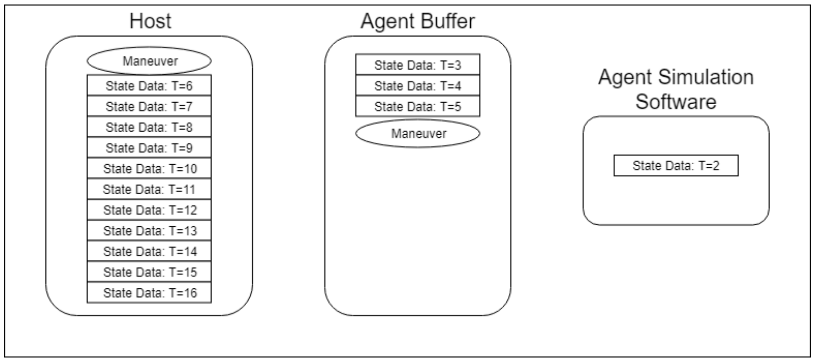


Figure 2.9: Host receives maneuver plan and re-propagates states.

next agent time step in the most restrictive case where:

$$T_m = T_s + \frac{1}{R_a} \tag{2.2}$$

Once the maneuver is planned, the agent transmits the maneuver plan to the host which must discard its pre-planned state buffer and immediately re-propagate the states based on the planned maneuver as shown in Fig. 2.5.2. Finally, the host most transmit this new set of states to the agent buffer as seen in Fig. 2.5.2

Synchronous communication is therefore governed by the following inequalities:

$$T_p(S_{\text{sync}}) + 2T_n \leq T_m - T_s \tag{2.3}$$

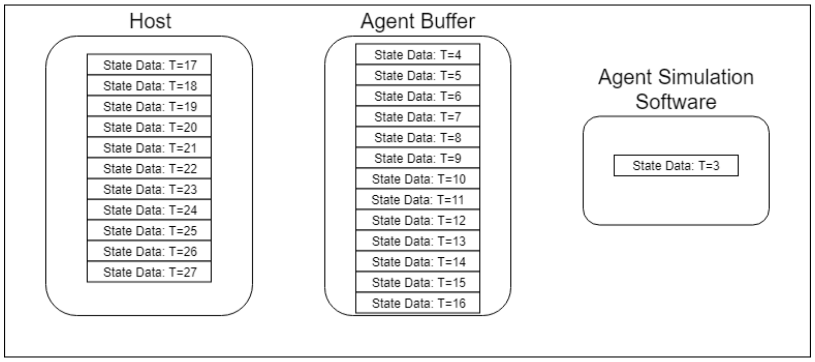


Figure 2.10: Agent receives new states and host resumes asynchronous propagation.

$$T_m - T_s \geq \frac{1}{R_a} \tag{2.4}$$

As mentioned before, the most restrictive case is the one where the simulated software expects the maneuver to begin on the next time step and the inequality in Eqn. 2.4 becomes an equality. In this case, the restriction becomes:

$$T_p(S_{\text{sync}}) + 2T_n \leq \frac{1}{R_a} \tag{2.5}$$

The example here visually shows a case where the synchronous data buffer S_{sync} is the same size as an asynchronous buffer S_b. However, this restriction can be eased by reducing the size of S_{sync} and therefore the also reducing the value of T_p for synchronous communication. This can ease the restriction and allow the system to continue operating in real-time as the host and agents resume asynchronous communication. Again, the most strenuous environment for this system is one in which all vehicles are maneuvering rapidly, but that is not often the situation for satellite constellations.

2.5.3 Network Communication Limitation

A fixed resource in this sort of system is the available network communication bandwidth. This sort of system can generate an immense amount of data and the price of network switches scales non-linearly with their speed. Therefore, for such a system, the amount of data passed along the network needs to be considered carefully. The following additional nomenclature is necessary for this discussion:

- B_n - The network bandwidth measured in bytes/s (or Gb/s)
- N_{sync} - The maximum anticipated number of synchronous communications.
- R_{Di} - The rate at which a particular data packet D_i is transmitted, including asynchronous state updates.
- S_{Di} - The size of the a particular data packet D_i, which includes asynchronous state updates.

And so the governing inequality for the network segment of the RT-HWIL simulator can be given as:

$$N_a \max_{\forall\ T+\Delta T} \left[\sum_i S_{Di} R_{Di} + N_{\text{sync}} S_{\text{sync}} \right] \leq B_n(T + \Delta T) \tag{2.6}$$

Which simply states the requirement that the maximum expected data transfer of planned, fixed rate communications ($S_{Di}R_{Di}$) and unexpected synchronous communications ($N_{\text{sync}}S_{\text{sync}}$) over any particular time interval $T + \Delta T$ be less than the network's transmission capability over that time interval ($B_n(T + \Delta T)$). As the planned communications will generally be fixed, this becomes a balancing act between the synchronous communication requirements and the network capability, which is limited by cost, and again where the value of S_{sync} can be tuned based on the system's ability to resume asynchronous communication.

2.5.4 Scalabilty Limits of Current Configuration

In summary, there are four inequalities which govern the scalability of a real-time HWIL simulator:

$$T_p(S_b)\frac{N_a}{N_p} + T_n \leq T_{bi}(S_a, R_a) \tag{2.7}$$

$$T_p(S_{\text{sync}}) + 2T_n \leq T_m - T_s \tag{2.8}$$

$$T_m - T_s \geq \frac{1}{R_a} \tag{2.9}$$

$$N_a \max_{\forall\ T+\Delta T} \left[\sum_i S_{Di} R_{Di} + N_{\text{sync}} S_{\text{sync}} \right] \leq B_n(T + \Delta T) \tag{2.10}$$

Eqn. 2.7 governs asynchronous communication, Eqns. 2.8 and 2.9 govern synchronous communication, and Eqn. 2.10 governs the network communications.

The current MADNS and COSMoS configuration has the following properties relative to these equations: Looking at the asynchronous communication inequality

Table 2.2: Current System Scalability Values

Equation Variable	Value
S_b	$100 \times 6 \times 64$ bits
S_{sync}	S_b
T_p	≤ 0.04 sec
N_a*	10 agents
N_p**	1 processor
T_n	≈ 0.001 sec
T_{bi}	10 sec
S_a	> 100 kB
R_a	10 Hz
T_m	$T_s + 3$ sec

*The system was unable to handle more than 10 agents due to issues discussed in Chapter 8

**The system was unable to parallelize due to issues discussed in Chapter 8

gives:

$$0.401 \text{ sec} \leq 10 \text{ sec} \tag{2.11}$$

The synchronous communication inequalities give:

$$0.042 \text{ sec} \leq 0.1 \text{ sec} \tag{2.12}$$

$$3 \text{ sec} \geq 0.1 \text{ sec} \tag{2.13}$$

All of which fall safely within the RT-HWIL requirements listed in this document.

Looking at bandwidth inequality requires examining all expected data packets on the network in a COSMoS simulation: Assuming a worst case scenario where over 1 second, all 10 agents require state updates, send system status updates, and require maneuver packets gives the following result:

Table 2.3: COSMoS Network Communications

Communication Data	Approximate size	Rate
State updates	38400 bytes	0.2 Hz
System Status packets	4000 bytes	$\approx$ 0.07 Hz
Agent time updates	200 bytes	10 Hz
Hardware interface packets	740 bytes	10 Hz

$$8.1 \times 10^5 \text{ bits} \leq 1 \times 10^9 \text{ bits} \tag{2.14}$$

These results suggest that, given the current system constraints and without modification, the synchronous communication inequality would potentially fail with 71 agents, the asynchronous communication would fail with 240 agents, and the network bandwidth requirement would fail with 12200 agents. As noted previously, it is possible to modify the synchronous communication inequality by reducing the size of a synchronous communication packet below that of a standard asynchronous packet, or by relaxing the maneuver time. For satellites, a 3-second lead time on a maneuver is extremely short, and so the synchronous communication is likely more forgiving than suggested here for even autonomous constellations. Therefore, it seems possible that the current architecture could support up to 1000 agents with a maneuver time margin of at least 45 seconds and 5 parallel cores processing asynchronous communication.

CHAPTER 3
DEVELOPMENT OF MADNS

3.1 A Scalable Simulation Framework

Early in the development for this project, it became clear that it could be used for more than just satellite constellation simulations. Satellite constellations represent only one kind of multi-agent network under autonomous or distributed control. Therefore, the decision was made to separate the distributed network functions from the specific simulation as seen in Fig. 3.1. The system is generally divided into startup, simulation, and closeout segments. The startup segment is responsible for ensuring that the host and all SBCs are operational and ready to begin simulation operations. Additionally, any initial data is generated and distributed to the agents during this phase before the real-time restrictions of the simulation phase take effect, thus reducing the initial load on the real-time restrictions. The simulation phase is the only segment required to run in real-time and is generally independent of the MADNS functions beyond requiring an adherence to the general MADNS API. The closeout functions confirm that all agents have completed their work, and then collects and logs all system data not logged during the setup and simulation phases. This division of labor allows for substantial flexibility in the development of multi-agent distributed network simulations beyond satellite constellations.

3.1.1 MADNS Data Flow

The MADNS data flow is highly versatile and can be modified based on a simulation's requirements. A data flow schematic for the MADNS benchmark tests is shown in Fig. 3.1. This test is described in greater detail in Section 6.1. As seen in Fig. 3.1,

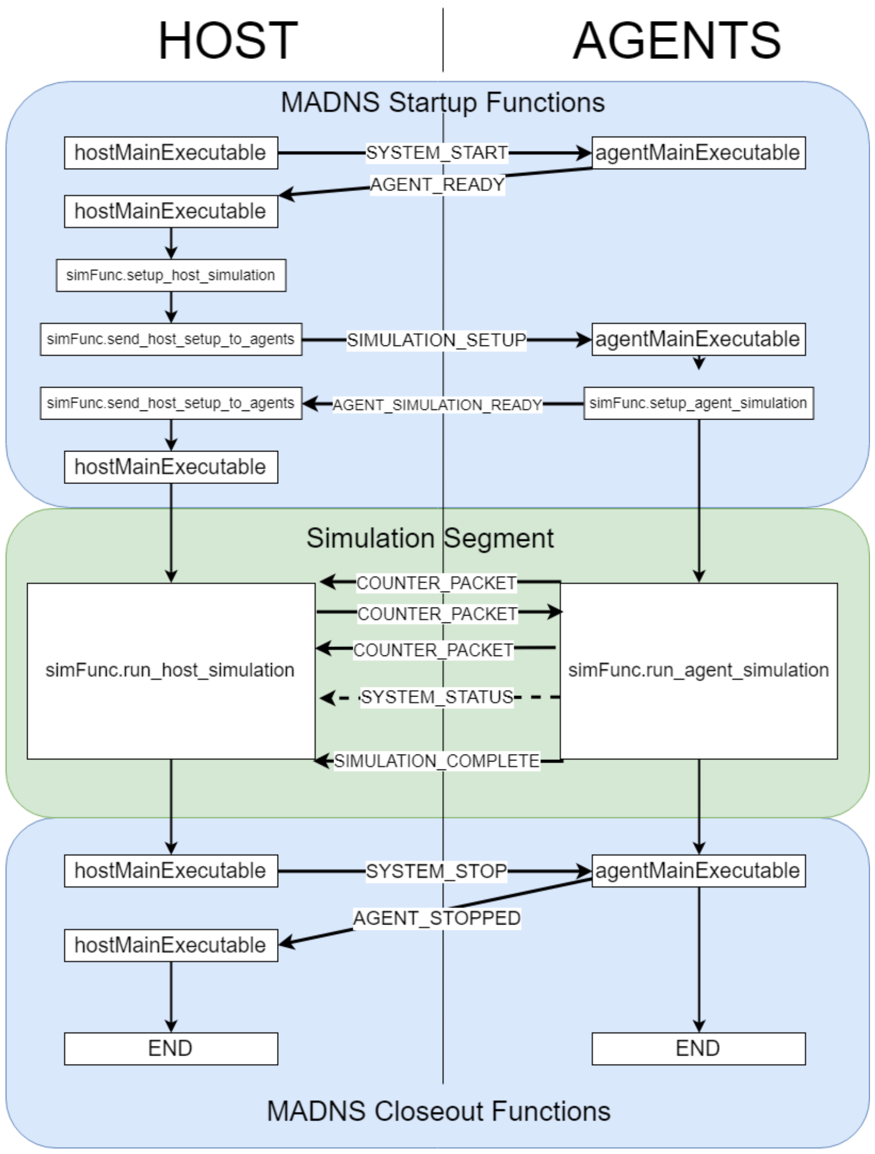


Figure 3.1: MADNS Benchmark Test Data Flow

the simulation data flow is segregated from the framework data flow. All MADNS data flows are represented by solid lines, which represent synchronous communication. Asynchronous communication paths are represented by dashed lines, the only asynchronous data items in the benchmark test are the system status packets generated by the SSA. Because the host simulation doesn't wait for or expect these packets, they can be sent at any time and are passed to the logging utility after clearing the simulation loop.

The segregation of the simulation segment, within the rounded box, gives simulation designers a wide latitude in structuring their data flows.

In this case, the "simulation" is the counting benchmark test discussed in Section 6.1. State packet communication for the benchmark tests is purely synchronous, which would usually cause scalability issues as discussed in Section 2.5. However, the "state propagation" algorithm in this case is `val += 1`, and therefore the "propagation time" T_p from Eqn. 2.3 is effectively 0. The only asynchronous data flows in the benchmark tests are the system status update packets sent from the agents at regular intervals. Finally, once every agent has completed its simulation task it sends a "SIMULATION_COMPLETE" packet to the host and the agent then waits for the host to close the system. After the host receives a "SIMULATION_COMPLETE" packet from every agent, the simulation segment is complete and the system returns to the MADNS framework to close out its operations.

3.2 Functional Design Schematics

Two functional design drawings are shown below representing the current state of the MADNS simulator and functions which were planned for the system but not yet implemented. Currently operational functions are presented inside boxes with a solid border and with a green fill, functions in development are represented by square boxes with a dashed border, and functions that are planned but not currently under development are represented by oval boxes with a dashed border. Only the first two categories will be discussed in this chapter while the planned functions will be discussed in Chapter 8

3.2.1 System Hardware

Currently, 24 SBCs have been integrated into the simulator in a single tower. These boards are all operational and accessible via the host computer. They are configured

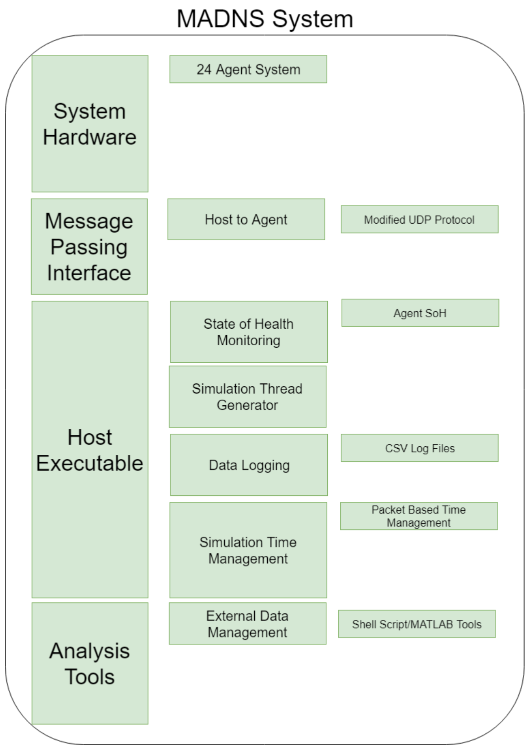


Figure 3.2: Current functional elements of the MADNS system.

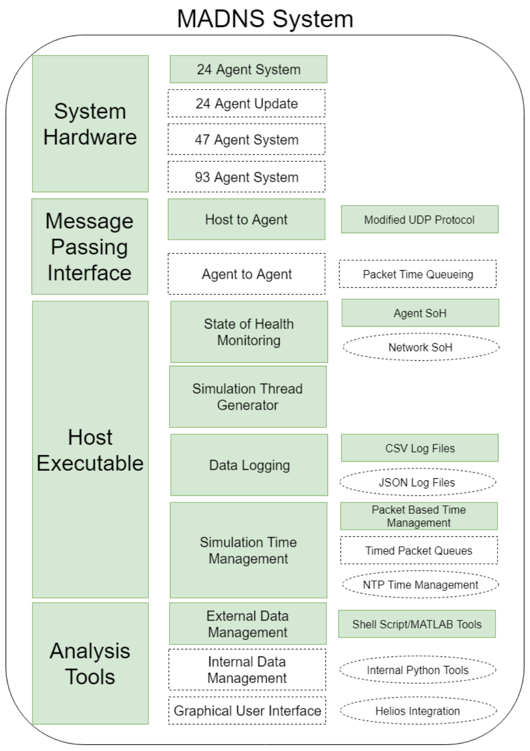


Figure 3.3: Planned functions of MADNS.

with fixed IP addresses on the MADNS internal network have had their Wi-Fi and Bluetooth capabilities disabled to increase performance and provide better system security.

3.2.2 Message Passing Interface

The custom message passing interface (MPI) is operational and provides the communications backbone for the simulator as described in Chapter 2. Existing MPI technologies, such the MPI and MPICH familiar to high-performance computing users, are not designed for applications where each agent is unique as discussed in Section 2.4. The MADNS MPI allows for the treatment of each agent as an unique individual. The MPI has proven reasonably robust in benchmark testing and actual operation as shown in Sections 6.1 and 6.2.

3.2.3 Host and Agent Executable

The host and agent executables were refactored between November 2017 and January 2018 to make the system as modular as possible. The primary functions are *hostMainExecutable.py* and *agentMainExecutable.py*, which are discussed in more detail in Appendix A.1.

3.2.4 Host to Agent Communication

The system has demonstrated its ability to pass messages across the network within real-time bounds. The MPI uses UDP sockets to send and receive messages, but the system will occasionally wait for confirmation of a packet being received in a hybrid TCP style. The current configuration has shown that it can operate with a packet round-trip time of < 50 and zero packet loss during benchmark testing as seen in Section 6.1 and previous work[32]. Additionally, the host-to-agent communication protocol can be used to connect to other systems through the Georgia Tech intranet. The current system uses the Python *Pickle* library due to its ease of use and despite its known security flaws [33].

3.2.5 State of Health Monitoring

Agents transmit state of health (SoH) packets to the host computer at regular intervals. These packets include statistics such as CPU usage, RAM usage, disk activity, temperature, network activity, and other relevant details. They are used to determine simulator performance and can help in the determination of simulation validity.

3.2.6 Simulation Thread Generation

The host and agent executables spawn threads using the Python Multiprocessing library. These threads communicate using Python *Multiprocessing.Queue* data structures for thread safety. However, this system has numerous drawbacks and needs to be reconsidered as discussed later in Section 8.2.1.

3.2.7 Data Logging

Currently, the system logs its data in plain text and CSV files. The amount of data created during operation is substantially less than in debug mode, but the MADNS system can creates more than 1 GB of data per hour with only 10 agents.

3.2.8 Simulation Time Management

Because the clocks between the host and agents are not synchronized, they system designates a *simT0* value which is used to reference all future times within the simulation. Because of network latency, there is no true way to set these T0 values simultaneously, but the MADNS system sets the host T0 before broadcasting command to the agents to set their T0 values. This leaves the host slightly ahead of the agents in simulation time, which is helpful in managing state propagation. Additionally, the simulation API developed through the creation of the benchmark test and COSMoS requires that agents send updates to the host with their current simulation time discretized to their operating rates. This keeps the host computer “aware” of where the agents are in simulation time and allows for better simulation synchronization with the eventual goal of allowing agent-to-agent communication.

3.2.9 External Data Management

Log files are gathered, organized, and packaged using a set of bash scripts at the end of a simulation. These files are processed using a suite of MATLAB scripts developed independently from the simulation code. Examples of current benchmark test output can be seen in a previously published work and in Section 6.1[32].

3.3 System Development Summary

The current development of MADNS has shown that the API can be used for two different simulations: the benchmark test and COSMoS. The MPI works well within its design specifications and has a simple interface where any arbitrary Python data object can be sent across the network using the *Pickle* protocol. The data logging system is currently able to store more that 1 GB/hour of data without any data loss and uses a similar interface to the MPI which allows for the storage of any data element through Python's *format* function. An exhaustive description of the functional elements and the data transfers can be found in Appendix A.

3.4 MADNS Real-Time Operations

MADNS is designed as a fundamentally real-time simulation framework. As such, its operational rate can be set by simulation designers as described in Appendix A.1.3. While these results will be discussed later in Chapters 6 and 7, benchmark testing has shown that the real-time operations are within expected parameters. Fig. 3.4 shows data gathered from a MADNS benchmark test which measured the round-trip time of a synchronous communications packet from the perspective of the host and the agents separately. This test was run at a fixed 20 Hz simulation rate. The round-trip time of 50 ms is exactly what would be expected from the 20 Hz simulation rate. Additionally, the 1-2 ms standard deviation seen in Fig. 3.5 is the expected σ value of Python's *time.sleep()* command, which is what enforces the real-time operations as described in Section A.1.3. This behavior is one of the most unqualified successes of the early MADNS system. The much larger σ value for agent 22 seen in Fig. 3.5 is the result of that agent being equipped with a hardware interface agent, and is discussed in later sections.

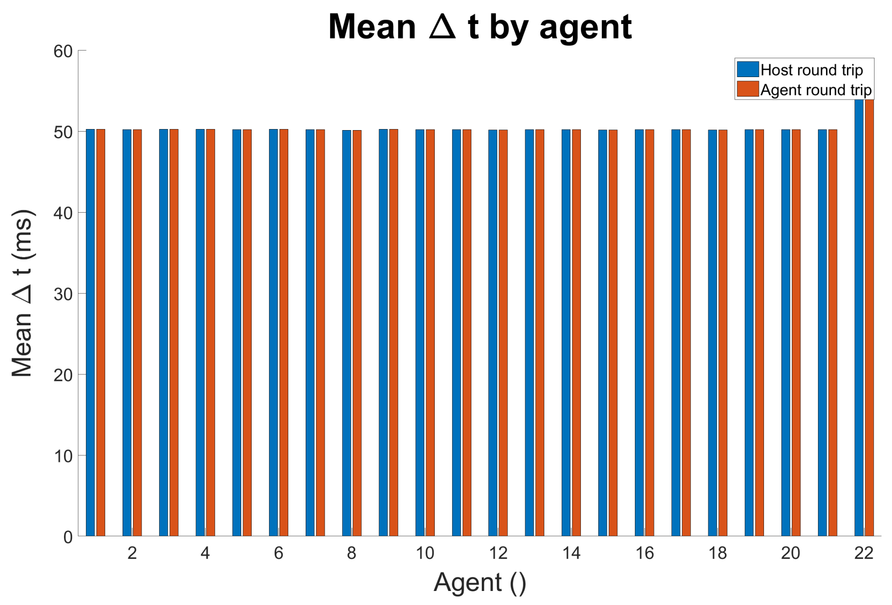


Figure 3.4: 8-hour test mean packet round-trip time

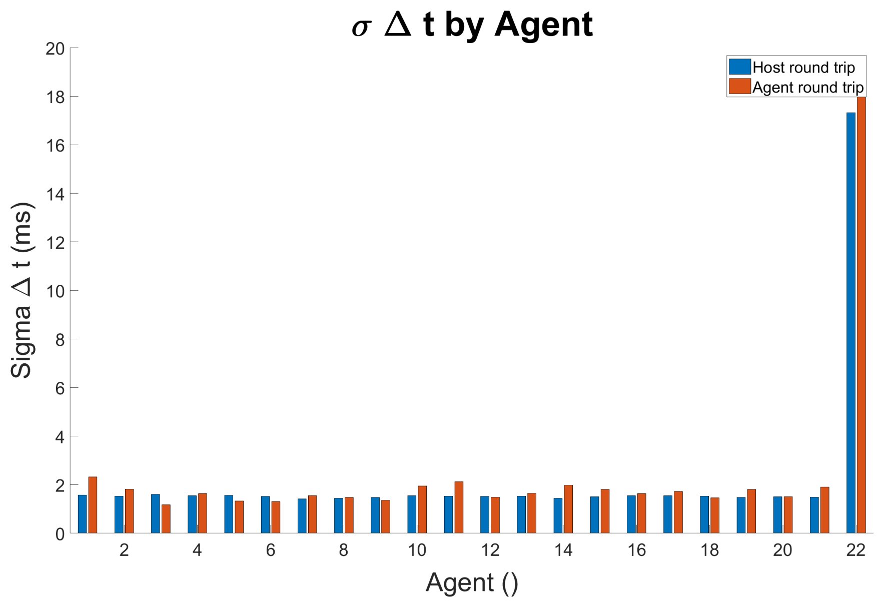


Figure 3.5: 8-hour test packet-round trip time standard deviation

CHAPTER 4
DEVELOPMENT OF COSMOS

4.1 Functional Design Schematics

A functional design drawing is shown below representing the development of both the COSMoS simulation. Currently operational functions are presented inside boxes with a solid border and green fill, functions with preliminary development complete are represented by square boxes with a dashed border and yellow fill, and functions that are planned but not currently under development are represented by oval boxes with a dashed border. Only the first two categories will be discussed in this chapter while the planned functions will be discussed in Chapter 8

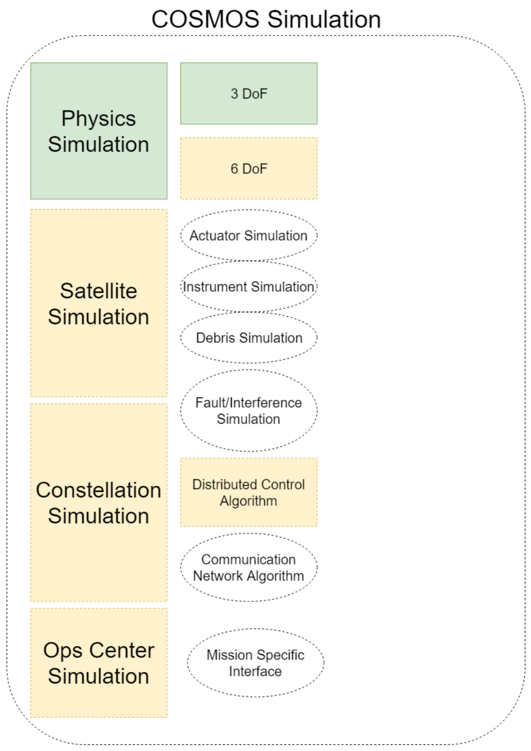


Figure 4.1: Conceptual design of the COSMoS system.

4.1.1 COSMoS

Physics Simulation: The COSMoS simulator is organized to incorporate external physics simulations. Currently, COSMoS uses the 2-body Keplerian orbit propagator created for the GT-SORT program. A more comprehensive 6-DoF simulation could easily be implemented by the incorporation of the Basilisk toolkit developed by the University of Colorado AVS lab. Unfortunately, there were substantial problems involved in the implementation of this toolkit as described in Section 8.2.2. However, the potential to incorporate different physics simulators is proven even though they may not be sustainable at an operational rate of 10 Hz as discussed in Section 8.2.

Satellite Simulation: The current satellite simulation is extremely basic, but it proves the viability of the system. The instrument simulation uses a Raspberry Pi Sense-Hat to simulate a hardware interface[34] on a single agent. This interface is discussed further in Chapter 5.

4.2 COSMoS Simulation Data Flow

As described in Section 3.1.1, the simulation data flow is largely segregated from the MADNS framework. The data flow for the COSMoS simulation is substantially different from the benchmark test as can be in Fig. 4.2. This figure only shows the simulation segment of Fig. 3.1 as the MADNS framework is identical to the benchmark test. Here, the simulation data flow is largely asynchronous with the host

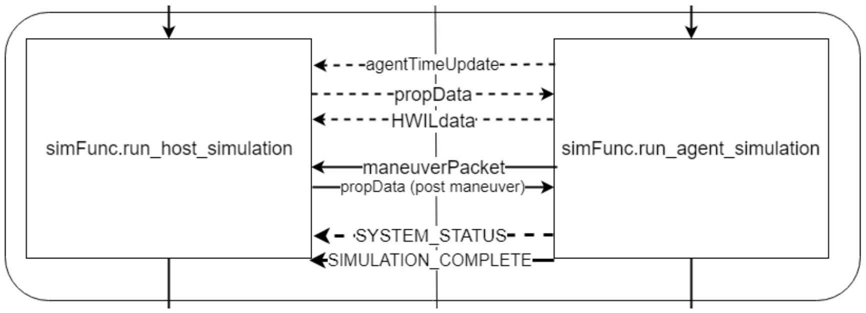


Figure 4.2: COSMoS Simulation Data Flow

sending periodic state updates (*propData*) to the agents and the agents reporting current simulation time (*agentTimeUpdate*) to the host. The only synchronous data process occurs when an agent sends a maneuver command to the host as discussed in Section 2.5. For the current version of COSMoS, those actuator commands are Δv values contained in a *maneuverPacket*. When this occurs, the host immediately re-propagates the state information based on the commanded Δv and transmits the newly propagated data back to the agent. Once this process is complete, the host resumes propagating on its asynchronous schedule and the agents continue sending simulation time updates at their simulation rates.

4.3 COSMoS Simulation Truth Propagation

COSMoS is designed to run at 10 Hz, giving an update rate of 0.1 seconds. However, early attempts to integrate Basilisk into COSMoS found that Basilisk would not propagate data in individual units of 0.1 seconds. To accommodate this behavior, the *HostSatelliteObject* and *AgentSatelliteObject* classes were developed in the *sattliteFunctions.py* file. These objects are described in greater detail in Section A.3.4, but their data management is of interest here. The orbit propagator is tasked with propagating 10 seconds of orbit in 0.1 second increments. A *HostSatelliteObject* stores those 10 seconds of data and marks a mid-point time at 5.0 seconds for later use. This initially propagated data is distributed to the agents in the "SIMULATION_SETUP" packets as shown in Fig. 3.1. This allows the system to begin its asynchronous propagation at the real-time simulation start to avoid major synchronous communication loads as discussed in Section 2.5.

When the real-time simulation commences, the host computer continually checks each satellite object to see if the simulation time has passed the midpoint time. As the agents report their simulation time, the host computer discards old states to limit the amount of data stored in its propagation buffers. Once an agent simulation time passes its midpoint, the host computer propagates another 10 seconds of data and appends it to the existing state data. Because the host has been discarding old state data, the propagated data now runs from approximately $t = 5.0$ to $t = 20.0$, and so the midpoint is now at approximately $t = 12.5$ seconds. This process repeats uninterrupted unless the agent transmits a maneuver command to the host.

The times described above are approximate by design and assist in managing the system scalability issues. The propagator currently in use on COSMoS takes between 0.01 and 0.03 seconds to propagate 10 seconds worth of data. Therefore, after at most 5 propagations, the host computer has already violated a 10 Hz real-time operating rate. However, that violation leads to the satellite object midpoint times for re-propagation drifting with respect to each other. On the first propagation cycle, every agent re-propagates at $t = 5.0$ seconds and then asynchronously sends the updates to the agents. However, on the second cycle the fist agent will re-propagate at $t = 12.5$ seconds, but the sixth agent will re-propagate on the next 10 Hz cycle at $t = 12.6$ seconds, and so on. Within the first minute of simulation, the agent re-propagation times will distribute themselves to reduce the number of real-time violations on the host computer. So long as the host's real-time violations do not create violates of the agent's real-time requirements they do not compromise the real-time operation of the system.

Additionally, in the current implementation the host computer does not log its state data. While this will need to be corrected in the future when actuator models are applied and the agents no longer receive noumenonological "truth" data, it currently reduces the amount of data generated by the system to a more manageable level. All data used to develop the COSMoS plots shown in Chapters 6 and 7 was recorded by the agents and collected at the end of each simulation. It should also be noted that a "manageable" level is still in excess of 1 gigabyte per hour

4.4 Contribution Summary

The following measurable benchmarks were originally proposed for Contribution 2:

1. Demonstrate the ability to run a multi-hour, 3-DoF, real-time simulation using each available SBC as a single satellite.

2. Demonstrate the ability to test a guidance algorithm, independent of other flight code elements, running an agent.

3. Demonstrate the ability to run an analysis framework simulation data in post-processing.

Of these, the first was a partial success limited by the selection of programming language as described in Section 8.2.2. The second is also a partial success as the ability to have agents respond to individual guidance controls has been demonstrated, but there were no tests of full GNC algorithms or large scale simulations. The results of these tests can be seen in Section 6.2.1. The third is a full success as demonstrated by the results seen in Chapter 6.

CHAPTER 5
CONTRIBUTION 3: HARDWARE-IN-THE-LOOP INTEGRATION

The core function of MADNS is to serve as a platform for real-time hardware-in-the-loop (RT-HWIL) simulation on a large scale. Most other similar systems rely on standard distributed computing models, which are highly developed and more efficient that the model implemented on MADNS. However, these models are not designed for real-time operations and are certainly not designed for a real-time hardware interface.

The real-time operations are handled by the rate limiter library described in Section A.1.3. The HWIL aspect is currently demonstrated through the use of a Raspberry Pi Sense-HAT add-on board[34]. The Sense-HAT is equipped with a 3-axis gyroscope, 3-axis accelerometer, and 3-axis magnetometer, as would be expected on a spacecraft IMU, as well as temperature and pressure sensors. While the Sense-HAT is not as fast or powerful as one might expect true spacecraft sensors to be, it does provide proof of concept for integrating hardware devices into the MADNS framework and COSMoS simulation.

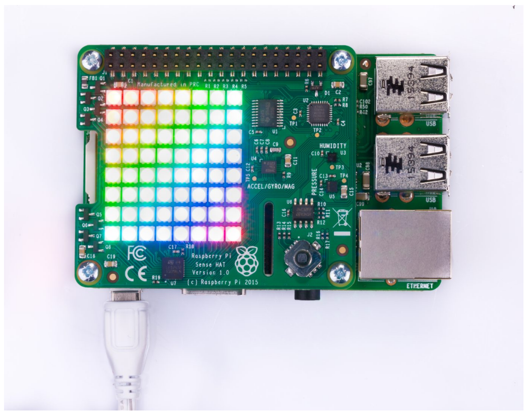


Figure 5.1: Sense HAT[34]

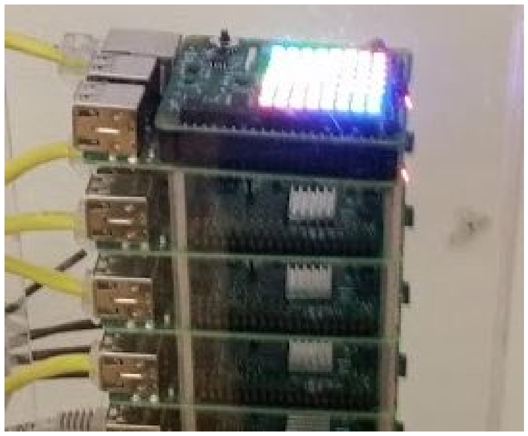


Figure 5.2: Sense HAT installed on MADNS.

5.1 Sense Hat Capabilities

The Sense HAT comes equipped with the following sensors[35]: For the purposes

Sensor	Range	Accuracy
3-axis linear accelerometer	$\pm2/\pm4/\pm8/\pm16\ g$	$\pm90\ mg$
3-axis magnetometer	$\pm4/\pm8/\pm12/\pm16$ gauss	±1 gauss
3-axis rate gyroscope	$\pm245/\pm500/\pm2000$ deg/s	±30 dps
Pressure sensor	260-1260 hPa	±1 hPa
Temperature	0 - 80^{o}C	$\pm1^o$C
Humidity	0 - 100 %rH	±4.5 % rH

of testing MADNS and COSMoS, the accelerometer, magnetometer, gyroscope, pressure, and temperature sensors were queried to create a hardware interface data packet for the simulation. These queries took more computing time than expected and introduced the system delays mentioned in Section 3.4. These delays are discussed in more detail in Section 8.2.1.

It is important to note that the Sense-HAT is not a true hardware-in-the-loop element as it is not "in-the-loop". If the Sense-HAT were being used in a true HWIL sense, it would need to be attached to an external device, such as a Helmholtz cage or centrifuge. The external device would then receive state data from the host and would create a simulated environment (such as a magnetic field) to give the Sense-HAT's sensors the expected operational environment signals. Alternately, a device such as a GPS Simulator could be used in place of the Sense-HAT which would create a signal fed either directly to the SBC or to an intermediate device such as a GPS receiver, and that data would then be passed to the FSW running on the SBC. The Sense-HAT in this test is purely open loop, but it demonstrates MADNS's ability to take in high rate hardware data, process it, and transmit it to the host without impeding other operations.

5.2 HWIL Libraries and Drivers

A major advantage of using the Sense-HAT was avoiding the need to code hardware interface drivers. Sense-HATs come with a Python ready *sense_hat* interface module with an extremely simple API. Even in the absence of such a pre-built driver library, the selection of Raspberry Pi SBCs is a major advantage in creating hardware interfaces. The Rasperry Pi was designed with a large number of GPIO digital and analog I/O pins, and so any device should be easily to integrate so long as it can communicate through 5V input/output channels. Raspberry Pis can also be outfitted with adapters for more advanced communication protocols, such as RS-422, using commercial off-the-shelf products (COTS).

5.3 MADNS HWIL Interface

The current MADNS HWIL interface is extremely simple and is found in the *hwilUtilities.py* file. The MADNS API requires the following functions in the interface library:

- *setup_hwil_interface*, which creates a hardware interface object callable by the simulation
- *hwil_query_and_send*, which queries the hardware interface object and then routs its return data elsewhere in the simulation
- *hwil_process*, a currently deprecated function originally designed to spawn a parallel HWIL communication process

Any other functions found in the HWIL library are simulation specific.

In the current version of COSMoS, the functions are extremely simple thanks to the pre-built Sense-HAT interface. For example, *HWIL.setup_hwil_interface* reads as:

```
def setup_hwil_interface():
        sense = SenseHat()
        sense.set_rotation(0)
        # Enable gyro, accel, and mag
        sense.set_imu_config(True, True, True)
        return sense
```

In a case where the hardware driver had to be built for the simulation, this would necessarily be much more complicated. The *HWIL.hwil_process* function was designed to operate similarly to the network communication and logging processes described in Section A.1, however this was one of the first to fall victim to the Python multiprocessing issues described in Section 8.2.1. In most applications, sensors would be run in parallel with a main process and queried when data was needed, however because of the limitations found in Python, the Sense-HAT is queried in serial with the agent process as described in Section A.3.2. This has caused some minor performance issues which are discussed in Chapter 7, but otherwise the demonstration of MADNS's ability to perform RT-HWIL operations is a complete success.

5.4 Benchmark Summary

The following measurable benchmarks are proposed for Contribution 2:

1. Demonstrate the ability to interface with an external hardware device.

2. Analyze recorded hardware measurements in post processing.

This task was a full success as shown in Chapter 7.

CHAPTER 6
MADNS AND COSMOS RESULTS AND VALIDATION

Several simulations were run on the MADNS system to test its performance and demonstrate its ability to provide the RT-HWIL simulation capacity that satellite mega-constellations will require. These tests include benchmark testing using a simple counting algorithm, hardware interface tests, and satellite simulation tests using the COSMoS simulation framework. The results of these tests are presented and discussed below.

6.1 Real-Time Operation Benchmarking

The benchmarking algorithm used on MADNS is a simple counting function. The host computer creates a set of packets with a stored value of 0 in the *simFunc.setup_host_simulation* step, which distributed to the agents using *simFunc.send_host_setup_to_agents*. When an agent reaches the "SIMULATION_START" phase, it begin incrementing the packet value by 1 using *simFunc.incerement_counter* and transmits the packet back to the host. The host receives these packets and uses the same *simFunc.increment_counter* function to increment them packet after which it passes the incremented packet back to the appropriate agent. This test continues until each agent hits the maximum simulation time specified in *localConfig.py*.

The following benchmark tests were run for 3 minutes, 6 minutes, 1 hour, and 8 hours. Results shown here include the average round-trip time for the packets and the standard deviation of the round-trip time. A complete data set for each test can be found in Appendix C.

6.1.1 3-Minute Test

The first test ran for 3-minutes with the simulation rate set at 20 Hz in *localConfig.py* and generated the data presented below.

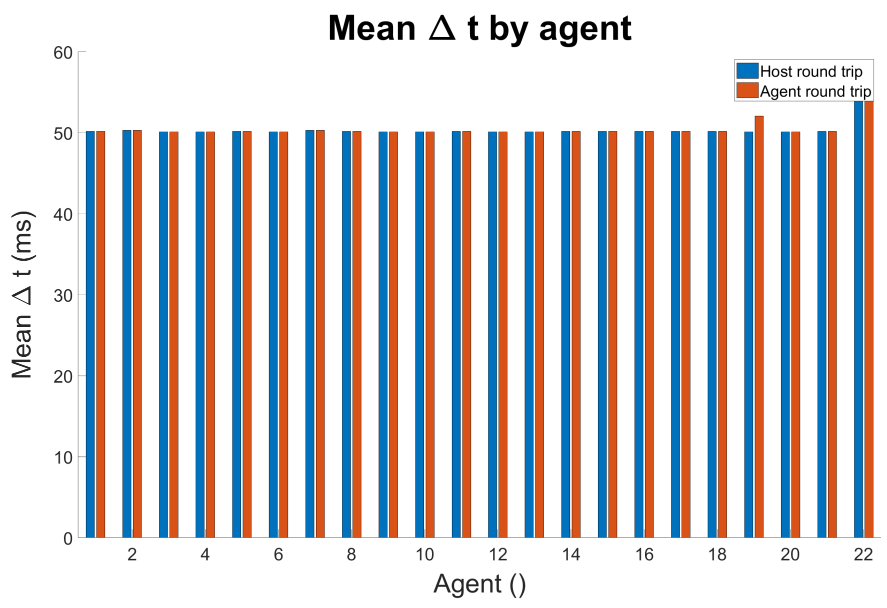


Figure 6.1: Mean packet round-trip time

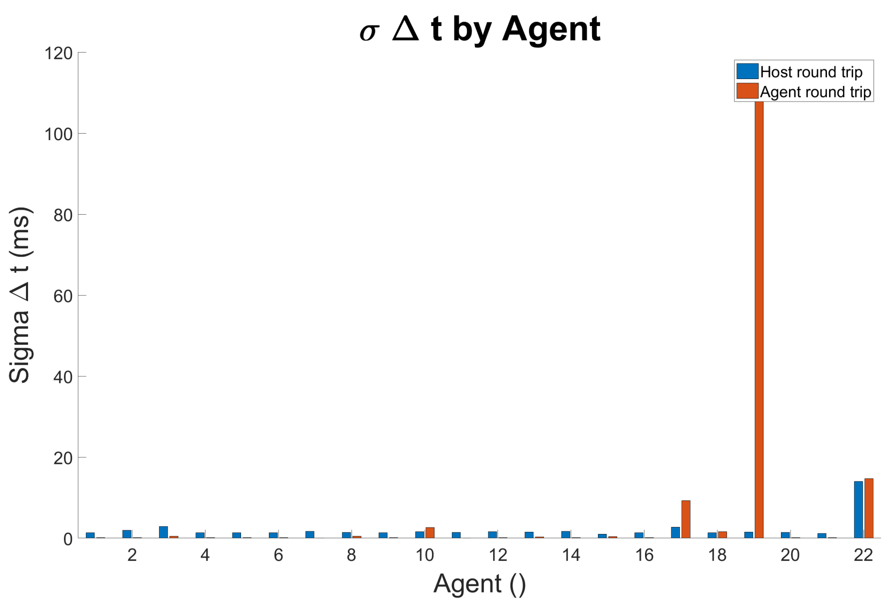


Figure 6.2: Packet-round trip time standard deviation

The round-trip packet time nearly perfectly matches the desired 20 Hz operation rate with the exception of Agent 22, which was equipped with the Sense-HAT. The Sense-HAT introduced processing delays on the agent as discussed in Section 8.2.3. The processing delays seen on Agent 19 are likely random nose as will be demonstrated in the following tests. This test created 127 MB of data.

6.1.2 6-Minute Test

Another test was performed for 6-minutes with the simulation processing rate still set at 20 Hz. The results are seen below:

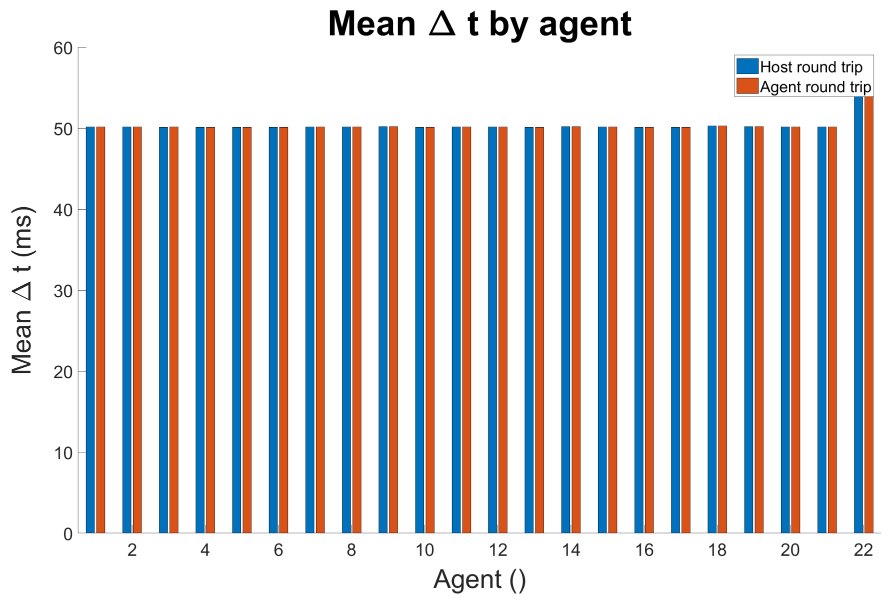


Figure 6.3: 6-minute test mean packet round-trip time

Here it becomes clear that the issues experienced by Agent 19 in the 3-minute test were simply the result of random noise while the delay on Agent 22 is clearly a part of the system. This test generated 249 MB of data.

6.1.3 1-Hour Test

A test was performed, again at 20 Hz, for one hour to confirm that the system could sustain operations. The results are seen below:

This test generated 2.4 GB of data.

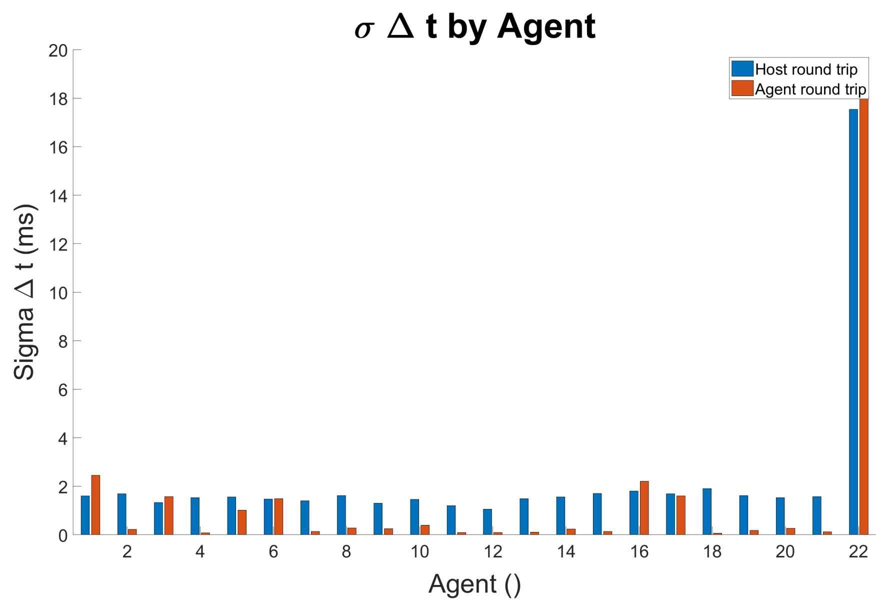


Figure 6.4: 6-minute test packet-round trip time standard deviation

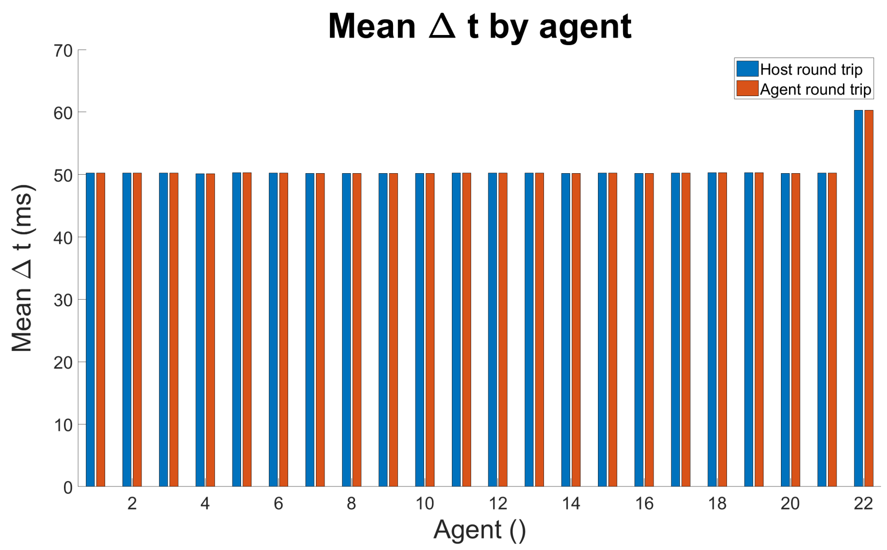


Figure 6.5: 1-hour test mean packet round-trip time

6.1.4 8-Hour Test

A final test was performed for eight hours to prove the system's robustness. A full set of the results can be found in Appendix C with a limited set shown below:

The system demonstrated nearly identical results to the shorter tests. Additionally, core temperature readings were recorded by the system status agent (SSA) and

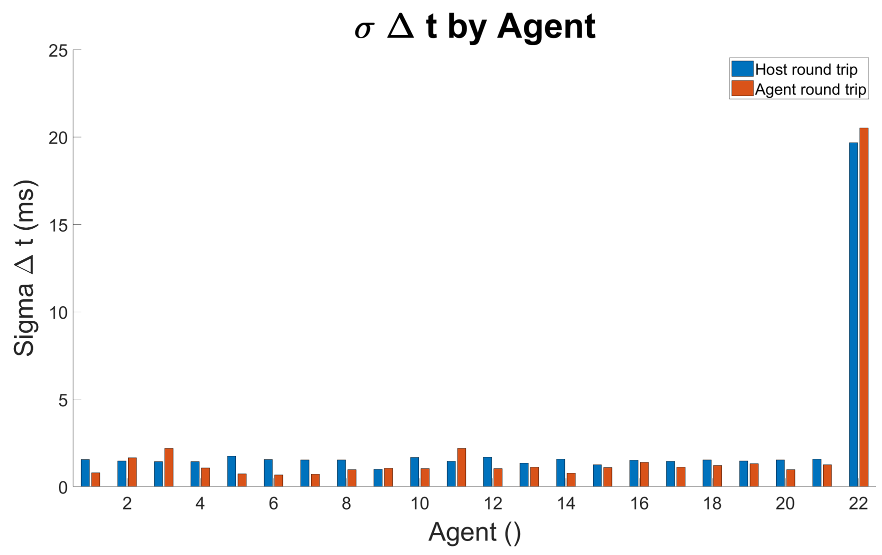


Figure 6.6: 1-hour test packet-round trip time standard deviation

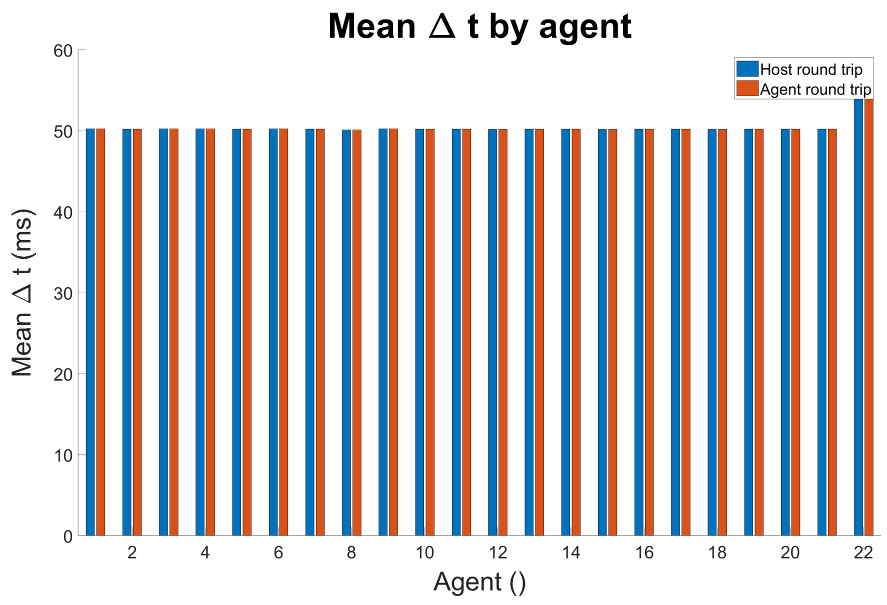


Figure 6.7: 8-hour test mean packet round-trip time

showed the temperature performance of the SBCs to be well within their operating parameters.

This test generated 19.1 GB of data.

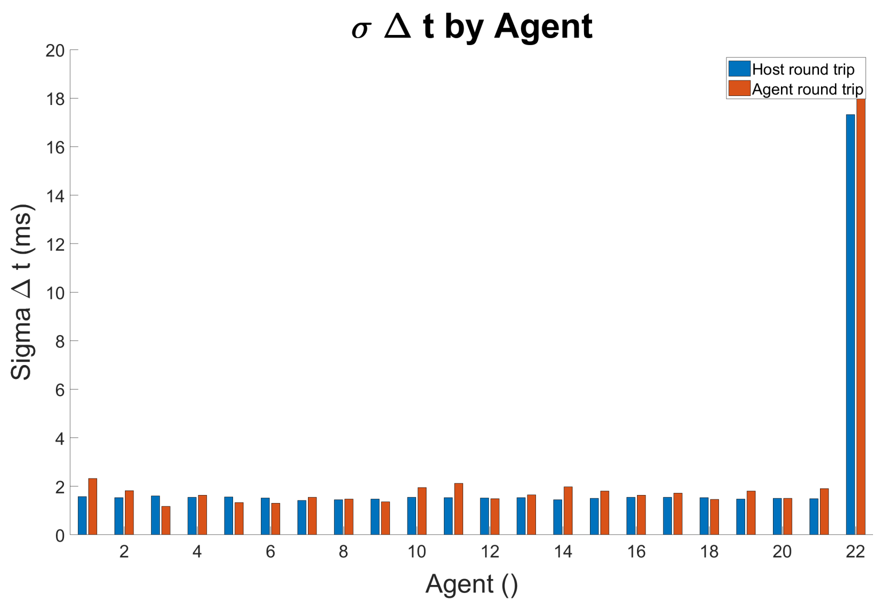


Figure 6.8: 8-hour test packet-round trip time standard deviation

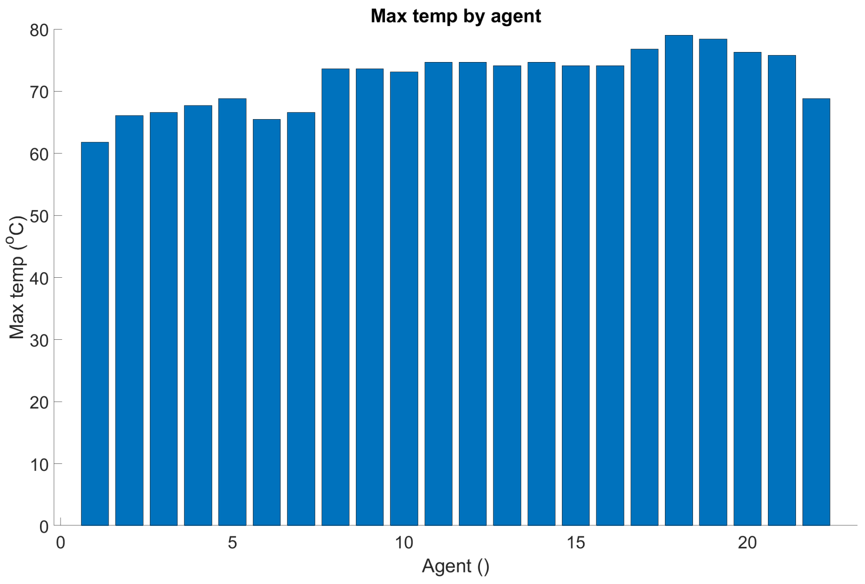


Figure 6.9: 8-hour test maximum temperature by agent

6.2 COSMoS Satellite Simulations

The COSMoS simulations were run using only 10 agents due to a deficiency discovered in the MPI when packet sizes became larger than the 3 floating point values used for benchmark testing. However, the results show system's capability including its ability

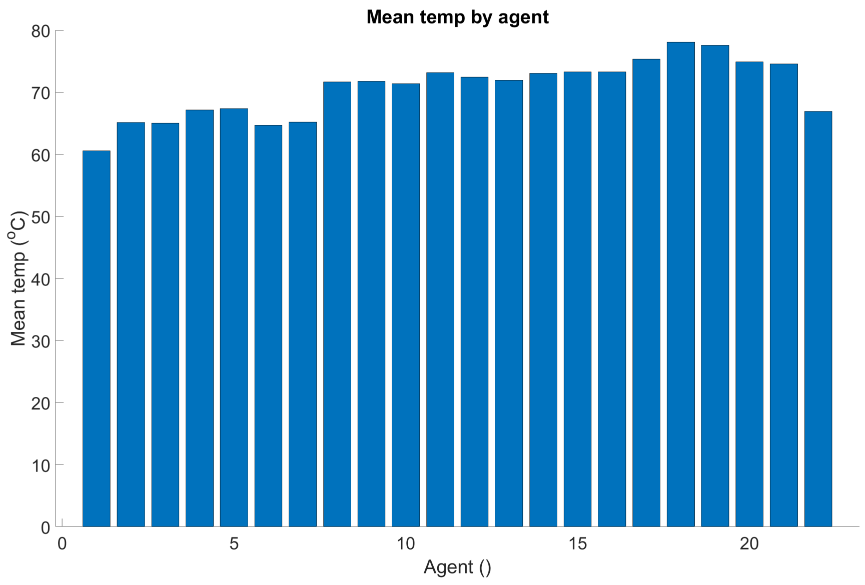


Figure 6.10: 8-hour test mean temperature by agent

to process HWIL interface data while simultaneously propagating orbits. The HWIL results are discussed further in Chapter 7.

The satellites selected for these simulations are IRIDIUM 8, 7, 6, 5, 4, 914, 12, 10, 13, and 16. Satellite initial states were generated from the Celestrak TLE database using TLEs from 11/05/2017.

Tests were run using 3 minute, 6 minute, 1 hour, and 4 hour propagation times. The data from the short runs matches that from the longer ones and will not be presented here. All orbit data was propagated from the host computer and distributed to the agents as described in Chapter 4. The data presented here was "processed"' and recorded on the agents with no data being stored from the host. The agent "processing" in question involved running the current **r** and **v** vectors through a guidance algorithm which is shown below:

```
def guidance_algorithm(r, v):
    return 0
```

However, it should be simple enough to replace that function with a more active GNC algorithm.

The state data was then recorded on the agent and collected by the host at the end of the simulation. While no noise or measurement models were applied to the data in this case, the implementation of a sensor noise model between the propagator and agent would be a relatively easy task. All data analytics shown below were performed in post-processing. A sample of the 5.5 GB of data collected from the 4-hour run is shown below:

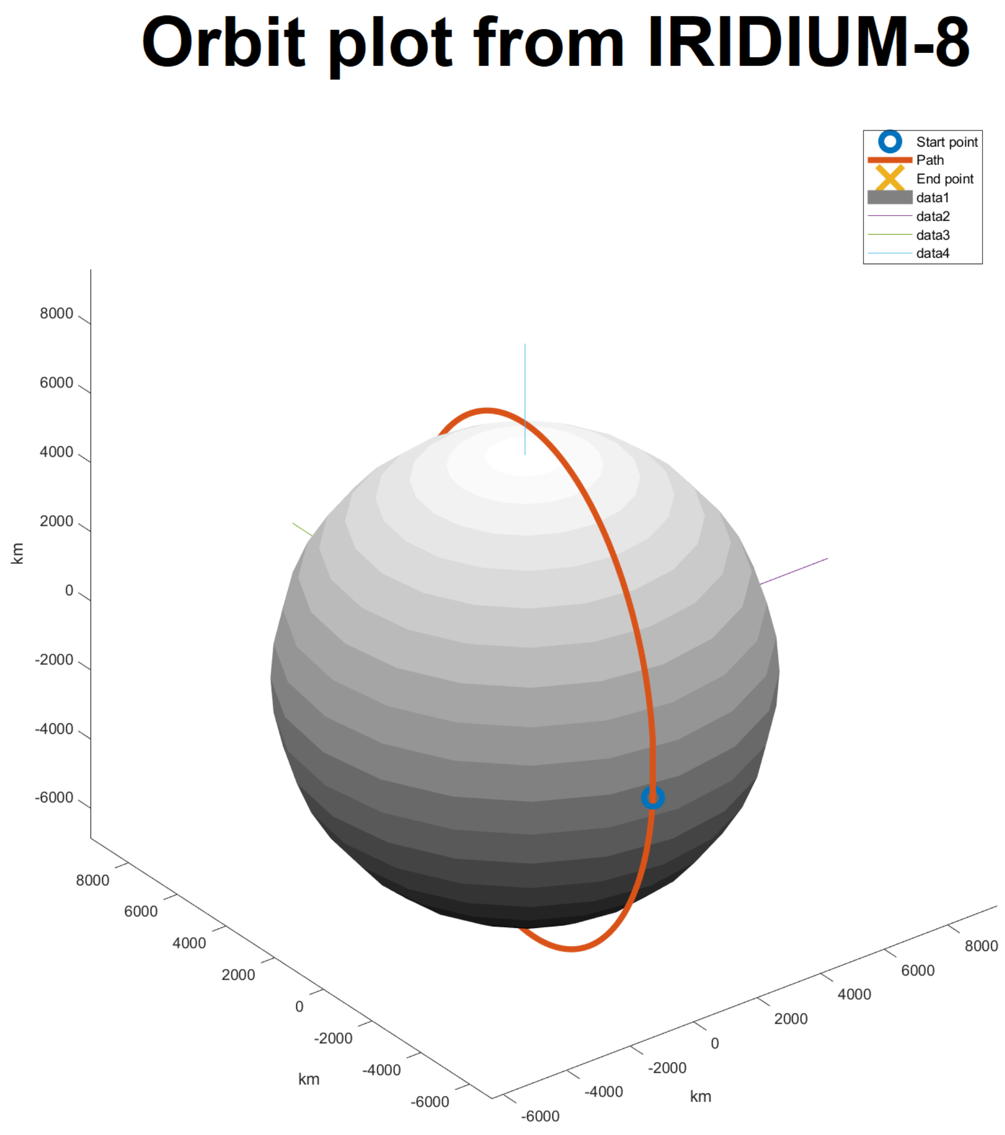


Figure 6.11: IRIDIUM-8 4-hour orbit plot

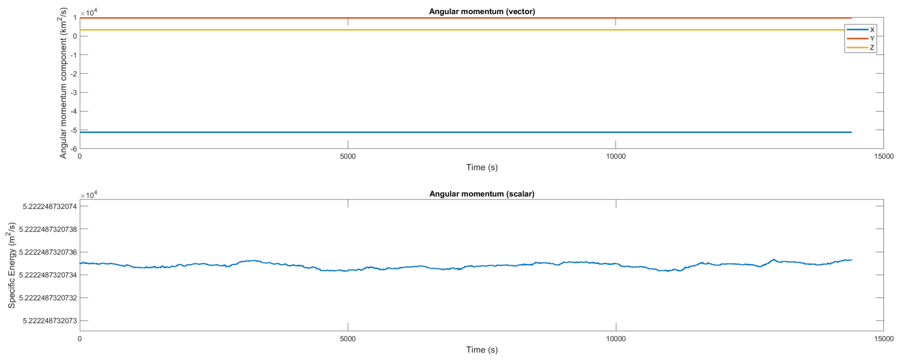


Figure 6.12: IRIDIUM-8 4-hour angular momentum

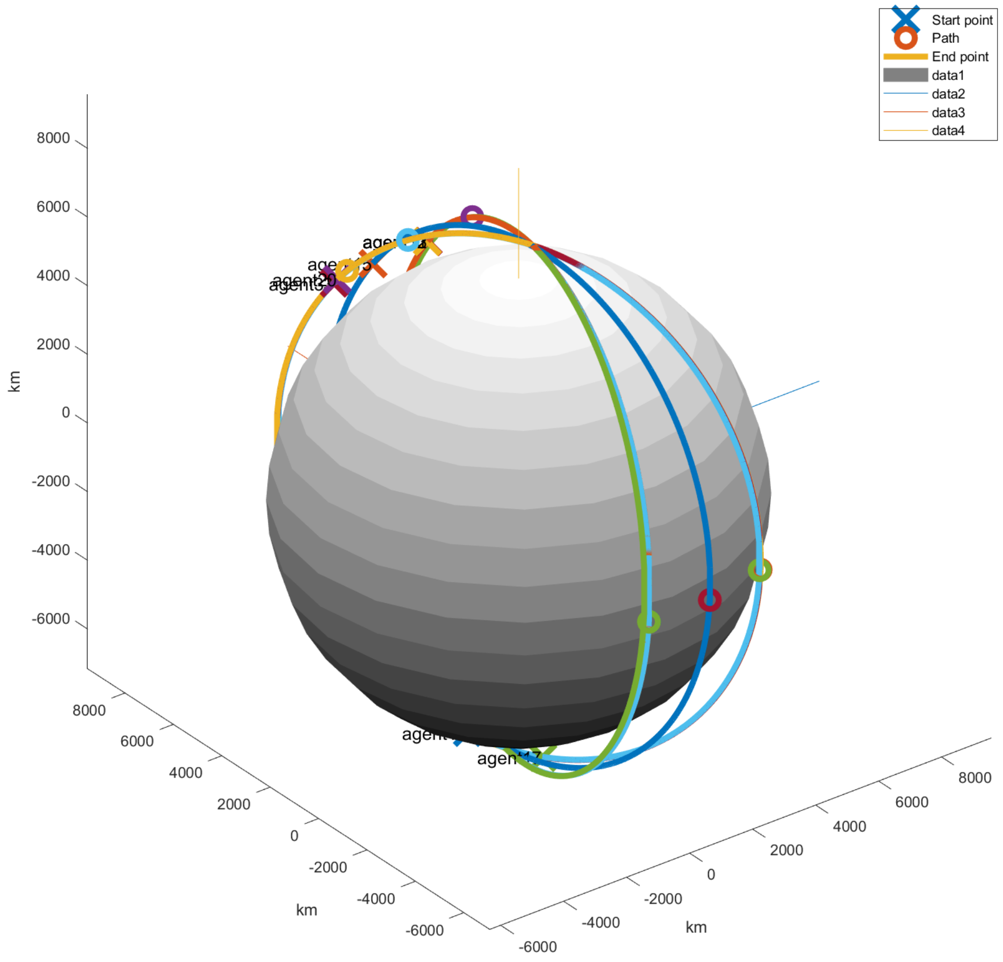


Figure 6.13: COSMoS 4-hour simulation all orbits

6.2.1 Simulation with primitive guidance algorithm integration

The guidance algorithm implemented here was extremely basic. The algorithm executed at a prescribed maneuver time and injected a satellite into a Hohmann transfer orbit with a target apogee of 42,164 km. Because the Iridum satellites are in polar orbits this is not a true Geostationary Transfer Orbit (GTO), but it is a useful stand-in for the purpose of testing the COSMoS guidance capabilities. Unfortunately, this algorithm also introduced a bug which caused the system to fail if more than two agents were run simultaneously. That bug has yet not been resolved, although work is in progress. The full set of results from this test can be found in Appendix D, but a subset is shown below:

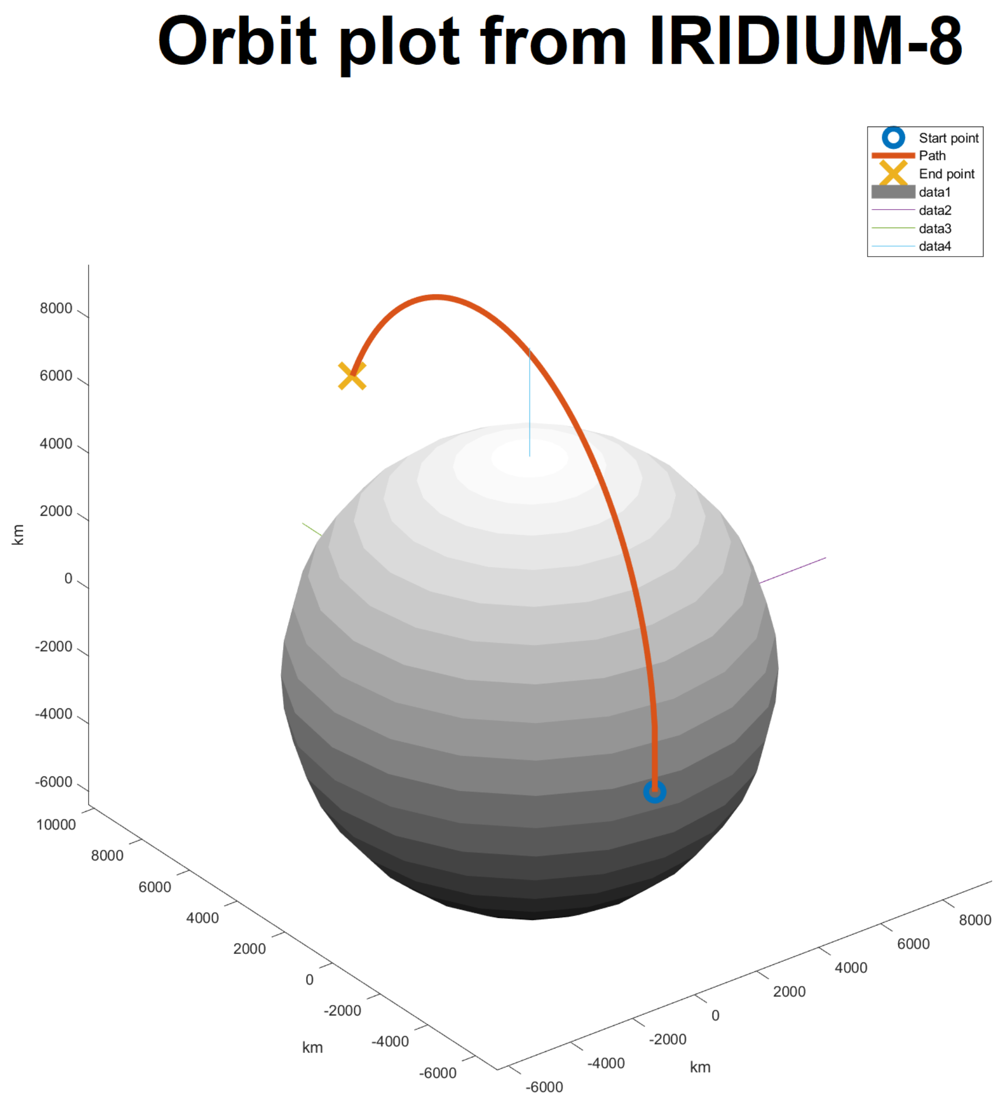


Figure 6.14: IRIDUM-8 1-hour orbit plot with guidance command

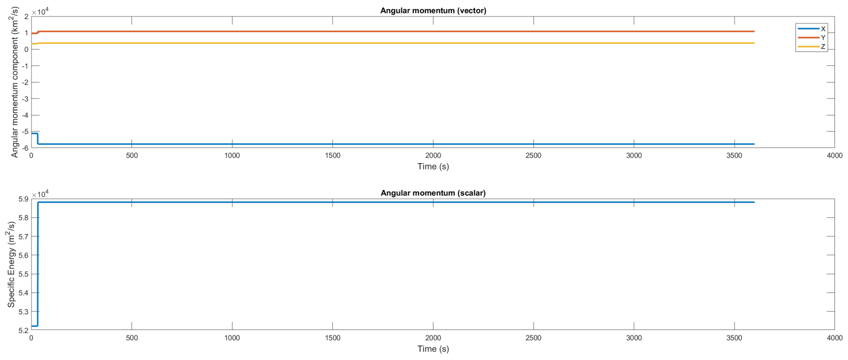


Figure 6.15: IRIDIUM-8 1-hour angular momentum with guidance command

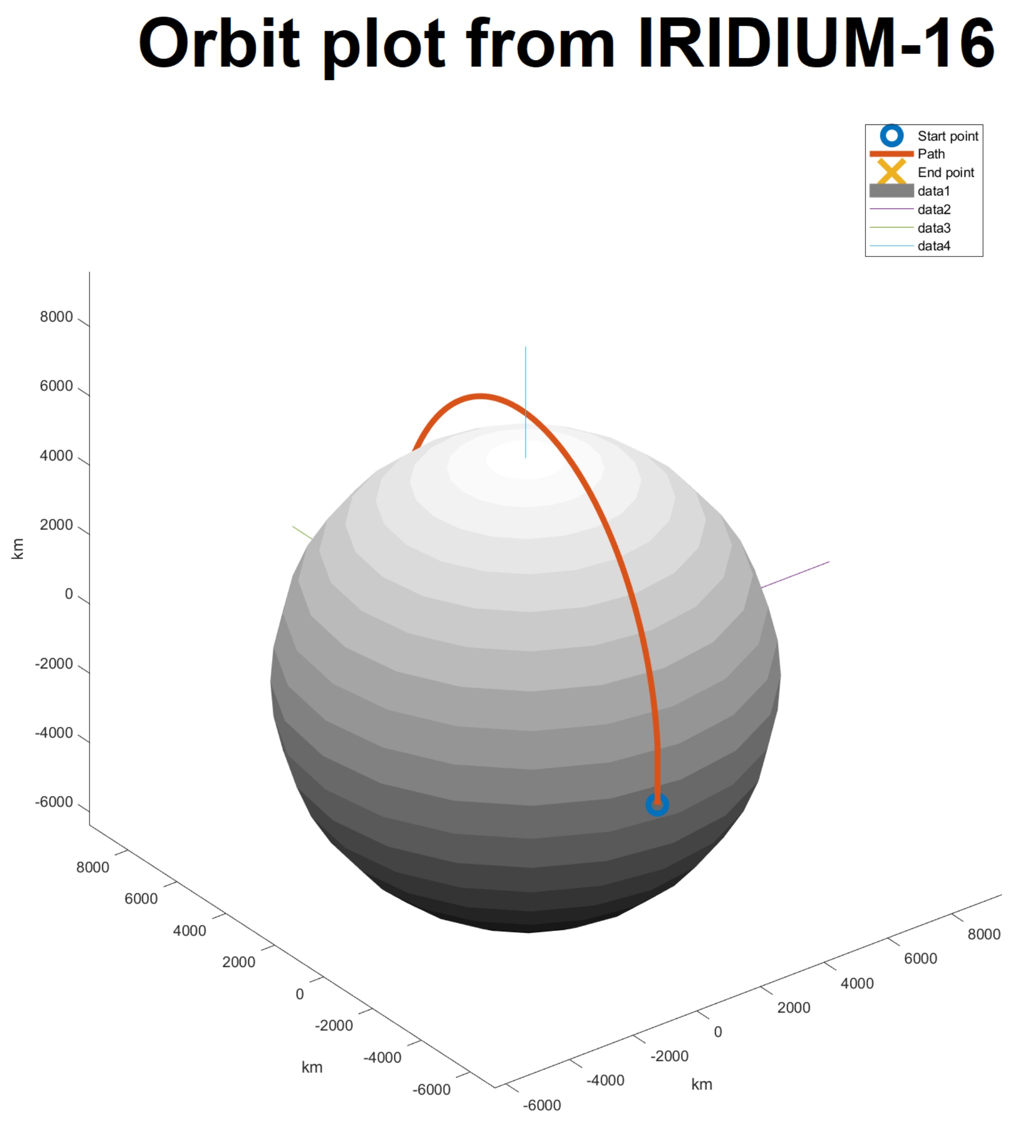


Figure 6.16: IRIDUM-16 1-hour orbit plot without guidance command

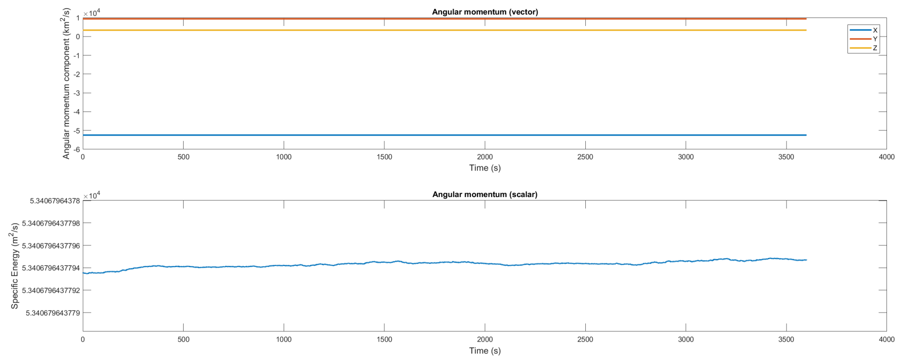


Figure 6.17: IRIDIUM-16 1-hour angular momentum without guidance command

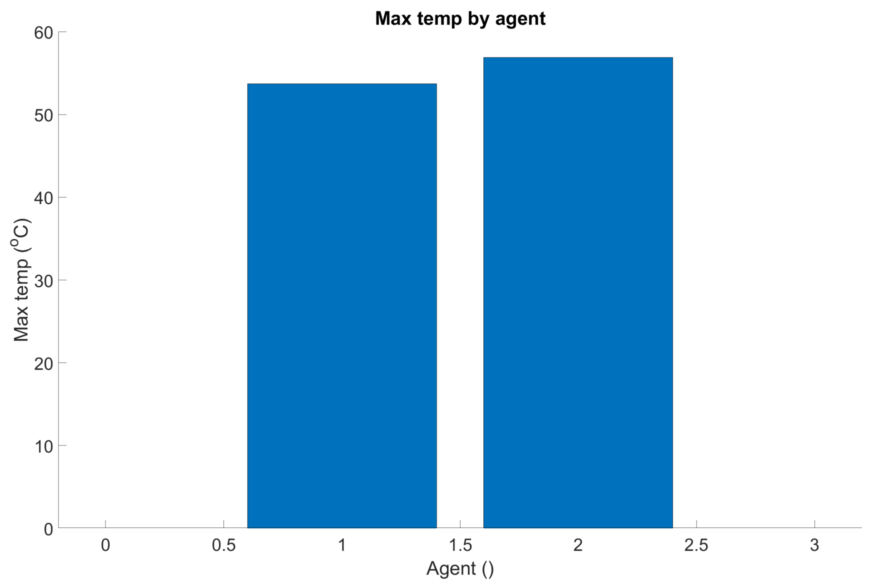


Figure 6.18: 1-hour COSMoS test maximum temperature by agent

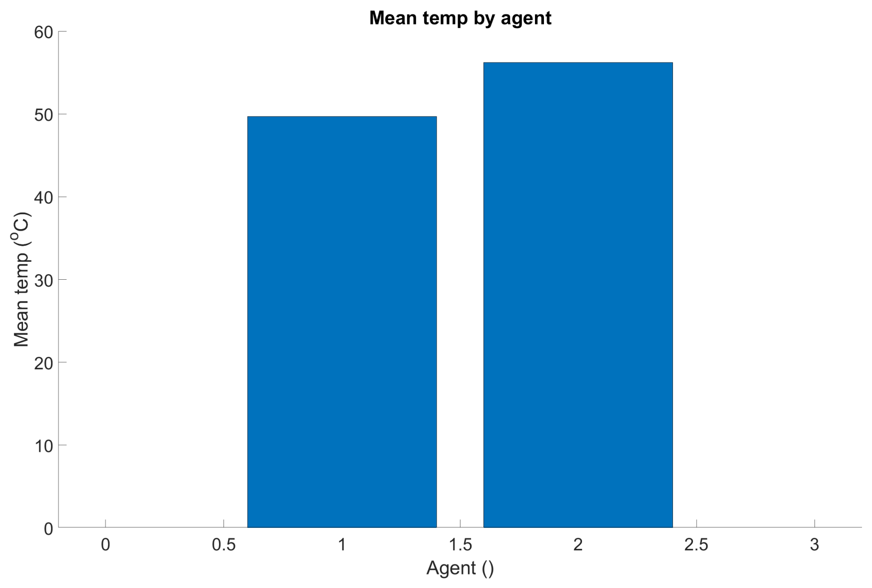


Figure 6.19: 1-hour COSMoS test mean temperature by agent

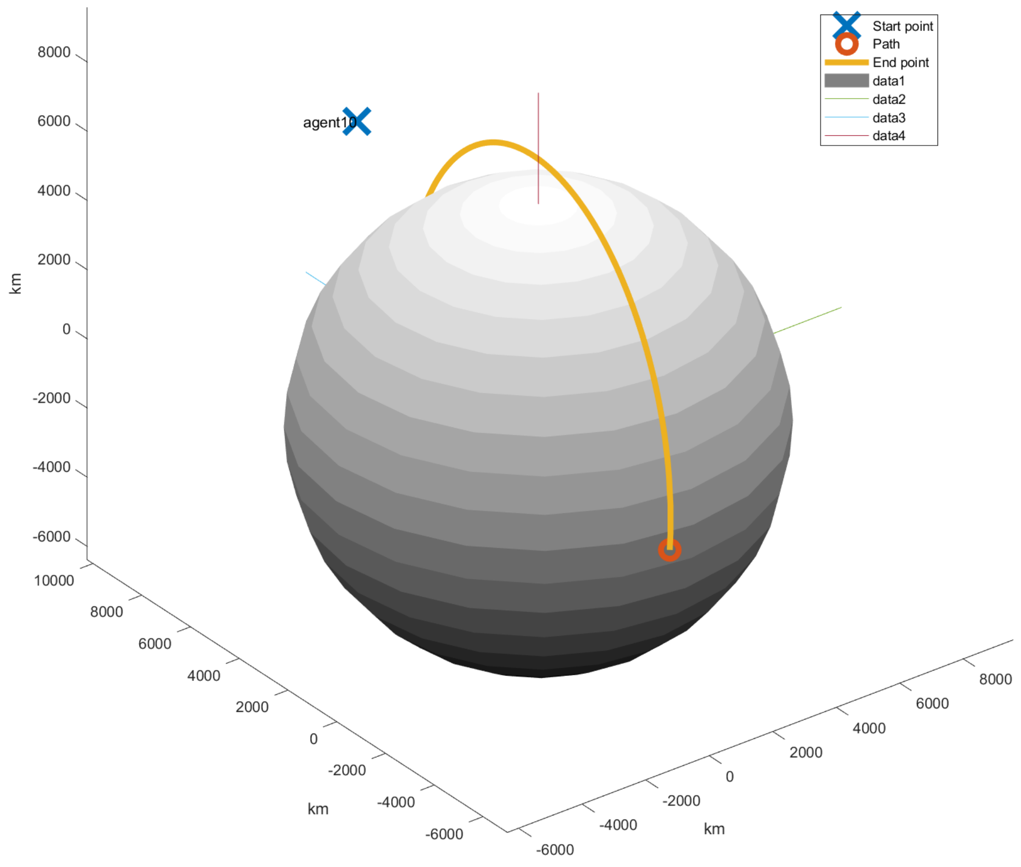


Figure 6.20: COSMoS 1-hour simulation all orbits

CHAPTER 7
HARDWARE-IN-THE-LOOP RESULTS AND VALIDATION

All of the tests described in Chapter 6 were HWIL enabled with the Sense-HAT connected to the top device in the SBC tower. Every test showed a performance reduction on that device including multiple real-time operations violations, but this is due to a known issue described in Section 8.2.3. The data collected from the Sense-HAT is presented below for the MADNS benchmark tests and COSMoS satellite simulations.

7.1 Hardware Interface Tests

Data was recorded from the Sense-Hat during each of the MADNS benchmark tests described in Section 6.1. The original plan was to sensor readings at rate of 10 Hz separate from the main agent process, but the issues described in Section 8.2.3 prevented this, and so the actual sampling rate was lower than the 20 Hz simulation rate. However, the data was recorded from agent 33 and transmitted to the host computer where it was logged to demonstrate the robustness of the data transmission and logging system.

The data recorded includes readings from the 3-axis gyroscope, 3-axis accelerometer, 3-axis magnetometer, temperature, and pressure sensors. The sixth plot shows the time delay in simulation time between when the first reading was taken on the agent and when the host computer logged the transmitted packet. The long delay seen on the first packet in each case is the result of how the benchmark test was triggered by the host computer and when the host *simT0* was reset.

The slow, gradual temperature increase seen on the 8-hour run is entirely reasonable because the run was started around 2 AM Atlanta time and completed around

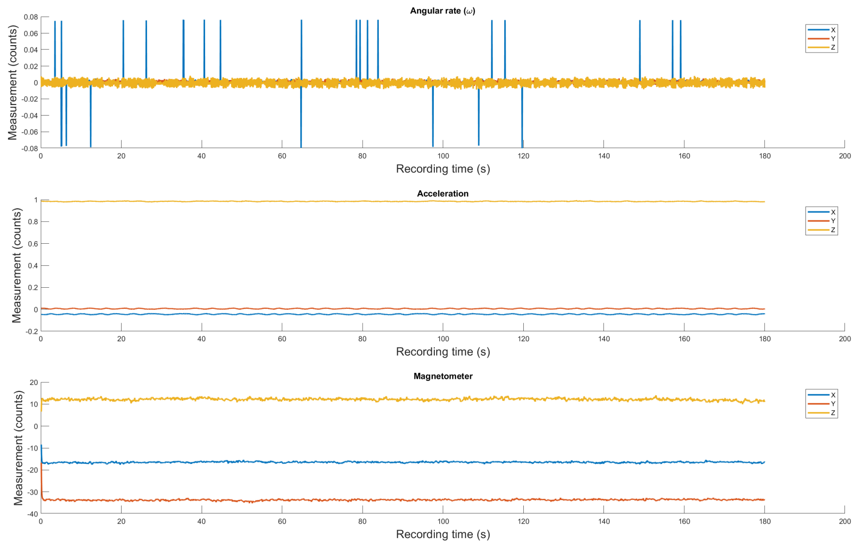


Figure 7.1: 3-minute gyro, accel, and magnetometer measurements

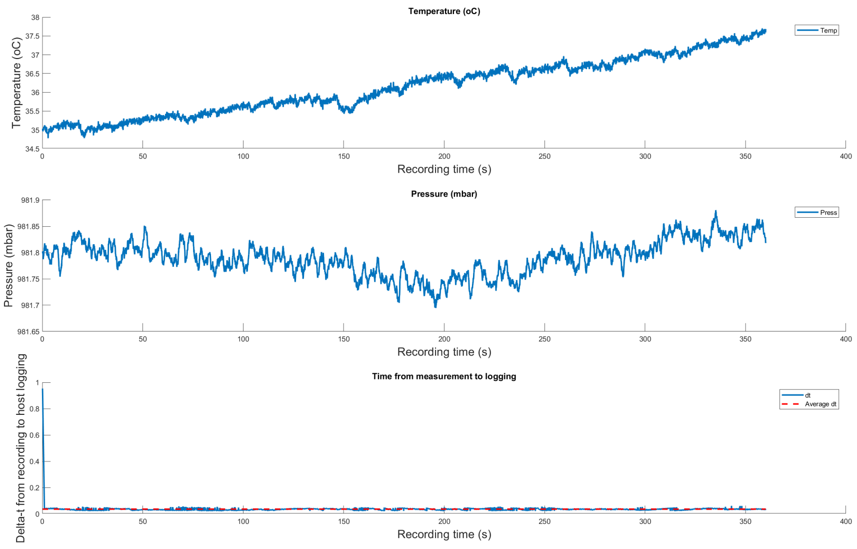


Figure 7.2: 3-minute temp, pressure, and δt measurements

10 AM. As the run was initiated from Albuquerque, there is no easy confirmation of the ambient room temperature in ESM-101 during that period, but the behavior seems reasonable.

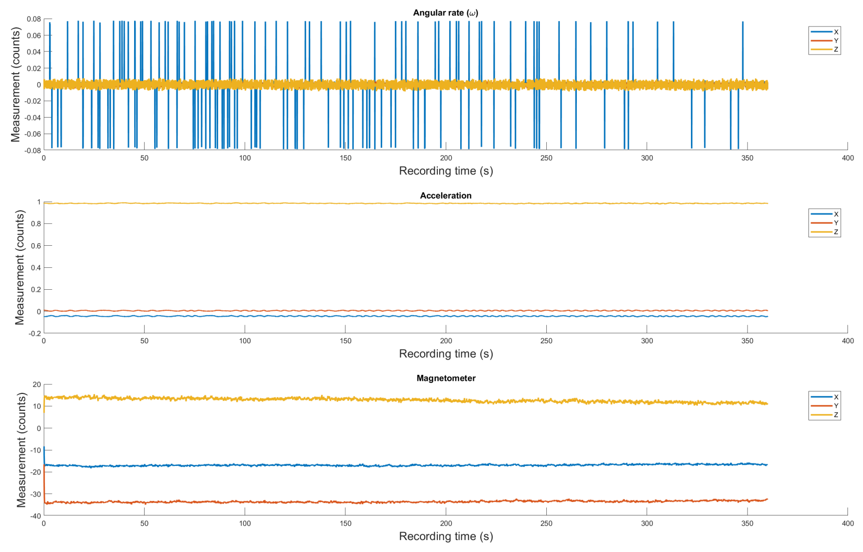


Figure 7.3: 6-minute gyro, accel, and magnetometer measurements

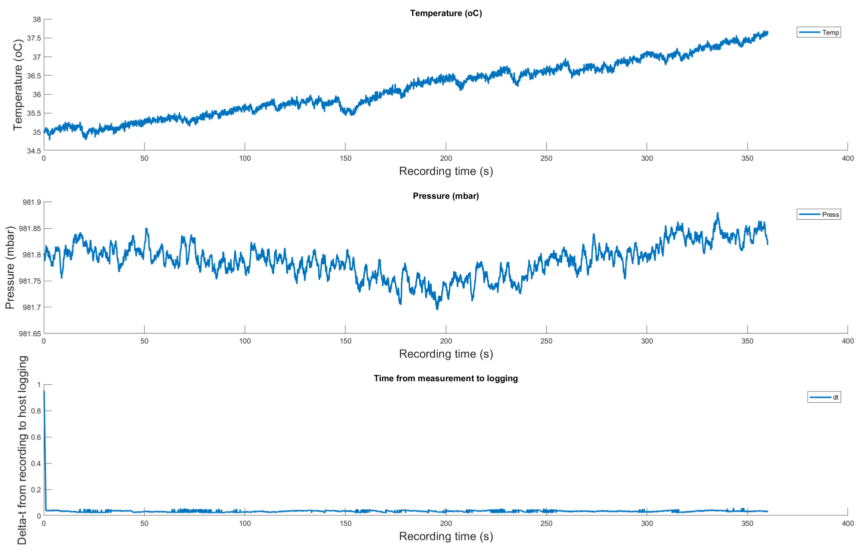


Figure 7.4: 6-minute temp, pressure, and δt measurements

7.2 COSMoS Satellite Simulation HWIL Results

Unfortunately, the HWIL interface was disabled by a bug for the 3-minute, 6-minute, and 4-hour tests, but it was collected during the 1-hour test. As noted in Chapter 6, IRIDIUM-16 is the second agent instead of IRIDUM-8 as it was simulated on the

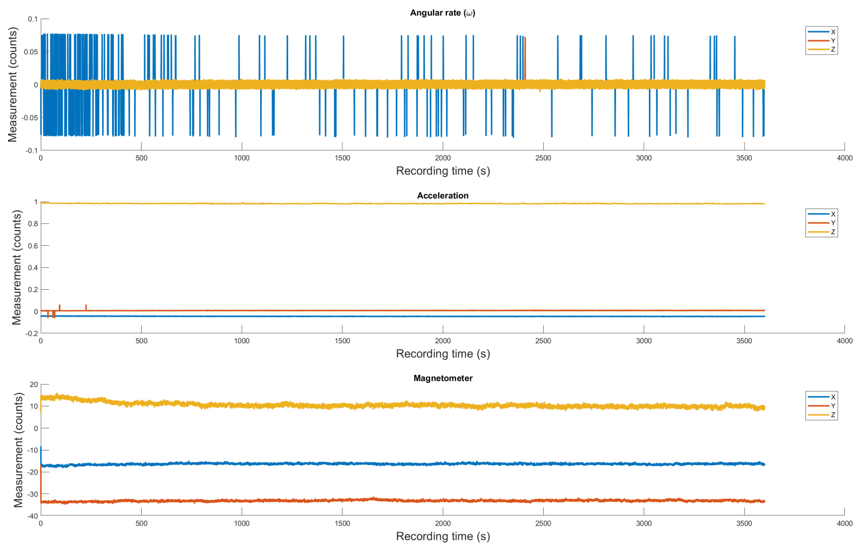


Figure 7.5: 1-hour gyro, accel, and magnetometer measurements

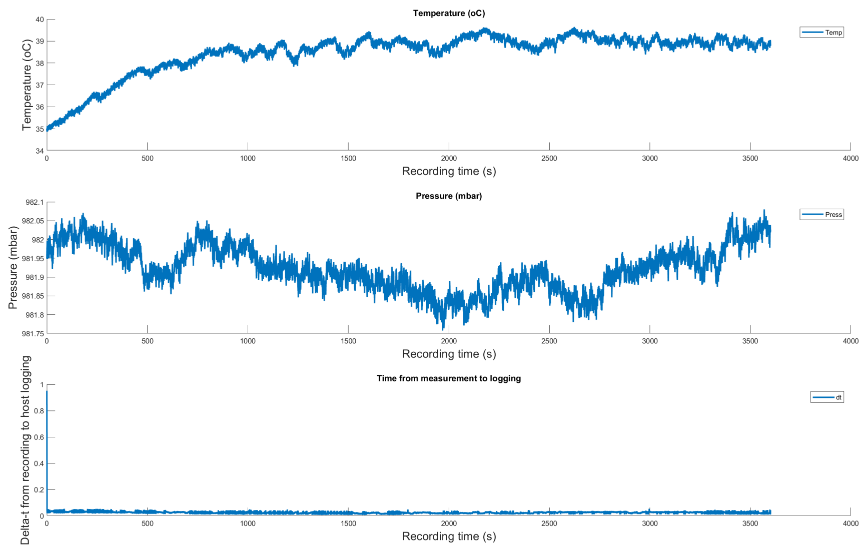


Figure 7.6: 1-hour temp, pressure, and δt measurements

HWIL enabled agent 33.

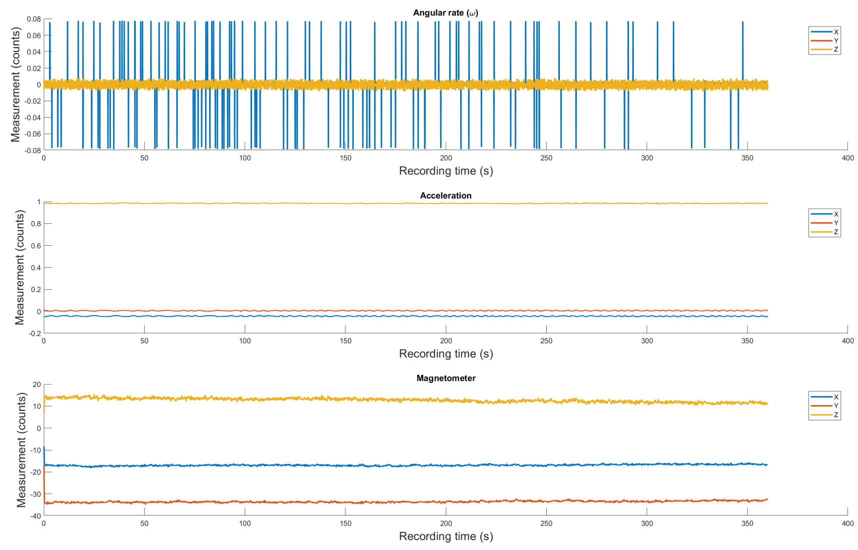


Figure 7.7: 8-hour gyro, accel, and magnetometer measurements

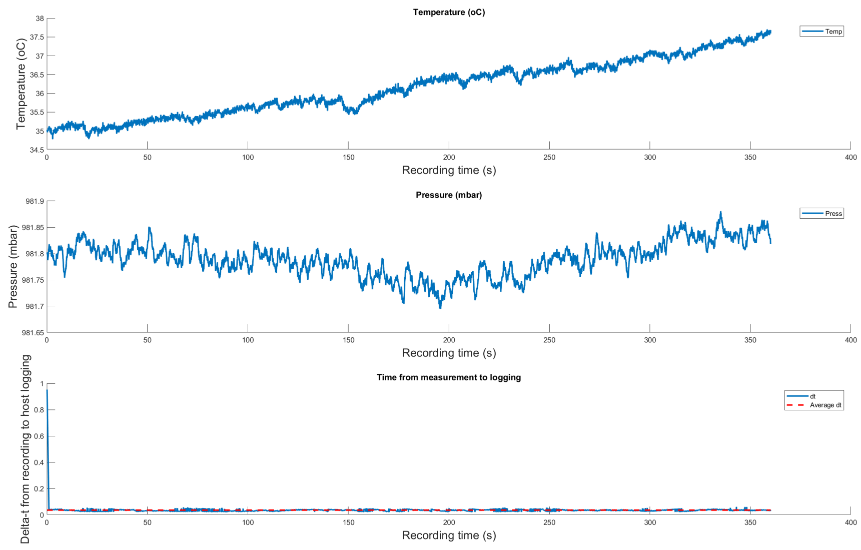


Figure 7.8: 8-hour temp, pressure, and δt measurements

7.2.1 Guidance integration HWIL measurements

The HWIL interface was also enabled during the guidance test described in Section 6.2.1. The SSA results have been included here for easy comparison with the Sense-HAT temperature measurements.

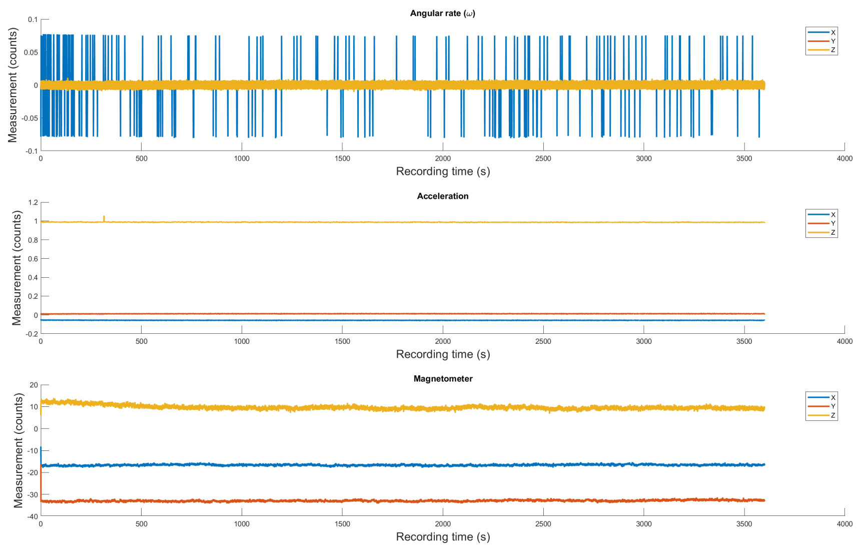


Figure 7.9: 1-hour gyro, accel, and magnetometer measurements

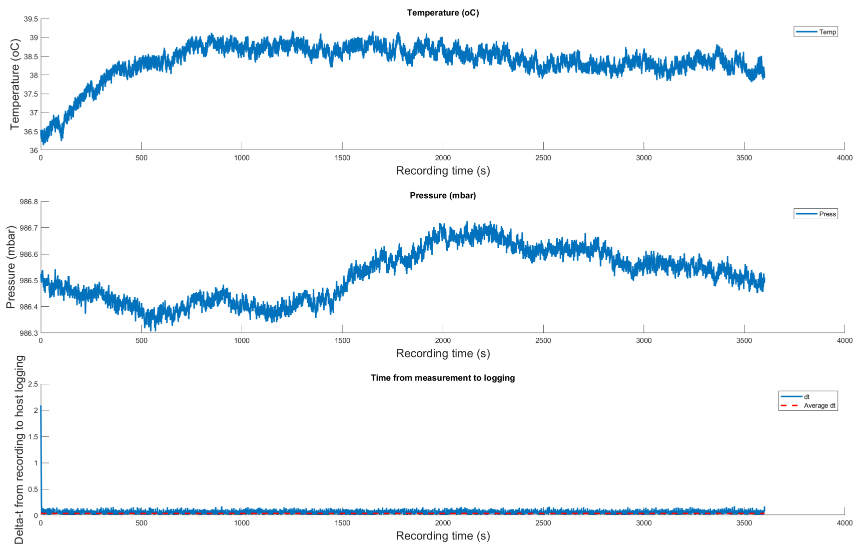


Figure 7.10: 1-hour temp, pressure, and δt measurements

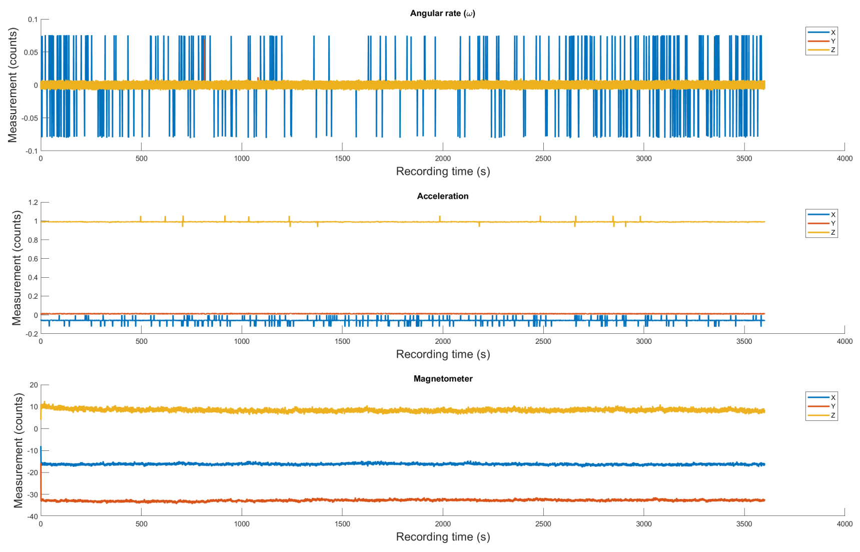


Figure 7.11: 1-hour gyro, accel, and magnetometer measurements

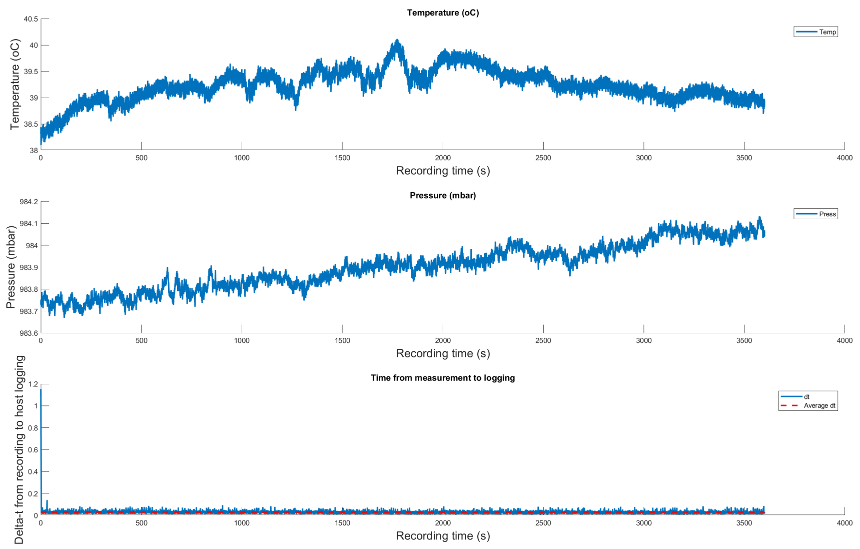


Figure 7.12: 1-hour temp, pressure, and δt measurements

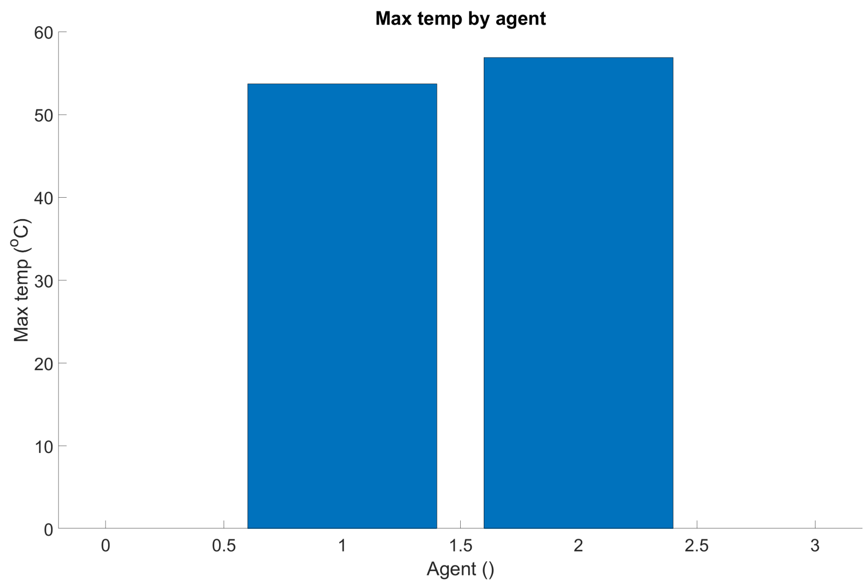


Figure 7.13: 1-hour COSMoS test maximum temperature by agent

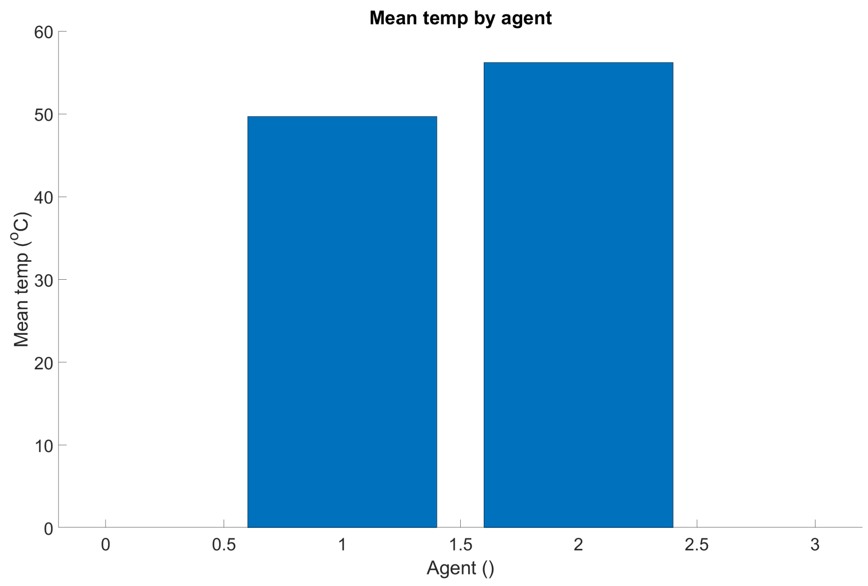


Figure 7.14: 1-hour COSMoS test mean temperature by agent

CHAPTER 8
SUMMARY AND FUTURE WORK

8.1 Summary of Completed Work

Based on the results given in Chapter 6 and the scalability analysis provided in Section 2.5, the MADNS framework and COSMoS simulation point the way towards massive-scale RT-HWIL simulation for multi-agent distributed networks using only commercial-off-the-shelf (COTS) hardware. In particular, the MADNS framework shows how it is possible to build a system which can simulate multiple agents, hosted on SBCs, in an RT-HWIL environment, and without requiring that the framework be tailored to a particular simulation application. The theoretical limits of the current MADNS/COSMoS system, as discussed in Section 2.5.4, could manage up to 1000 agents with minimal modifications. That said, there are a number of technical challenges which have restricted the system's performance to below what is theoretically possible. These challenges will be briefly discussed here, but are listed in much greater detail in Appendix B. Additionally, there are number of functions shown in the MADNS and COSMoS functional diagrams (Figs. 3.2, and 4.1) which are very close to implementation with the current architecture.

8.2 Unexpected Limitations

The creation of MADNS and COSMOS was always an ambitious project, but the difficulty involved was unclear in two major points. The largest problem that arose is the relative weakness of Python's multiprocessing libraries. Secondly, the incorporation of different propagators into the framework was more difficult than expected because the Basilisk being written in Python 2.7 while the MADNS framework is in Python

3. A third, minor, issue is due to processing delays introduced by the Sense-Hat, but it is more an outgrowth of the multiprocessing issue than a separate problem. Each of these is discussed in the following subsections.

8.2.1 Python Multiprocessing Limitations

What follows is a brief discussion of some of the issues associated with creating MADNS in Python. A more thorough discussion can be found in Appendix section B.1. Python multiprocessing has several well-characterized limitations, which the *Multiprocessing* library is designed to avoid this problem. However, while *Multiprocessing* makes processes more parallel than *Threading*, it does not make them multi-core parallel[36]. Another issue is that *Multiprocessing.Process* processes do not always exit cleanly. In many cases, these persistent and un-killable processes caused the MADNS system to hang and be unable to log all its data. Furthermore, the Python *Multiprocessing.Queue.qsize()* and *Multiprocessing.Queue.empty()* functions are not precise. This is a known issue in Python as the official documentation states that the *qsize()* method will only return an approximation of the number of data items in the queue. In multiple early attempts to run MADNS, at the conclusion of a run the system would simultaneously report *loggingQueue.empty() = True* and *loggingQueue.qsize() = 19342*. There is a high probability that this discrepancy is what caused the *Multiprocessing.Process* processes to fail to exit.

Finally, the *Multiprocessing.Queue* structures do not fail gracefully. Early benchmark tests found that when the system was set to run at 20 Hz each message had a round-trip time of 1.1 seconds. The problem was tracked to the host computer, where the *Multiprocessing.Queue* objects were failing without throwing exceptions. In low-usage cases, this problem can be rectified by setting a lower timeout on the *Queue.get()* commands, but that caused serious problems with heavy usage. In two cases, during particularly heavy usage, the host computer hung completely and required manual rebooting by disconnecting the power. Switching from parallel processing to serial reduced the time delay to 50ms on each round trip as would be expected with the 20 Hz processing rate. This is what forced the number of processors (N_p in the scalability discussion) to always be 1, and what placed a limit on the asynchronous communication inequality (Eqn. 2.1) for the current implementation.

8.2.2 Propagator Integration

As mentioned in Section 4.1.1, the original plan was to use CU Boulder's Basilisk astrodynamics toolkit. Basilisk is a powerful toolkit capable of propagating satellites with a speed-up of at least 365-to-1 (year-in-a-day). It also includes modules for atmospheric drag, solar radiation pressure, actuator models, non-rigid bodies, and the integration of guidance and control algorithms. However, Basilisk is written in Python 2 and MADNS is in Python 3. An attempt was made to use the *execnet* library to bridge this gap, but *execnet* was not capable of sustaining astrodynamic propagation at a rate of 10 Hz. Instead a simple 2-body Keplerian propagator developed by the GT-SORT group was inserted in place of Basilisk. While this did not provide a robust physics simulation for COSMoS, it did provide an opportunity to test the combined MADNS and COSMoS system.

8.2.3 HWIL Processing Delays

As noted in Chapters 6 and 7, the Sense-Hat imposed significant processing delays on agent 22 in the benchmark tests. A Sense-HAT query can take anywhere from 0.005 to 0.5 seconds before it returns its measurements. If not for the multiprocessing issues discussed in Section 8.2.1, this would not have been a problem. Unfortunately, it resulted in a 10 ms delay with a wide standard deviation on the HWIL-enabled agent as seen in Chapters 6 and 7. Should MADNS be rewritten in a language with more robust multiprocessing capabilities, this issue should be easy to correct.

8.3 Near-Term Additional Implementations

There are two major elements of MADNS and COSMoS which are not implemented in the current system, but which should require minimal effort to implement without any changes to the system. These items are both critical to the execution of a full-scale satellite constellation simulation and multi-agent network simulations in general.

8.3.1 Agent-to-Agent Communication

A critical concept discussed in Chapters 1 and 2 is the ability to simulate satellite-to-satellite (S2S) communications. While this was not implemented in the current version of MADNS and COSMoS, the path forward is clear and should be achievable with minimal work. The MPI design allows for the easy segregation of any packets labeled "agentComms" or something similar. Once selected out by *simFunc.host_simulation_function*, a time-of-flight algorithm can be implemented to determine at what point the message should arrive at its target based on the time it originated at its sender. This packet could then be routed to an agent with a specified delivery time.

Once at the agent, this packet would be routed to and stored in the *AgentSatelliteObject* instantiated on the agent. From there, a method such as *AgentSatelliteObject.deliver_message* can compare the agent's current *simTime* to the delivery time in the communication data and deliver it to the appropriate process at the appropriate *simTime*. This relatively simple implementation would allow for the simulation of satellite-to-satellite and satellite-to-ground communications without substantial reworks of either the MADNS or COSMoS frameworks.

8.3.2 Communication with an Operations Center and Ground Station

While this seems like a substantial challenge, the current construction of MADNS and COSMoS makes the task relatively trivial. The Georgia Tech Operations Center currently uses Python's *JSON* library to encode its packets for transmission across the Internet and GT network. Although MADNS currently uses *Pickle*, the transition to *JSON* should be relatively painless. As the COSMoS *simFunc.host_simulation_function* already cycles through every *HostSatelliteOjbect* on every iteration, it would be trivial to pick out the current propagated states, bundle them, and transmit the consolidated packet to the Ops Center. This was not implemented in the current version due to the long list of more pressing issues, but is an important next step in the development of MADNS and COSMoS. While this is posed as providing communications with a satellite operations center, it can be generalized to provide communications with any external system which can be adjudicated by the host computer to provide real-time communications as in agent-to-agent communications.

8.4 Conclusions

While MADNS and COSMoS cannot currently serve as a large- or massive-scale satellite constellation simulator, is has proven out several key technologies for the development of such a system. The system, as it stands, has shown the ability to handle real-time HWIL communications between multiple agents and a simulation host while simultaneously showing that the inequalities presented in Section 2.5 are reasonable to first order. Additionally, the structure of this system has shown that asynchronous host state propagation can provide a viable method for maintaining real-time operations on agents without requiring that the simulation truth be generated in real-time. This makes the creation of a massive-scale RT-HWIL system more feasible through the management of update times, synchronization rates, agent data buffers, and by carefully calibrating the size and duration of necessary synchronous state updates.

Now that companies are receiving official approval to deploy massive satellite constellations[37], and other companies are doing it without asking permission[38], it is even more important that there be an independently verifiable system for testing autonomous constellation operations. Companies will always find ways to pass regulatory test regardless of whether or not their products actually meet specifications[39]. However, an independent, open-source testing model with easy integration of independent flight codes and separately created simulation environments provides both constellation managers and regulators with a framework that can test hardware and software in a verifiable and transparent simulation environment. This project is an initial step towards creating a generalized framework for massive scale satellite constellations and other multi-agent distributed networks, such as self-driving cars and drone swarms, as these systems make their way into general use.

Appendices

APPENDIX A

MADNS AND COSMOS CODE DOCUMETATION

A.1 MADNS Code Base Details

The MADNS code base is written in Python 3 and is contained in four main files designed to modularly incorporate a variety of simulations. An associated set of bash scripts allows for the easy execution of the system, consolidation of log files after a simulation, and assist in a variety of debugging tasks.

All of these scripts are described in general below.

A.1.1 Bash Scripts

The Bash scripts are generally quite simple but are extremely useful in the use of the MADNS system.

piTargets.sh

This script is simply a listing of which SBCs will be used during a particular simulation. It should match the variable *agentList* found in the *localConfig.py* script discussed in Section A.1.2. Every other script that interfaces with the SBCs relies on *piTargets.sh*, so it is important to be sure the variable is set correctly here.

runMainExecutable.sh

This program executes several commands in preparation for running a simulation on MADNS. The order of operations is:

1. The script reboots all SBCs to clear any potential network issues and waits 25 seconds for them to come on-line. It then pings all of them to be sure they are alive.
2. The system executes *systemSetup.py* to generate a unique run number for the simulation.
3. The system executes *pushCodeToPis.sh* which syncs all SBCs with a current copy of the MADNS and simulation codebases.
4. The script executes the *hostMainExecutable.py* script on the host and in parallel opens an ssh connection to each SBC and executes *agentMainExecutable.py*
5. When all MADNS scripts have exited, the system copies all log files from the agents to the host computer and zips them for transfer to other systems.

consolidateLogs.sh

This script is the last step executed by *runMainExecutable.sh* and is responsible for collecting all log files and zipping them for easy transfer.

killCluster.sh

This script kills all Python instances operating on the host computer and reboots the SBCs. The system still does not exit gracefully from all error conditions and this script is useful in quickly ending runs which have gone awry.

manualRun.sh

This script is a variation on *runMainExecutable.sh* which performs the first three steps, but then only executes *hostMainExecutable.py* on the host computer. The assumed intent is for the user to manually execute *agentMainExecutable.py* on one or more agents to assist in debugging. Attempting to use this script with more than a handful of agents is extremely impractical.

rebootCluster.sh

A simple script that connects to each SBC via ssh and executes `sudo reboot`.

pushCodeToPis.sh

This script uses a forked rsync process to copy all MADNS and simulation code to the SBCs. It is always executed by *runMainExecutable.sh* and *manualRun.sh* before executing a simulation.

pingPis.sh

A simple script that pings all SBCs listed in *piTargets.sh* to confirm that they are active.

wipeLogFiles.sh

This script erases all log files on each SBC to avoid filling up the 32 GB SD card storage devices.

installPiSoftware.sh

This script allows for the automated installation of software on every SBC listed in *piTargets.sh* using the *apt-get* command.

uninstallPiSoftware.sh

This script performs the inverse function of *installPiSoftware.sh*.

piUpdate.sh

Similar to *installPiSoftware.sh*, this script runs `sudo apt-get update` on every SBC in parallel.

python2PackageInstall.sh and python3PackageInstall.sh

These scripts allow for the automated installation of Python packages using *pip2* and *pip3* respectively.

A.1.2 Python Scripts

The core of MADNS is a set of five main Python files supported by several data storage files and a few minor scripts. The five core functions are described below along with the more important subsidiary files. A full description can be found in Appendix A.

localConfig.py

This script is similar to the *piTargets.sh* file associated with the bash scripts described above. The script tunable parameters essential to the operation of MADNS and also variables related to the simulation being run on MADNS. The most important variables are described in Table A.1.2. Many of these variables are contained inside a Python dictionary called *simConfigData*. All such variables are listed inside quotation marks in the table (e.g. 'numAgents')

There are many more settings described in the file itself, but these are a representative sample of the tunable parameters in MADNS.

Table A.1: MADNS Configuration Variables

Variable Name	Description
VERBOSITY	This variable takes a value from 0-5 and sets how verbose the system will be during execution.
simRunTime	This variable is used in several other places in this file and sets how many seconds the simulation will run for in simulation time.
numWorkers	This variable was used to set the number of parallel pool workers, but is currently deprecated.
'numAgents'	This variable sets the number of SBCs to be used during a run. It appears redundant but can be useful in checking for system configuration errors.
'agentList'	This list explicitly sets the ID numbers for the SBCs being used in a simulation. An arbitrary configuration of the available SBCs can be used as desired.
simulationPath	This variable sets the home directory for the simulation files (such as COSMOS)
hostSimulationFile	The system allows for separate host and agent simulation files, although COSMOS currently only uses one.
'HWILagents'	This variable allows the designation of which agents have HWIL interfaces to avoid executing unnecessary code on agents not so equipped.

A.1.3 Real-Time Execution Rate Limiter

The most critical library in the MADNS system is *rateLimiterLib.py* (RL). This library is very simple as its primary function is:

```
def limit_rate(rate, start):
        sleepTime = 1./ rate - (time.time() - start)
        if sleepTime > 0:
                time.sleep(sleepTime)
        return sleepTime
```

This function allows any process in MADNS to be limited to an execution rate specified in *localConfig.py.* This is critical to real-time operations, but also protects the system from polling by the multiprocessing network communication functions. Early tests had the SBCs all running at their maximum operating temperature and with one core operating at 100% usage at all times. The culprit was the listening server described below. Without any *time.sleep()* commands, the while loops in the servers repeated at the processor rate and caused substantial overheating on the SBCs. While this could be resolved with a simple *time.sleep(0.001)* command, the *rateLimiterLib* aloows the enforcing of processing rates for real-time operations. Python's *time.sleep()* command usually accurate to better than 10ms, which has been sufficient for the current applications but may not be for future use cases. This issue is discussed further in Chapter 8.

Other functions found in the RL library are used to document real-time execution violations using the logging utilities described below.

Main Executables

The core MADNS functions are *hostMainExecutable.py* and *agentMainExecutable.py.* The fundamental function of these files is to run a distributed state machine which passes through a series and runs the simulation as shown in Table A.2.

The bulk of the simulation work is done during the SIMULATION_START and RUN_SIMULATION phases on the host and agents respectively. The general outlines of these functions are discussed below.

hostMainExecutable.py

The *hostMainExecutable* relies on the *networkCommLibrary*, *systemStatusAgent*, and *loggingUtilities* functions. It also allows for the import of an arbitrary simulation function using the syntax:

Table A.2: MADNS State Machine

Host State	Agent State
HOST_STARTUP	AGENT_READY
SIMULATION_SETUP	SIMULATION_SETUP
HOST_SIMULATION_READY_STARTUP	
SIMULATION_ START	RUN_SIMULATION
	SIMULATION_COMPLETE
SIMULATION_COMPLETE	
BEGIN_SHUTDOWN	AGENT_SHUTDOWN
CLEANUP_FUNCTIONS	END_AGENT
END_HOST	WAITING_SERVER_STOP
WAITING_SERVER_STOP	
exit	exit

```
    sys.path.insert(0, localConfig.simulationPath)

    simFunc = importlib.import_module(localConfig.hostSimulationFile)
```

This allows the host simulation file to be set in the *localConfig.py* settings. The simulation file needs to follow an interface described later and in Section 4.1.1, but beyond that any code can be placed within the simulation framework.

Before the state machine begins, the host main executable spawns a listening server, a transmit server, and a logging process. The listening server remains open until after the agents have all reported complete and communicates through the *Multiprocessing.Queue* object instantiated as *incomingQueue*. It remains running until the host places data in a the *serverStopQueue* Queue object. These three processes all run at rates specified in *localConfig.py* to avoid consuming excessive resources on the host computer.

If the host fails to establish these three processes, the state machine enters with the state "EMERGENCY_SHUTDOWN" which immediately closes the host and broadcasts "EMERGENCY_SHUTDOWN" to the agents so they also close. There are several other points in the state machine where "EMERGENCY_SHUTDOWN" can be set, including by the agents. If an agent enters an "EMERGENCY_SHUTDOWN" state, it broadcasts that to the host which rebroadcasts to all agents and begins its own shutdown process.

If the three processes have been established, the host enters its state machine with the state "HOST_STARTUP". During the startup phase, the host uses the *networkCommLibrary.py* (hereafter referred to as *NCL*) *query_and_wait* function which allows the host to broadcast a message to all agents and wait for a reply. In this case, the host broadcasts "SYSTEM_START" and waits for every agent to reply with "AGENT_READY", after which the host changes its state to "SIMULATION_SETUP".

During "SIMULATION_SETUP", the host executes the simulation function *simFunc.setup_host_simulation*, which is one of the simulation functions required by the API. Regardless of how it operates, *setup_host_simulation* must return the next desired host state, a list of classes associated with the simulation (*simClasses*), a list of all Queue objects required by the simulation, lists of all log files and data files required by the simulation, and a list containing the initial state data required by the simulation. If the function runs correctly it returns the state value "HOST_SIMULATION_READY" and the host progresses.

During "HOST_SIMULATION_READY" the host system executes *simFunc.send_host_setup_to_agents*, another required simulation function. This function transmits the initial simulation data to the agents and waits for the agents to report "AGENT_SIMULATION_READY". This behavior is a required element in the MADNS API. If all agents report ready, the function returns "SIMULATION_START" or it returns "EMERGENCY_SHUTDOWN" if something has gone wrong such as an agent not reporting.

During "SIMULATION_START" the host executes *simFunc.run_host_simulation* which is where the true business of MADNS occurs. This function call is wrapped in try/except with special provisions made for a *KeyboardInterrupt* exception. This allows the system to exit relatively cleanly if the operator needs to end the process for any reason. This function blocks the execution of the main executable until either the simulation is either complete or interrupted and returns the next *hostState* and any processing pool functions created by the simulation. Currently, the *statePool* variable does nothing due to the issues discussed in Section 8.2.1, but should that be resolved it allows the main executable to ensure that the whole process exits cleanly once the simulation is complete. A required API element is that *run_host_simulation* return "SIMULATION_COMPLETE" once all agents have reported "SIMULATION_COMPLETE".

Currently, the "SIMULATION_COMPLETE" phase executes a second *query_and_wait* to be sure that all the agents have reached the "AGENT_STOP" phase discussed later. The host then moves on to the "CLEANUP_FUNCTIONS" phase, during which the host attempts to empty all its *Multiprocessing.Queue* objects and log any orphaned packets for later analysis. There is space here for analytic functions to determine how the simulation performed, but none have been implemented so far. Next, the host enters "END_HOST" where the host closes its listening server, transmit server, and closes the logging process once the *loggingQueue* Queue object is empty. This was the source of considerable trouble during development as the Python *Multiprocessing.Queue* objects have some substantial deficiencies described in Section 8.2.1.

Finally, the host enters "HOST_CLOSEOUT" where it kills any persistent processes using the *Multiprocessing.Process.terminate()* function to send *SIGTERM* to any function which did not exit normally. Again, this is an outgrowth of the issues described in Section 8.2.1. Once all processes have terminated, the host executable ends and *runMainExecutable.sh* continues to collect data from all the agents, concluding host operations for the simulation.

agentMainExecutable.py

The agent executable functions similarly to the host executable. When started, it attempts to spawn the same listening, transmit, and logging processes found on the host. Once those processes are alive, it enters its state machine with similar steps to the host. Only the important distinctions will be addressed here.

The first notable difference from the host executable is the establishment of agent specific variables in the *simConfigData* dictionary such as *"localAgentNum"*, which are used to identify the agent during communications with the host. The next is the establishment of a system status agent (SSA) using the *systemStatusAgent.py* library discussed later. While such an agent could be established on the host, it has so far only been required on the agents.

The agent then enters "AGENT_READY" where it waits for the host to transmit "SYSTEM_START" and returns "AGENT_READY" to the host. After receiving "SYSTEM_START", the agent moves into the "SYSTEM_START" state and waits for a "SIMULATION_SETUP" packet generated by *simFunc.send_host_setup_to_agents* on the host. Upon receiving the setup packet, the agent passes the data to *simFunc.setup_agent_simulation*. When the simulation is prepared, the agent triggers its system status agent and moves to "AGENT_SIMULATION_READY", which it transmits to the host.

The agent then waits for the host to broadcast "SIMULATION_START" and moves into "RUN_SIMULATION" where, like the host, it executes *simFunc.run_agent_simulation* which blocks until the agent has completed its simulation. Upon exit from *run_agent_simulation*, the agent transmits "SIMULATION_COMPLETE", which the host is waiting for from all agents before it moves into "END_SIMULATION". The agents then wait for the host to broadcast "SYSTEM_STOP" before entering "AGENT_SHUTDOWN" where the agent performs its cleanup functions. These functions currently include recording any orphaned packets and killing the logging and server processes as on the host. Once completed, the agent transmits "AGENT_CLOSED" to the host without using the now dead *outgoingServer* process, and the agent process is complete.

Network Communication Library

All network communication tools are stored in *networkCommLibray.py*, referred to as *NCL*. The NCL contains the following functions:

- *format_outgoing_message*
- *query_and_wait*
- *outgoing_server*
- *udp_listen*
- *broadcast*

- *udp_send*

Most of these functions are as self explanatory as they appear, but several require some comment.

NCL.query_and_wait

This function allows the host to request information from all agents at once and wait for their responses. The function takes in the arguments *broadcastMessage*, *returnMessage*, and *nextState*. *NCL.query_and_wait* will use *NCL.broadcast* to broadcast the *broadcastMessage* to all agents, and then wait for a packet carrying the *returnMessage* to arrive from every agent. If any agents fail to respond within 1 second, the function will rebroadcast the *broadcastMessage* to any agents which have failed to report several more times. If all the agents report, the function returns *nextState* to progress the host's state machine. Otherwise, the function returns "EMERGENCY_SHUTDOWN" to prompt exit behavior due to a missing agent.

NCL.outgoing_server and NCL.udp_send

These two functions provide the service one might expect, but they do so within the MADNS framework. Both are limited by the *'serverRate'* set in *localConfig.py* to avoid excessive resource consumption and both communicate with the host through *Multiprocessing.Queue* objects. The outgoing server runs at its specified rate and continually checks the *outgoingQueue* object for any messages which need to be sent. This helps guarantee that messages will be sent in a timely manner regardless of whatever else the host or agent is occupied with. The *udp_litsen* function is also designed as a separate process and, when it receives a message, inserts it into the *incomingQueue* object for processing by a main executable or a simulation function. As these are both spawned processes using Python's *Multiprocessing.Process* protocol, they need to receive orders to stop from the main executable. This is handled using a *Multiprocessing.Queue* object called *serverStopQueue*. Both UDP are wrapped in a `while serverStopQueue.empty()` loop and complete their operations and exit only when the main executable has placed something into the *serverStopQueue*. This has led to some issues noted in Section 8.2.1, but in general the functions behave well or are terminated using a *SIGTERM* during host and agent cleanup phases. Both functions execute *serverStopQueue.get()* to log their reason for stopping, so it is important that the main executable place two objects in the queue to stop both processes.

All MADNS packets are *Pickle*ed Python lists and have the same form:

[Packet type, Target IP address, [Data packet]]

Because of the use of *Pickle*, the data packet can be any arbitrary Python data structure, but there are some limitations. There is code in this function to attempt the collection of large packets, but it has never worked. Instead, the socket receive buffer has been increased to 2^{17} bytes. This is a sub-optimal solution, but it was necessary to make COSMoS work in the short run.

Every packet must contain the target IP due to the use of the UDP protocol. Early tests showed that every agent received a copy of every UDP packet during transmission and acted on them. *NCL.udp_listen* filters packets by comparing the target IP to the agent's (or host's) IP address stored in *simConfigData* and passed to the outgoing server process when it is spawned.

Logging Utilities

The utilities stored in *loggingUtilities.py*, referred to as LU, are responsible for storing all data generated during the simulation. Early tests showed large delays in the ability of agents to handle the benchmark test, and this problem was traced to disk I/O on the SBC's SD cards. By creating a separate logging utility using the *loggingQueue* Queue object, this load was moved away from the main process and the result was that the benchmark test began to operate at its expected rate as described in Chapter 6.

LU.logging_process

Most functions in the LU library simply create log files and log directories, but *LU.logging_process* is the heart of the system. Like *NCL.udp_listen* and *NCL.outgoing_server*, this function is designed to operate as a separate process spawned using *Multiprocessing.Process*. The process starts when instructed, loops at a rate specified in *localConfig* and acts when it finds data in the *loggingQueue* Queue object. The process ends only when *loggingQueue.empty()* reports true and the main executable has placed something into the *loggingStopQueue*.

LU.log_print

This is the primary debugging tool for MADNS. The *LU.log_print* function is what implements the *VERBOSITY* value set in *localConfig*. While many of the *log_print* calls are currently in case they damaged performance, in general any message being sent to the *loggingQueue* is also passed to *log_print*. This function compares the verbosity level attached to a particular log packet with the *VERBOSITY* level set at startup. If the verbosity of the packet is less than the global *VERBOSITY*, the message is printed to the screen as well as being written to disk. Because MADNS creates a lot of data at very high rates, this setting must be used with great care.

A.1.4 System Status Agent

The library *systemStatusAgent.py* was one of the first scripts created for MADNS, and the code shows its age in many ways. The primary function of the agent (SSA) is to gather data about a computer's operation and log it for future reference during a simulation. In general, the utility collects data on core temperature, CPU usage, memory usage, disk usage, and network I/O.The utility is capable of collecting network traffic information, but this was never implemented.

An important note is that this utility only works on the agents because it requires the Python *psutil* library. Unfortunately, *psutil* requires elevated permissions to install because it interacts directly with the operating system, and Georgia Tech IT department did not want to install it on the host computer. However, as cluster operators have root access to the SBCs, it was trivial to install there.

A.1.5 Minor Utilities

The *runManagementLib.py* file has a much more grandiose name than its function deserves. This script simply creates and reads the *currentRun.py* and *runList.txt* files, which designate a unique run number for every simulation. These run numbers are used by the main executables and *runMainExecutable.sh* to organize the log files generated during a simulation.

There are a number of other utilities found in the MADNS Git repository which are not document here, such as *localMathLib.py*, because they were never implemented.

A.2 MADNS Files and function calls

The following documents all MADNS function calls and their return values, organized by file with commentary where appropriate.

A.2.1 hostMainExecutable.py

The following code occurs during the imports:

```
# Hack? to use all cores

os.system("taskset -p 0xff %d" % os.getpid())



# Import simulation module based on local config setting

sys.path.insert(0, localConfig.simulationPath)

simFunc = importlib.import_module(localConfig.hostSimulationFile)
```

The first command sets the processor affinity for all spawned processes to any available processor so the OS assigns them freely. The second inserts the simulation file directory and loads the host simulation file as textitsimFunc. Both of these settings are managed in *localConfig.py*.

```
def spawn_listening_server(incomingQueue, hostIP, listenPort, serverLogFile
    ↪ , serverStopQueue, loggingQueue, serverRate, rateLogFile,
    ↪ rateDataFile):
    return mp.Process(target=NCL.udp_listen, args=(incomingQueue, hostIP,
        ↪ listenPort, serverLogFile, serverStopQueue, loggingQueue,
        ↪ serverRate, rateLogFile, rateDataFile))


def spawn_transmit_server(outgoingQueue, ipPrefix, targetPort,
    ↪ serverLogFile, serverStopQueue, loggingQueue, serverRate,
    ↪ rateLogFile, rateDataFile):
    return mp.Process(target=NCL.outgoing_server, args=(outgoingQueue,
        ↪ ipPrefix, targetPort, serverLogFile, serverStopQueue,
        ↪ loggingQueue, serverRate, rateLogFile, rateDataFile))


def spawn_logging_process(loggingQueue, loggingStopQueue, loggingRate,
    ↪ systemLogFile, HOST_START_TO):
    return mp.Process(target=LU.logging_process, args=(loggingQueue,
        ↪ loggingStopQueue, loggingRate, systemLogFile, HOST_START_TO))


def check_for_stupid(simConfigData):
    if simConfigData['hostPoolWorkers'] == 0:
        LU.log_print('hostMainExecutable: check_for_stupid: ERROR:
            ↪ hostPoolWorkers == 0', 0)
    if not simConfigData['numAgents'] == len(simConfigData['agentList']):
        LU.log_print('hostMainExecutable: check_for_stupid: ERROR: numAgents
            ↪  != len(agentList)', 0)
    else:
        return 0
```

```
def main():
        return 0
```

A.2.2 agentMainExecutable

```
def spawn_listening_server(incomingQueue, hostIP, listenPort, serverLogFile
    ↪ , serverStopQueue, loggingQueue, serverRate, rateLogFile,
    ↪ rateDataFile):
    return mp.Process(target=NCL.udp_listen, args=(incomingQueue, hostIP,
        ↪ listenPort, serverLogFile, serverStopQueue, loggingQueue,
        ↪ serverRate, rateLogFile, rateDataFile))


def spawn_transmit_server(outgoingQueue, ipPrefix, targetPort,
    ↪ serverLogFile, serverStopQueue, loggingQueue, serverRate,
    ↪ rateLogFile, rateDataFile):
    return mp.Process(target=NCL.outgoing_server, args=(outgoingQueue,
        ↪ ipPrefix, targetPort, serverLogFile, serverStopQueue,
        ↪ loggingQueue, serverRate, rateLogFile, rateDataFile))


def spawn_logging_process(loggingQueue, loggingStopQueue, loggingRate,
    ↪ systemLogFile, HOST_START_TO):
    return mp.Process(target=LU.logging_process, args=(loggingQueue,
        ↪ loggingStopQueue, loggingRate, systemLogFile, HOST_START_TO))


def spawn_ssa_process(simConfigData, hostIP, outgoingQueue, loggingQueue,
    ↪ SSAstopQueue, systemStatusLogFile, rateLogFile, rateDataFile,
    ↪ agentFlag):
    return mp.Process(target=SSA.main, args=(simConfigData, hostIP,
        ↪ outgoingQueue, loggingQueue, SSAstopQueue, systemStatusLogFile,
        ↪ rateLogFile, rateDataFile, agentFlag))


def check_for_stupid(simConfigData):
    if simConfigData['hostPoolWorkers'] == 0:
        LU.log_print('hostMainExecutable: check_for_stupid: ERROR:
            ↪ hostPoolWorkers == 0', 0)
    if not simConfigData['numAgents'] == len(simConfigData['agentList']):
        LU.log_print('hostMainExecutable: check_for_stupid: ERROR: numAgents
            ↪  != len(agentList)', 0)
    else:
        return 0


def main():
    if agentState == "AGENT_SHUTDOWN":
        return 0
        else:
                logMsg = "agentMainExecutable: Reached end of executable
                    ↪ without agentState == 'WAITING_SERVER_STOP'"
                print(logMsg, file=sys.stderr)
                LU.write_log(systemLogFile, logMsg, AGENT_START_TO)

                logMsg = "agentMainExecutable: Exiting agent executable
                    ↪ abnormally"
                print(logMsg, sys.stderr)
                LU.write_log(systemLogFile, logMsg, AGENT_START_TO)

                return 1
```

A.2.3 loggingUtilities.py

```
def log_print(message, level):
    if level <= localConfig.VERBOSITY:
        print(message)


def get_current_utc():
    return datetime.datetime.utcnow().strftime("%Y_%m_%d_%H_%M_%S")


def write_log(fhandle, message, TIME_0=0):
    return 0


def make_log_folder(logPath):


def setup_log(logPrefix, logPath, runNumStr):
    return logFile


def data_log_output(fhandle, dataPacket):
    return 0


def logging_process(loggingQueue, loggingStopQueue, loggingRate, logFile,
    ↪ TIME_0):
    while not loggingQueue.empty() or loggingStopQueue.empty():
            <...>
    return 0
```

A.2.4 networkCommLibrary.py

```
def format_outgoing_message(messageList, targetIP):
    return msg
```

```
def query_and_wait(agentList, systemLogFile, transmitLogFile, loggingQueue,
    ↪  ipPrefix, targetPort, broadcastMessage, returnMessage, nextState,
    ↪ incomingQueue, FUNC_TO):
    return nextState


def outgoing_server(outgoingQueue, ipPrefix, targetPort, logFile,
    ↪ serverStopQueue, loggingQueue, serverRate, rateLogFile, rateDataFile
    ↪ ):
    return 0


def udp_listen(incomingQueue, localIP, listenPort, logFile, serverStopQueue
    ↪ , loggingQueue, serverRate, rateLogFile, rateDataFile):
    return 0


def broadcast(targetList, ipPrefix, targetPort, messageList, logFile,
    ↪ loggingQueue, TIME_0=0):
    return 0


def udp_send(sock, targetDevice, ipPrefix, targetPort, message, logFile,
    ↪ loggingQueue, TIME_0):
    return bytesVal
```

A.2.5 rateLimiterLib.py

```
def limit_rate(rate, start):
    sleepTime = 1. / rate - (time.time() - start)
    # LU.log_print('RLL: limit_rate: sleepTime: {}'.format(sleepTime), 5)
    if sleepTime > 0:
        time.sleep(sleepTime)
    return sleepTime


def rate_limit_log(rlTest, processName, loopName, rateLogFile, rateDataFile
    ↪ , TIME_0):
    if rlTest <= 0:
        if rlTest <= -0.005:
            logMsg = 'RLL: ERROR: {}: Rate overflow: {} loop: {}'.format(
                ↪ processName, loopName, rlTest)
            LU.log_print(logMsg, 0)
            LU.write_log(rateLogFile, logMsg, TIME_0)
            LU.write_log(rateDataFile, [processName, loopName, rlTest])
            return 1
        else:
            logMsg = 'RLL: Warning: {}: Rate overflow: {} loop: {}'.format(
                ↪ processName, loopName, rlTest)
            LU.log_print(logMsg, 6)
            LU.write_log(rateLogFile, logMsg, TIME_0)
            LU.write_log(rateDataFile, [processName, loopName, rlTest])
            return 0
    else:
        # logMsg = '{}: Rate report: {} loop: {}'.format(processName,
            ↪ loopName, rlTest)
        # LU.log_print(logMsg, 3)
        # LU.write_log(rateLogFile, logMsg)
        return 0


def report_rate_overflows(rateOverflow, processName, loopName, rateLogFile,
    ↪  TIME_0):
```

```
    logMsg = 'Total rate overflows from {}: {}: {}'.format(processName,
        ↪ loopName, rateOverflow)
    LU.write_log(rateLogFile, logMsg, TIME_0)
    LU.log_print(logMsg, 0)
    return 0
```

A.2.6 runManagementLib.py

```
def increment_run_number(runListHandle, runDateTime):
    return runNum, runStr


def new_run_file(runListHandle, runDateTime):
    return runNum


def read_run_num(runListHandle):
    return runNum, runStr
```

A.2.7 systemStatusAgent.py

```
class StateOfHealth:
    def __init__(self, temp, ram, diskSpace, diskIO, networkIO, networkConn
        ↪ , cpuUsage):
        self.temp = temp
        self.ram = ram
        self.diskSpace = diskSpace
        self.diskIO = diskIO
        self.networkIO = networkIO
        self.networkConn = networkConn
        self.cpuUsage = cpuUsage
```

```
class TimeDataPacket:
    def __init__(self, time0, currTime, prevTime, deltaTime, tick, trigger)
        ↪ :
        self.time0 = time0
        self.currTime = currTime
        self.prevTime = prevTime
        self.deltaTime = deltaTime
        self.tick = tick
        self.trigger = trigger

def get_ip_address(ifname):
        return addr

def time_update(timeData, SYSTEM_STATUS_T0):
    return TimeDataPacket(timeData.time0, currTime, prevTime, timeData.
        ↪ deltaTime, timeData.tick, timeData.trigger)

def get_cpu_temp():
    return float(strOut[strOut.index('=') + 1:strOut.rindex("'")])

def get_ram_info():
    return [ram_total, ram_used, ram_free, ram_percent_used]

def get_disk_space():
    return ([disk.total, disk.used, disk.free, disk.percent])

def get_disk_io():
    return [diskIO.read_count, diskIO.write_count, diskIO.read_bytes,
        ↪ diskIO.write_bytes, diskIO.read_time, diskIO.write_time, diskIO.
        ↪ busy_time, diskIO.read_merged_count, diskIO.write_merged_count]


def get_network_io():
    return ([net.bytes_sent, net.bytes_recv, net.packets_sent, net.
        ↪ packets_recv, net.errin, net.errout, net.dropin, net.dropout])


def get_network_conn():
    return ('NetConnInfoHereEventually')


def get_cpu_usage():
    return cpuPct


def get_general_info():
    return bootTime


def build_msg_list(timeData, soh, localIP):
    return msgList


def run_benchmark(iterCount, numThreads):
    return totTime


def main(simConfigData, localIP, outgoingQueue, loggingQueue, SSAstopQueue,
    ↪  logFile, rateLogFile, rateDataFile, runningOnAgent):
    return 0
```

Notes: In this function, the spawned process is actually *SSA.main*. The *runBenchmark* function is not directly related to the overall MADNS benchmark counting test. Instead, it allows the system to stress itself by running a prime number factorization algorithm parallel to whatever else MADNS is doing. In theory, this allows the system to be tested under high agent load although the results of these tests have not been representative of how COSMoS behaves.

A.3 COSMoS Code Base Details

This section describes the COSMoS code base and the MADNS simulation API in general. COSMoS is a particular application for MADNS, and so care will be taken to state where functions are purely a part of COSMoS and where they are a required part of MADNS.

A.3.1 hwilUtilites.py and dummyHWIL.py

The HWIL utilities will be discussed in greater detail in Chapter 5, however it should be noted that the HWIL utilites are part of the MADNS interface. The HWIL utilities are required to have the following functions:

- *setup_hwil_interface*, which returns the HWIL interface driver used by the simulation
- *hwil_query_and_send*, which uses the HWIL interface driver to query the hardware interface and then passes the data to other agent functions via a Queue object or simply transmits it to the host as in the current COSMoS use case.
- *hwil_process*, which was designed as a multiprocess function, but has suffered the same fate as may of its cousins.

The *dummyHWIL.py* script exists to be loaded by any agent without a hardware interface. It contains the same functions as *hwilInterface.py*, but they are all of the following form:

```
    def setup_hwil_interface():

        return 0
```

A.3.2 COSMOSmainFile.py - MADNS Core Functions

The MADNS framework allows for separate host and agent simulation files, but COSMoS currently only uses one file for both. As noted in the main executable discussion, this function is loaded as *simFunc* by the host and agent executables. Using a single file is not the most efficient path as it requires either installing every toolbox required by the host (such as *matplotlib*, *numpy*, and *SciPy*) on every agent or providing dummy functions as described in Section A.3.1. In this case, that was done for *SciPy*, but most of the other libraries are used by both the host and agents.

The following functions are required parts of the MADNS API. In a template MADNS simulation file, they are found below a divider which separates simulation specific functions from MADNS required functions. The hope is that users will not have to modify the MADNS required functions, but will be able to do all their work above the divider.

simFunc.setup_agent_simulation

As described earlier, this function accepts the initial simulation data provided by the host and prepares it for use on the agent simulation. It is also responsible for creating classes, queues, log files, and data files used by the particular simulation. When complete, the function must report "AGENT_SIMULATION_READY" to advance the agent's state machine. For COSMoS, the primary function is the instantiation of an *AgentSatelliteObject* from *satelliteFunctions.py* discussed below.

simFunc.setup_host_simulation

This function creates classes, queues, log files, and data files like its counterpart on the agents, but it has several additional functions. The host setup function reads in data files and parses them to create instances of host class objects and package a data packet to be sent to the agents. For COSMoS, this involves the creation of *HostSatelliteObjects* from the *satelliteFunctions.py* library.

simFunc.send_host_setup_to_agents

This function is a close relative of *NCL.query_and_wait*. It does everything that *query_and_wait* does, but it keeps track of which data elements have been sent as well as which agents have reported that they received the data and are ready. Because the initial state data lists are more complicated than a simple *NCL.broadcast* message, it was not possible to use *NCL.query_and_wait*, but it is a good guide for understanding this function.

simFunc.run_host_simulation

This function is the interface between MADNS and the simulation code being run on the host computer. First it runs the simulation specific *simFunc.start_host_simulation* to mark the host's *simT0* variable for tracking simulation time however the specific simulation wishes to do that. It performs an *NCL.query_and_wait* to be sure that all agents are running the simulation, and then enters a rate limited (using *RL.limit_rate*) while loop until every agent has completed its simulation. The loop executes a simulation specific *simFunc.host_simulation_function()* using a common MADNS interface, which will return any packets not specifically used by the simulation. These packets include, but are not limited to, SSA updates, "EMERGENCY_STOP" requests, and the most important "SIMULATION_COMPLETE" packets. Because it is expected that *host_simulation_function* will access the *incomingQueue*, the function must report any packets it finds which were not meant for the simulation, or *None* if the *incomingQueue* was empty or contained a simulation packet. When an agent reports "SIMULATION_COMPLETE", the function removes that agent's number from the *agentProcessesRunningSimulation* list. Once that list is empty, the simulation *while* loop ends, and the host state machine progresses to "SIMULATION_COMPLETE".

It should be noted that this function is also responsible for spawning the host state pool. This was the source of most problems discussed in Section 8.2.1, and has therefore been commented out. However, should it be reactivate at any point it is important to know why it is here and not in *setup_host_simulation*. When the pool functions are spawned using the, currently, deprecated *state_pool_generator* function, they are provided with *time.time()* as their *POOL_T0* value. It is important that this value be as close as possible to the actual start of the simulation so that all pool workers and the main simulation share, approximately, the same *simT0* value for later data analysis.

simFunc.run_agent_simulation

This function is almost identical to *simFunc.run_host_simulation* except that its while loop terminates when the current simulation time (*time.time() - simT0*) meets or exceeds the simulation run time set in *localConfig*. It is also responsible for invoking calls to the HWIL libraries if the agent it is running on has HWIL devices. When its while loop exits, the function is also responsible for sending "SIMULATION_COMPLETE" to the host.

A.3.3 COSMOSmainFunction.py - Simulation Specific Functions

These functions are all found above the divider in a MADNS main simulation file. While some of these functions are required by the MADNS framework, their contents are extremely simulation specific and it is expected that their internal workings will be heavily modified. It is hoped that their inputs and outputs can remain a constant interface as much as possible.

simFunc.parse_simulation_data_file

This function is responsible for parsing the data files used by *setup_host_simulation*. So long as the function accepts a file name and returns a valid Python data structure, the parser itself can be modified to match any data format.

simFunc.generate_state_packet_data

This function accepts a target agent ID and the data being sent to that agent. It simply formats the data into a MADNS compliant packet (as described in the NCL section), and then returns that packet for eventual passing to *send_host_setup_to_agents*.

simFunc.parse_state_data

Should initial state data, or simulation state data, require any processing on the host or agent side, this function can be used to provide that. For COSMoS, it reads:

```
    def parse_state_data(packet):

        return packet
```

simFunc.host_class_setup

This function takes the initial state data and creates instances of simulation classes on the host. The assumed needs of these classes, based on the experience of creating COSMoS are the general simulation configuration data, the initial state data, and all three multiprocessing queues. The function must return a list of classes which is later used to map incoming data to a particular agent's class on the host.

simFunc.host_simulation_function

This is the central function in any MADNS simulation. As mentioned before, this function is called by *simFunc.run_host_simulation* in a rate limited loop. Any activities which the simulation needs to perform to generate noumenons are placed here as are any actuator models which convert those noumenons to phenomenons for use by the agents. In many cases, all those functions will be handled by an instantiated simulation class, but those class activities are triggered here. Additionally, if the simulation accesses the *incomingQueue* it must return any unused packets to *run_host_simulation* or return *None* if the packet was used by the simulation and not needed by the MADNS framework.

For COSMoS, this function loops through all the satellite objects stored in *simClassess*, and triggers their propagation methods, but avoids excessive propagation as described in the discussion of *satelliteFunctions.py* below.

simFunc.agent_simulation_function

Similar to *host_simulation_function*, this is responsible for running the simulation on the agent. For COSMoS, its current purpose is to add incoming host propagation data to the agent's data buffer through the *AgentSatelliteObject* methods, run the GNC algorithm attached to the satellite object, transmit the agent's current simulation time back to the host to coordinate the simulation, and store the agent's current state for future data processing.

simFunc.start_host_simulation and simFunc.start_agent_simulation

These two functions simply mark the host's and agent's *simT0* at an appropriate time just before beginning simulation operations.

simFunc.simulation_pool_function - deprecated

Should the multiprocessing issues ever be resolved, this is where the simulation specific processing pool will be configured.

A.3.4 COSMoS Satellite Functions

The *satelliteFunctions.py* file contains the two classes used by COSMoS, *HostSatelliteObject* and *AgentSatelliteObject*. The *HostSatelliteObjects* are responsible for interfacing current state data with the propagator and applying impulsive Δv values sent from an *AgentSatelliteObject* for the simulation. Object methods of note include *clear_old_prop_data* which empties old values from the host's stored propagation data based on the agent's reported simulation time, *update_mid_time* which updates the trigger time for further propagation, and *apply_maneuver* which re-propagates the state data based on a commanded Δv sent from an agent. The *AgentSatelliteObjects* are somewhat simpler, but they do contain the rudimentary GNC algorithm currently used by COSMoS in the *run_guidance* and *calculate_dv* methods. While currently rudimentary, these can eventually be replaced with more interesting GNC algorithms for a true simulation.

A.3.5 COSMoS State Propagator

The current propagator in use for COSMoS is a simple 2-body Keplerian propagator developed for GT-SORT. The code is stored in the file *gtSortAlgo.py* and is mostly the same as that used on GT-SORT except for the *propagate* and *convert_prop_data* functions which have been modified to fit the data into the format used by COSMoS as opposed to that used by GT-SORT.

A.4 COSMoS Files and function calls

A.4.1 COSMOSmainFunction.py

The code presented here is actually found in *simpCOSMOSmainFunction.py*. The original *COSMOSmainFunction.py* file still contains the code needed to call Basilisk through *execnet* and is, therefore, completely useless for running simulations. That said, it does contain information about how to implement Basilisk inside the MADNS framework.

```
def parse_simulation_data_file(fileName):
    return satTLEs


def generate_state_packet_data(targetDevice, dataFileElement):
    return initialStatePacket


def parse_state_data(packet):
    stateData = packet
    return stateData
```

While the current *parse_state_data* function is extremely simple, it can be modified as needed to work with various simulation classes.

```
def agent_class_setup(simConfigData, incomingQueue, outgoingQueue,
    ↪ loggingQueue):
    return simClasses


def host_class_setup(simConfigData, initialStateData, incomingQueue,
    ↪ outgoingQueue, loggingQueue):
    return simClasses


def host_simulation_function(simClasses, simConfigData, loggingQueue,
    ↪ incomingQueue, outgoingQueue, logFile, dataFile, TIME_0=0):
        <...>
        if not incomingQueue.empty():
            msg = incomingQueue.get()
            if msg[0] == 'agentTimeUpdate':
                LU.log_print('simFunc: host_simulation_function: received
                    ↪ agentTimeUpdate: {}'.format(msg), 3)
                # simClasses[agentSatelliteMapping[msg[1]]].
                    ↪ clear_old_prop_data(msg[2])
                simClasses[agentSatelliteMapping[msg[2][0]]].
                    ↪ clear_old_prop_data(msg[2][1])
            elif msg[0] == 'maneuverPacket':
                simClasses[agentSatelliteMapping[msg[1][1]]].apply_maneuver([
                    ↪ msg[1][1]])
                return None
            else:
                return msg
```

The most important thing here is that *host_simulation_function* return any messages needed by the system for logging, exiting, or any other purpose not directly related to the simulation.

```
def start_host_simulation(simConfigData, simClasses, simQueues,
    ↪ incomingQueue, outgoingQueue, loggingQueue, transmitLogFile,
    ↪ simLogFiles, simDataFiles, rateLogFile, rateDataFile, HOST_START_TO)
    ↪ :
    return 0


def start_agent_simulation(simConfigData, initialStateData, HWILinterface,
    ↪ simClasses, simQueues, incomingQueue, outgoingQueue, loggingQueue,
    ↪ transmitLogFile, simLogFiles, simDataFiles, rateLogFile,
    ↪ rateDataFile, TIME_0):
    return 0


def agent_simulation_function(simConfigData, HWILinterface, simClasses,
    ↪ simQueues, incomingQueue, outgoingQueue, loggingQueue,
    ↪ transmitLogFile, simLogFiles, simDataFiles, rateLogFile,
    ↪ rateDataFile, TIME_0):
```

The *agent_simulation_function* doesn't have a return statement as it is unlikely that the agent will be receiving any packets not meant for simulation once the simulation has started. It might be useful to build an exit hatch in the event of "EMERGENCY_SHUTDOWN" in the future.

```
def simulation_pool_function(simConfigData, loggingQueue, statePoolQueue,
    ↪ statePoolStopQueue, outgoingQueue, logFile, dataFile, rateLogFile,
    ↪ rateDataFile, TIME_0=0):
    return statePoolStopQueue.get()
```

This function is currently deprecated due to the multiprocessing issues.

```
def setup_agent_simulation(simConfigData, logRunPath, runNumStr,
    ↪ systemLogFile, incomingQueue, outgoingQueue, loggingQueue, TIME_0):
    return "AGENT_SIMULATION_READY", simClasses, simQueues, dataFiles,
        ↪ logFiles, initialStateData, HWILinterface


def setup_host_simulation(simConfigData, logRunPath, runNumStr,
    ↪ systemLogFile, incomingQueue, outgoingQueue, loggingQueue, TIME_0):
    return "HOST_SIMULATION_READY", simClasses, simQueues, dataFiles,
        ↪ logFiles, initialStateData


def send_host_setup_to_agents(simConfigData, incomingQueue, outgoingQueue,
    ↪ loggingQueue, initialStateData, simulationLogFile, transmitLogFile,
    ↪ HOST_START_TO):
    if agentsReady:
        return "SIMULATION_START"
    else:
        return "EMERGENCY_SHUTDOWN"


def run_host_simulation(simConfigData, simClasses, simQueues, incomingQueue
    ↪ , outgoingQueue, loggingQueue, transmitLogFile, simLogFiles,
    ↪ simDataFiles, rateLogFile, rateDataFile, HOST_START_TO):
```

```
    return "SIMULATION_COMPLETE", None


def run_agent_simulation(simConfigData, initialStateData, HWILinterface,
    ↪ simClasses, simQueues, incomingQueue, outgoingQueue, loggingQueue,
    ↪ transmitLogFile, simLogFiles, simDataFiles, rateLogFile,
    ↪ rateDataFile, TIME_0):
    return "SIMULATION_COMPLETE", None
```

A.4.2 hwilUtilities.py and dummyHWIL.py

The following HWIL utilities are part of the MADNS framework and have th following interfaces:

```
def setup_hwil_interface():
    return interface


def hwil_query_and_send(hostIdentifier, hwilProcessRate, hwilStopQueue,
    ↪ HWILdriver, logFile, dataFile, outgoingQueue, loggingQueue, TIME_0):


def hwil_process(hostIdentifier, hwilProcessRate, hwilStopQueue, HWILdriver
    ↪ , logFile, dataFile, outgoingQueue, loggingQueue, TIME_0):
```

The dummy function exists because the main simulation function contains the following import statement:

```
try:
    import hwilUtilities as HWIL
except ImportError:
    import dummyHWIL as HWIL
```

The dummy function prevents Python from throwing a syntax error on loading the simulation file but doesn't load up memory or any other resources as its code is empty:

```
def setup_hwil_interface():
    return None



def get_sense_hat_data(sense):
    return None


def hwil_process(hwilProcessRate, hwilStopQueue, HWILdriver, outgoingQueue,
    ↪  TIME_0):
    return None
```

A.4.3 gtSortAlgo.py

```
def ode_2body(t, x, mu):
        return dx


def convert_prop_data(xs, timeArray):
    return propData


def propagate(propTime, timeStep, x0):
        return propData
```

A.4.4 satelliteFunctions.py

```
class HostSatelliteObject:
    def __init__(self, agentID, satID, oe, propDuration, propTimeStep,
        ↪ incomingQueue, outgoingQueue, loggingQueue):


    def set_start_time(self):
```

```
    def update_mid_time(self):


    def rv_2_oe(self):


    def initial_propagation(self):
        return GTS.propagate(self.propDuration, self.propTimeStep, x0)


    def orbit_propagation(self, maneuverTime=None):
        return GTS.propagate(self.propDuration, self.propTimeStep, x0)


    def get_prop_data(self, propStartTime=None):
        return self.currentSimTime, self.rN, self.vN


    def propagation_check(self):
        if (time.time() + self.agentID - self.simT0) > self.propTimeMid:
            return True
        else:
            return False


    def continue_propagation(self):


    def clear_old_prop_data(self, agentSimTime):


    def send_prop_data_to_agent(self):


    def apply_maneuver(self, maneuverPacket):


class AgentSatelliteObject:
    def __init__(self, agentID, satID, incomingQueue, outgoingQueue,
        ↪ loggingQueue):


    def get_rv(self):
        return self.propData[0]


    def initialize_prop_data(self, initPropData):


    def add_new_prop_data(self, newPropData):


    def set_start_time(self):


    def send_sim_time_to_host(self):


    def calculate_dv(self, maneuver=False):
        return dV


    def run_guidance(self, maneuver=False):



def apply_delta_v(v0, dv):
    return vf
```

A.4.5 pythonVersionSwitch.py

This function was developed to implement the *execnet* library's communication between Python 3 and Python 2. COSMoS still uses Basilisk's utility to convert classical orbital elements into the initial inertial position and velocity vectors, so this library is still in use.

```
def call_python_version(Version, Module, Function, ArgumentList):
```

```
# ArgumentList must be a list of strings where each string is an
    ↪ argument to the function
# i.e. function(a,b) receives ArgumentList = ['a', 'b']


gw = execnet.makegateway("popen//python=python{}".format(Version))
channel = gw.remote_exec("""
        from {} import {} as the_function
        channel.send(the_function(*channel.receive()))
    """.format(Module, Function))
channel.send(ArgumentList)
return channel.receive()
```

APPENDIX B

GENERAL LESSONS LEARNED

The creation of MADNS and COSMOS was always an ambitious project, but the difficulty involved was unclear in two major points. The largest problem that arose is the relative weakness of Python's multiprocessing libraries. Secondly, the incorporation of different propagators into the framework was more difficult than expected because the Basilisk being written in Python 2.7 while the MADNS framework is in Python 3. A third, minor, issue is due to processing delays introduced by the Sense-Hat, but it is more an outgrowth of the multiprocessing issue than a separate problem. Each of these is discussed in the following subsections.

B.1 Python Multiprocessing Limitations

Python multiprocessing has several well-characterized limitations. The most substantial is Python's Global Interpreter Lock (GIL)[40]. The GIL is the largest bottleneck in most Python multiprocessing applications because it forces all processes to run in serial when the Python *Threading* library is used. The Python *Multiprocessing* library is designed to avoid this problem, but it has major problems which have much less available documentation.

The first is that, without a call to the operating system, Python automatically assigns all *Multiprocess.Process* processes to the same core as the parent. Therefore, while *Multiprocessing* makes processes more parallel than *Threading*, it does not make them multi-core parallel[36]. It is possible to bypass this on Unix-based systems with a call to:

```
      os.system("taskset -p 0xff \%d" \% os.getpid())
```

However, when this is used with heavily parallelized processes it dramatically increases the chances that MADNS will throw `os.PIPE` errors at random points during the execution when it interacts with the operating system and other processes running on the computer.

Another issue is that *Multiprocessing.Process* processes do not always exit cleanly. Several tests showed that the server and logging processes described in Section A.1 would not terminate even when they reached the end of the function. Use of the *LU.log_print* function proved that the processes successfully processed their *StopQueue* elements and reached their respective *return 0* statements. However, when the main executables checked if *outgoingServerProcess.is_alive()*, the response was always that the process was still running. There are many reasons for this, including the weakness of the *Multiprocessing.Queue* objects, but it required that all processes spawned in the course of a MADNS run be forcibly terminated by using *Multiprocessing.Process.terminate()*. This function sends an operating system *SIGTERM* to the process, but it was not always successful. In many cases, these persistent and un-killable processes caused the MADNS system to hang and be unable to log all its data.

Furthermore, the Python *Multiprocessing.Queue.qsize()* and *Multiprocessing.Queue.empty()* functions are not precise. This is a known issue in Python as the official documentation states that the *qsize()* method will only return an approximation of the number of data items in the queue. In multiple early attempts to run MADNS, at the conclusion of a run the system would simultaneously report *loggingQueue.empty() = True* and *loggingQueue.qsize() = 19342*. There is a high probability that this discrepancy is what caused the *Multiprocessing.Process* processes to fail to exit. Python will not exit a process or register it as stopped it if has open access to a *Multiprocessing.Queue* object. Even though the processes were programmed to check that all queues were empty before exiting, Python would still not allow them to exit because one element of the *Queue* object believed that it was still open, full, and being accessed. There were multiple attempts to employ the *Mutliprocessing.JoinableQueue* objects as these claim to resolve the above issues, but they caused even more slowdowns, hangs, and crashes than the base queue objects. They also suffered from the same mismatch between *qsize()* and *empty()* methods. This leads to the final and most catastrophic issue discovered in the development of MADNS.

The *Multiprocessing.Queue* structures do not fail gracefully. During the November-January rewrite of the MADNS system, multiple tasks were launched as separate processes. Experiments with the *Multiprocessing.Pool* and *Multiprocessing.Process* libraries showed that processing functions created using *Pool* were significantly slower than those made with *Process*, so the MADNS code base used *Process*. The reason for the slow performance of *Pool* became apparent during heavy testing of the MADNS system. Early benchmark tests came back with the terrible results seen in Fig. B.1.

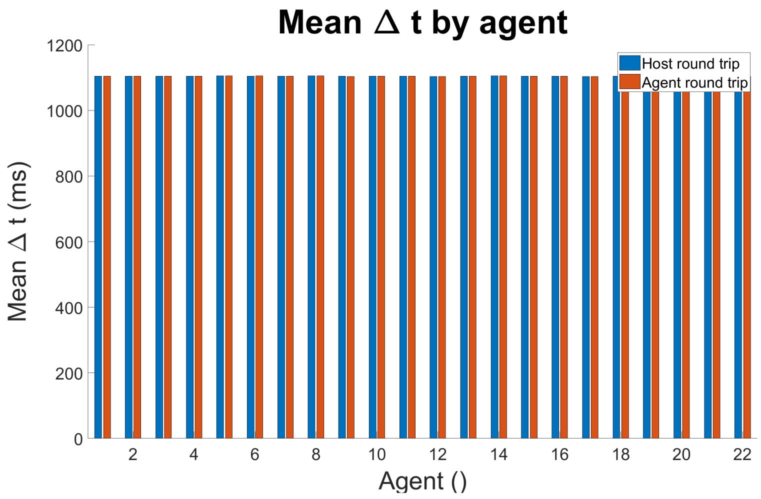


Figure B.1: Long delays found while using *Multiprocessing.Process*

The system was set to run at 20 Hz on both the host and agents, but each message had a round-trip time of 1.1 seconds. The problem was tracked to the host computer, where approximately 1.0 of the 1.1 second delay occurred. This was due to the system simply freezing when the *Multiprocessing.Queue* library failed. In low-usage cases, this problem could be rectified by setting a lower timeout on the *Queue.get()* commands, but that caused serious problems with heavy usage. In two cases, during particularly heavy usage, the host computer hung completely and required manual rebooting by disconnecting the power. Once the host was set to process all counter packets in serial, Fig. B.1 shows that the time delay was reduced to 50ms on each round trip as would be expected with the 20 Hz processing rate. The delay seen

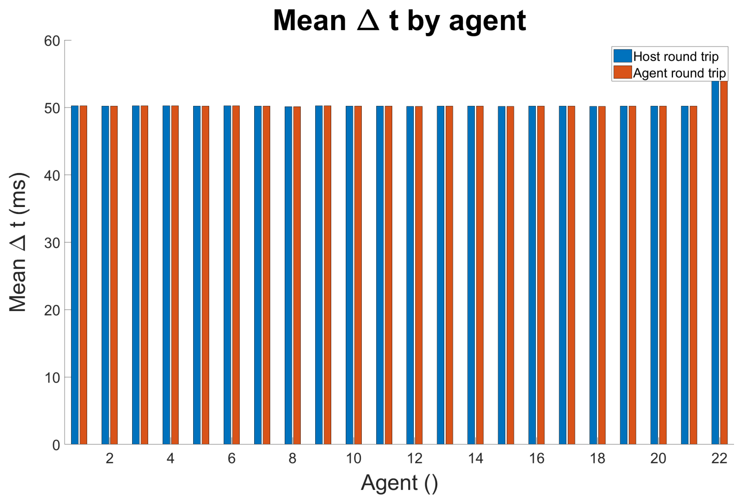


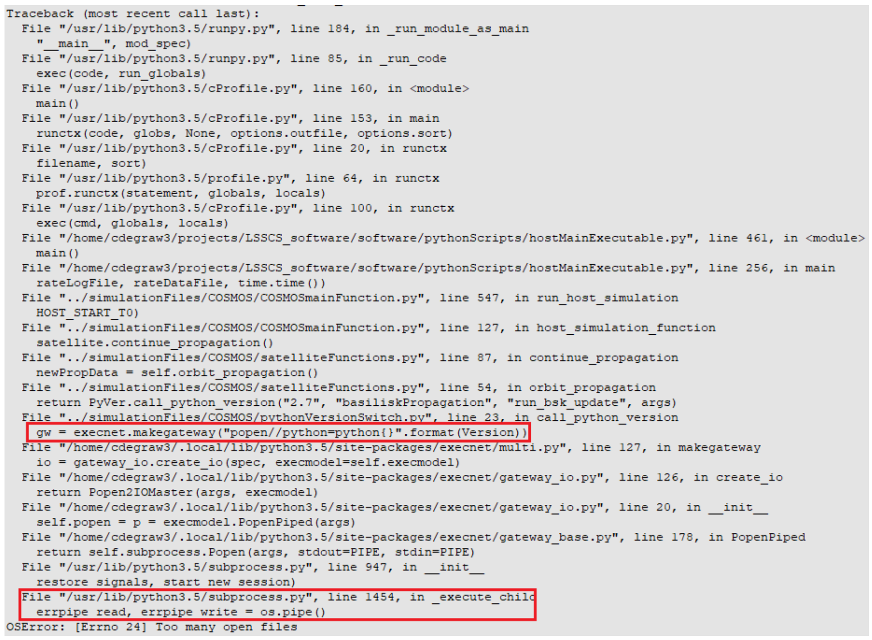
Figure B.2: Speedup caused by serial processing

on agent 22 was due to the implementation of the hardware interface which will be discussed in Section B.3.

B.2 Propagator Integration

As mentioned in Section 4.1.1, the original plan was to use CU Boulder's Basilisk astrodynamics toolkit. Basilisk is a powerful toolkit capable of propagating satellites with a speed-up of at least 365-to-1 (year-in-a-day). It also includes modules for atmospheric drag, solar radiation pressure, actuator models, non-rigid bodies, and the integration of guidance and control algorithms. However, Basilisk is written in Python 2 and MADNS is in Python 3. Tests showed that much of Basilisk's speedup is wasted unless its *SpacecraftObject* classes can be passed between functions, and without that COSMoS was unable to run in real-time. Therefore, the Python *execnet* library was used to communicate between Python 3 and Python 2 applications to preserve the integration speed.

Unfortunately, the *execnet* library was not written with multiprocessing, real-time execution in mind and certainly not astrodynamic propagation at a rate of 10 Hz. When tasked with propagating a single satellite, the system would fail in less than ten minutes with an *OS.Pipe* error as seen in Fig. B.2. When propagating 10 satellites, the system would fail within 45 seconds.



```
Traceback (most recent call last):
  File "/usr/lib/python3.5/runpy.py", line 184, in _run_module_as_main
    "__main__", mod_spec)
  File "/usr/lib/python3.5/runpy.py", line 85, in _run_code
    exec(code, run_globals)
  File "/usr/lib/python3.5/cProfile.py", line 160, in <module>
    main()
  File "/usr/lib/python3.5/cProfile.py", line 153, in main
    runctx(code, globs, None, options.outfile, options.sort)
  File "/usr/lib/python3.5/cProfile.py", line 20, in runctx
    filename, sort)
  File "/usr/lib/python3.5/profile.py", line 64, in runctx
    prof.runctx(statement, globals, locals)
  File "/usr/lib/python3.5/cProfile.py", line 100, in runctx
    exec(cmd, globals, locals)
  File "/home/cdegraw3/projects/LSSCS_software/software/pythonScripts/hostMainExecutable.py", line 461, in <module>
    main()
  File "/home/cdegraw3/projects/LSSCS_software/software/pythonScripts/hostMainExecutable.py", line 256, in main
    rateLogFile, rateDataFile, time.time())
  File "../simulationFiles/COSMOS/COSMOSmainFunction.py", line 547, in run_host_simulation
    HOST_START_T0)
  File "../simulationFiles/COSMOS/COSMOSmainFunction.py", line 127, in host_simulation_function
    satellite.continue_propagation()
  File "../simulationFiles/COSMOS/satelliteFunctions.py", line 87, in continue_propagation
    newPropData = self.orbit_propagation()
  File "../simulationFiles/COSMOS/satelliteFunctions.py", line 54, in orbit_propagation
    return PyVer.call_python_version("2.7", "basiliskPropagation", "run_bsk_update", args)
  File "../simulationFiles/COSMOS/pythonVersionSwitch.py", line 23, in call_python_version
    gw = execnet.makegateway("popen//python=python{}".format(Version))
  File "/home/cdegraw3/.local/lib/python3.5/site-packages/execnet/multi.py", line 127, in makegateway
    io = gateway_io.create_io(spec, execmodel=self.execmodel)
  File "/home/cdegraw3/.local/lib/python3.5/site-packages/execnet/gateway_io.py", line 126, in create_io
    return Popen2IOMaster(args, execmodel)
  File "/home/cdegraw3/.local/lib/python3.5/site-packages/execnet/gateway_io.py", line 20, in __init__
    self.popen = p = execmodel.PopenPiped(args)
  File "/home/cdegraw3/.local/lib/python3.5/site-packages/execnet/gateway_base.py", line 178, in PopenPiped
    return self.subprocess.Popen(args, stdout=PIPE, stdin=PIPE)
  File "/usr/lib/python3.5/subprocess.py", line 947, in __init__
    restore signals, start new session)
  File "/usr/lib/python3.5/subprocess.py", line 1454, in _execute_child
    errpipe read, errpipe write = os.pipe()
OSError: [Errno 24] Too many open files
```

Figure B.3: *OS.Pipe* error caused by *execnet*

It appears that this error is caused by insufficient garbage collection in the *execnet* module[41]. Overcoming this error would require elevated permissions on the host computer, a much greater understanding of OS pipes, or rewriting MADNS in Python 2. Unfortunately, none of these were possible in the course of this thesis.

Fortunately, the propagation code for GT-SORT is written in Python 3. This propagator is only a simple 2-body Keplerian approximation, but it was able to provide a functioning orbit propagator, and specifically one developed by an external group.

It should be noted that COSMoS still a Basilisk function to convert classical orbital elements gathered from the TLE file into inertial position and velocity vectors, so *execnet* is still used in COSMoS. However, any function using *execnet* takes approximately 0.5 seconds, while the same function executed purely in Python 2 takes approximately 0.05 seconds.

B.3 HWIL Processing Delays

As noted in Chapters 6 and 7, the Sense-Hat imposed significant processing delays on Agent 22 in the benchmark tests. A Sense-HAT query can take anywhere from 0.005 to 0.5 seconds before it returns its measurements. This would not be a problem in most cases as the HWIL interface unit could be a separate process running at its own rate independent of the main FSW algorithm. Unfortunately, the Python multiprocessing issues discussed in Section 8.2.1 were exacerbated on the SBCs, and the HWIL interface had to be run in serial with the primary agent functions as described in Section A.1.3. This resulted in an average of a 10 ms delay with a wide standard deviation on the HWIL-enabled agent as seen in Chapters 6 and 7. Should MADNS be rewritten in a language with more robust multiprocessing capabilities, this issue should be easy to correct.

B.4 Additional General Guides

The hardest lesson learned from this project is that Python 3 is not an appropriate language for this framework. The multiprocessing, speed, and cross-platform interface requirements are too high for the functionality that Python offers. Specifically, these requirements suggest that the framework will need to be written in C/C++ with the assistance of programmers who understand how code interacts with the operating system for processor affinity, cross-application communication pipes, and network communication.

Furthermore, real-time operations inside MADNS are handled using the *time.sleep()* command. In general, this command has an accuracy of $\approx$ 10 ms on a stock operating system[42]. On MADNS, this presented itself as a 1-2 ms standard deviation in the benchmark test timings as seen in Chapter 6. While this accuracy is sufficient for the current application, true real-time operation will require more substantial safeguards. Specifically, Ubuntu is able to implement a real-time kernel which could provide a much higher level of time accuracy if coupled with a C/C++ codebase[43].

B.5 Communication with an Operations Center and Ground Station

While this seems like a substantial challenge, the current construction of MADNS and COSMoS makes the task relatively trivial. The Georgia Tech Operations Center currently uses Python's *JSON* library to encode its packets for transmission across the Internet and GT network. Although MADNS currently uses *Pickle*, the transition to *JSON* should be relatively painless. As the COSMoS *simFunc.host_simulation_function* already cycles through every *HostSatelliteOjbect* on every iteration, it would be trivial to pick out the current propagated states, bundle them, and transmit the consolidated packet to the Ops Center. This was not implemented in the current version due to the long list of more pressing issues, but is an important next step in the development of MADNS and COSMoS.

B.6 Agent-to-Agent Communication

A critical concept discussed in Chapters 1 and 2 is the ability to simulate satellite-to-satellite (S2S) communications. While this was not implemented in the current version of MADNS and COSMoS, the path forward is clear and should be achievable with minimal work. The MPI design allows for the easy segregation of any packets labeled "agentComms" or something similar. Once selected out by *simFunc.host_simulation_function*, a time-of-flight algorithm can be implemented to determine at what point the message should arrive at its target based on the time it originated at its sender. This packet could then be routed to an agent with a specified delivery time.

Once at the agent, this packet would be routed to and stored in the *AgentSatelliteObject* instantiated on the agent. From there, a method such as *AgentSatelliteObject.deliver_message* can compare the agent's current *simTime* to the delivery time in the communication data and deliver it to the appropriate process at the appropriate *simTime*. This relatively simple implementation would allow for the simulation of satellite-to-satellite and satellite-to-ground communications without substantial reworks of either the MADNS or COSMoS frameworks.

B.7 MADNS

Referring again to the development guide seen in Fig. 3.2, there are a number of construction and algorithmic development tasks still in need of addressing.

B.7.1 System Hardware and Message Passing Interface:

Physically expanding the system is simply a matter of acquiring funding for purchasing more SBCs, their peripherals, and building more towers. However, the network communication algorithm proved unable to handle all 21 currently functioning agents when the transmission packet sized increased from the benchmark test's 3 floats to COSMoS's 100+ state element packets. This was temporarily addressed by expanding the incoming buffer size in the network communication library, but this fix is not very clean and was only able to sustain 10 agents during extended operations. Further development will require the creation of a more robust network communication protocol.

B.7.2 Agent to Agent Communication:

This is a critical element in the further development of MADNS and so was treated more fully in Section 8.3.1.

B.7.3 State of Health Monitoring:

Future versions of the SoH packet should include details about simulation and model health such as tracking of conserved quantities to limit numerical error.

B.7.4 Simulation Thread Generation:

As discussed above, the Python *Multiprocessing* library appears to be insufficient for the needs of the MADNS and COSMoS. Rewriting MADNS and COSMoS into

C/C++ (or at least Python 2.7 for Basilisk compatibility) will be a substantial task and will require the assistance of someone familiar with multiprocessing routines and OS pipe communication methods.

B.7.5 Data Logging:

Future developments include converting the CSV output to JSON files for data logging.

B.7.6 Simulation Time Management:

The time synchronization is effective, but not optimal. Improvements in the network communication and multiprocessing libraries will enable improvements in simulation time synchronization moving forwards.

B.7.7 Internal Data Management:

There are still no internal data management tools, including any real-time data displays. As this is another heavy multiprocessing load, it will require a substantial rework of the MADNS system and better multiprocessing capabilities before a real-time GUI becomes a possibility.

B.8 COSMoS

The current COSMoS simulation is much simpler than was originally hoped, but it has proved out the concept. Even with all the issues in running Basilisk under a Python 3 framework, Basilisk was able to propagate 10 seconds of high-fidelity in 0.01-0.03 seconds. By contrast, the much simpler GT-SORT propagator takes between 0.02 and 0.04 seconds to propagate 2-body Keplerian dynamics. Once the interface issues between Basilisk and COSMoS are resolved, it should be possible to run an extremely high fidelity simulation in real-time.

The following items are again drawn from the revised development guide found in Fig. 4.1.

B.8.1 Physics Simulation:

Basilisk is a full 6-DoF spacecraft simulator which includes non-rigid body dynamics and many perturbation models. Once it is integrated into COSMoS, the physics simulation will be extremely capable. If the multiprocessing and communications are managed better than is currently possible under Python 3, we should be able to incorporate any physics simulators beyond Basilisk.

B.8.2 Communication Network Algorithm:

The S2S communication simulation is still incomplete, but the revised MPI is fully capable of transmitting small telemetry messages and should be one of the less difficult future developments. The network has shown itself more than capable of passing the large state data packets, HWIL, and maneuver packets without loss or delay. Therefore, the addition of small, periodic S2S communications should be no major hurdle for the system.

B.8.3 Distributed Control Algorithm:

While the current GNC algorithm is much simpler than the planned Holzinger-McMahon algorithm[44, 45], there is no reason why replacing it with a more interesting algorithm should be any more difficult than replacing a function call.

APPENDIX C
SYSTEM BENCHMARK TEST RESULTS

The full set of results from the 8-hour system benchmark tests is shown below:

C.1 Counter function results

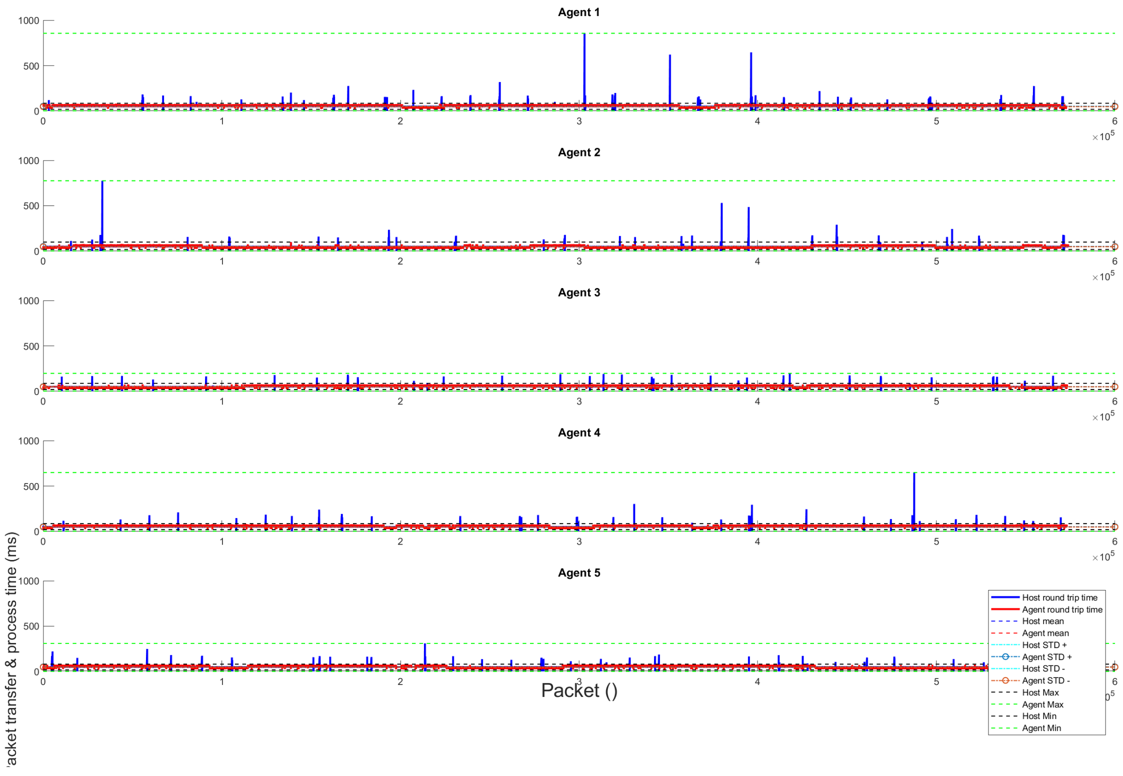


Figure C.1: 8-hour packet round-trip time for agents 1-5

Fig. C.1 demonstrates that no counter packets were lost during the benchmark test.

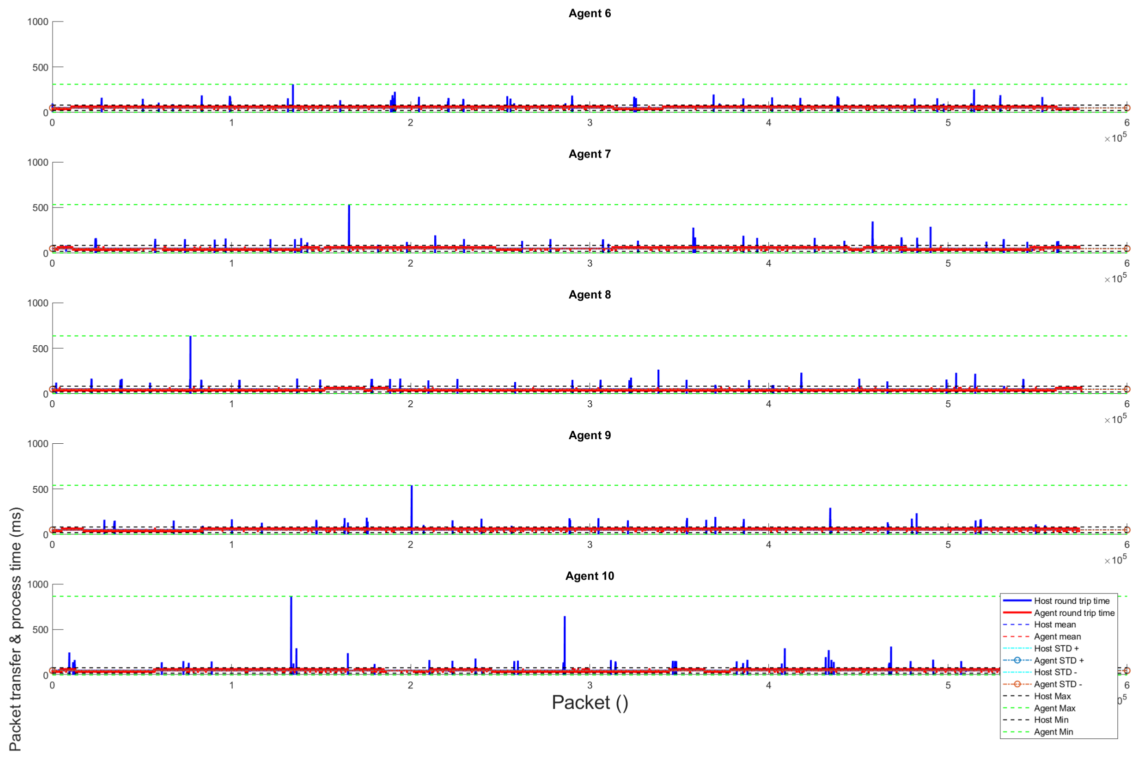


Figure C.2: 8-hour test packet round-trip time for agents 6-10

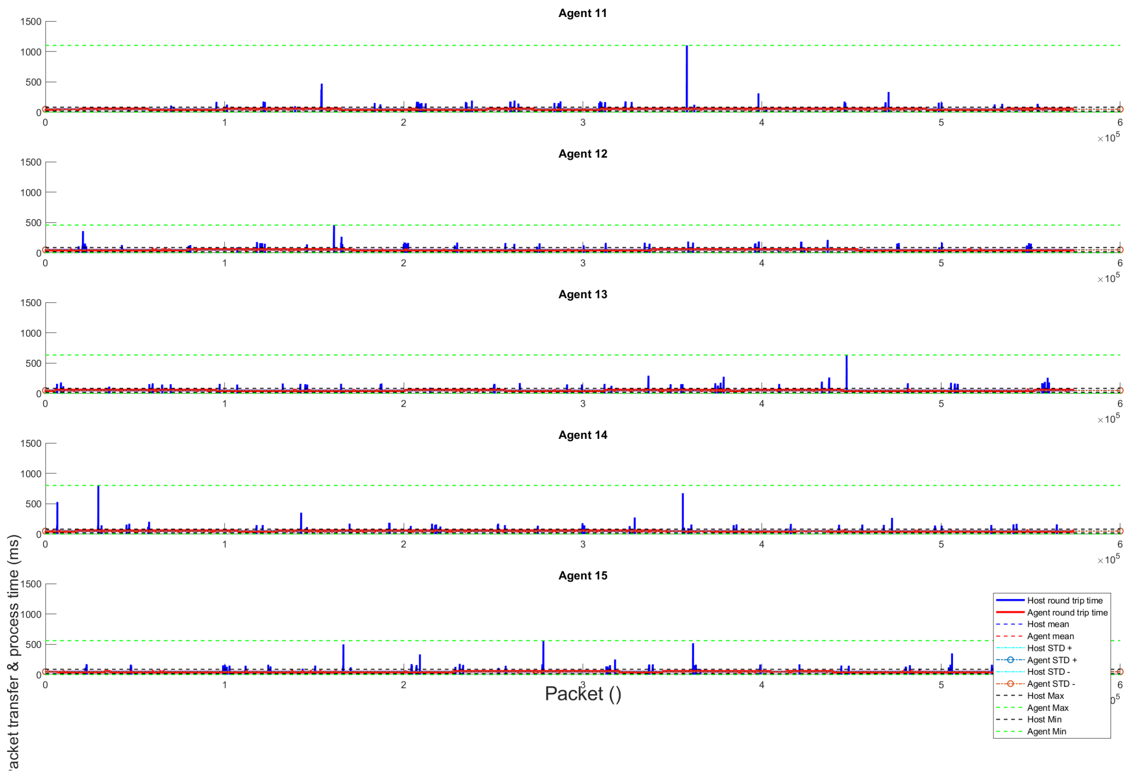


Figure C.3: 8-hour test packet round-trip time agents 11-15

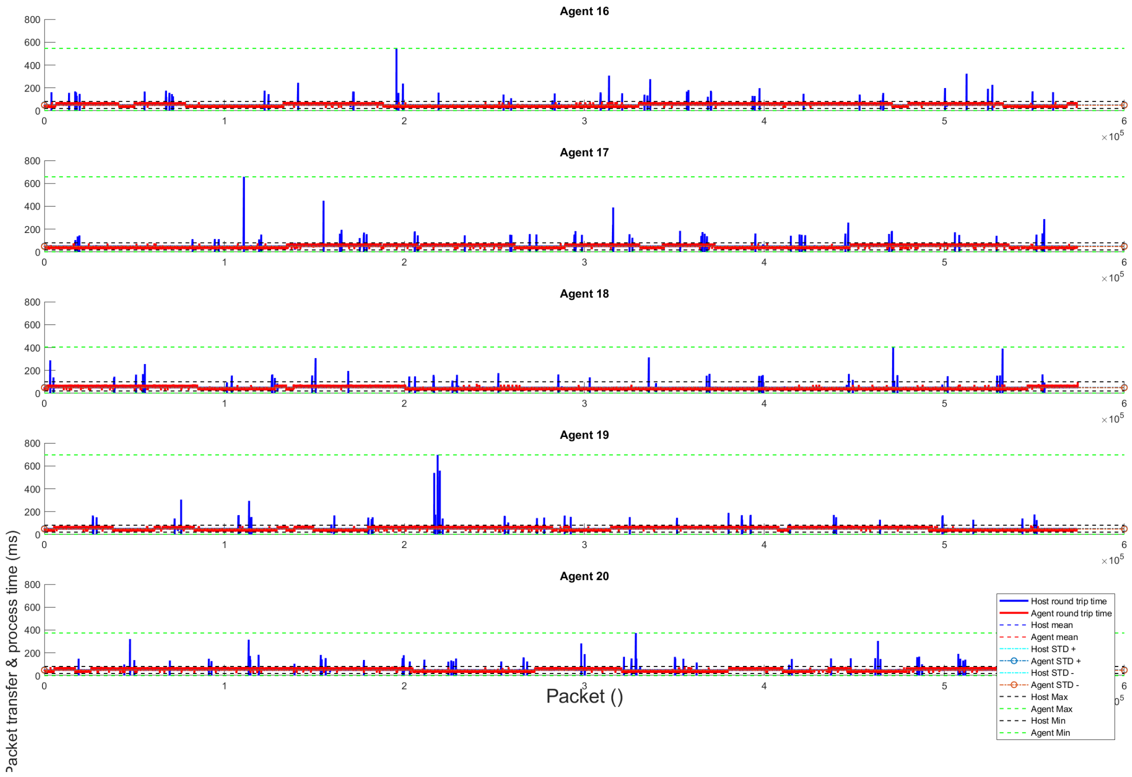


Figure C.4: 8-hour test packet round-trip time agents 16-20

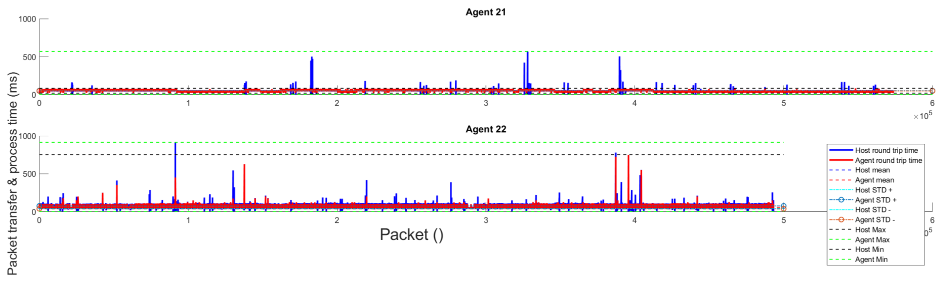


Figure C.5: 8-hour test packet round-trip time agents 21-22

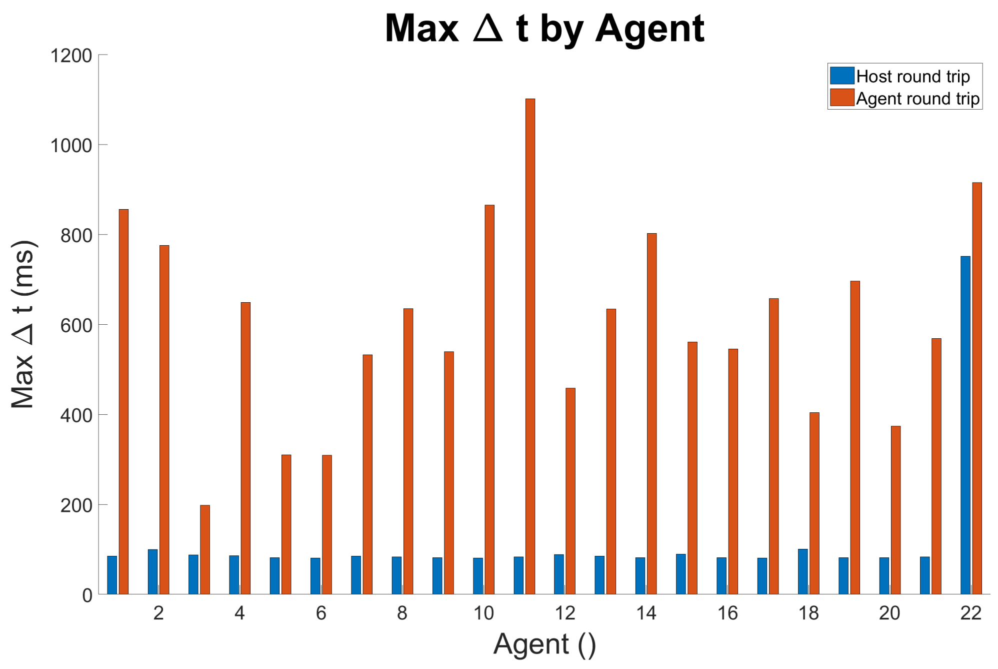


Figure C.6: 8-hour test packet maximum round trip time

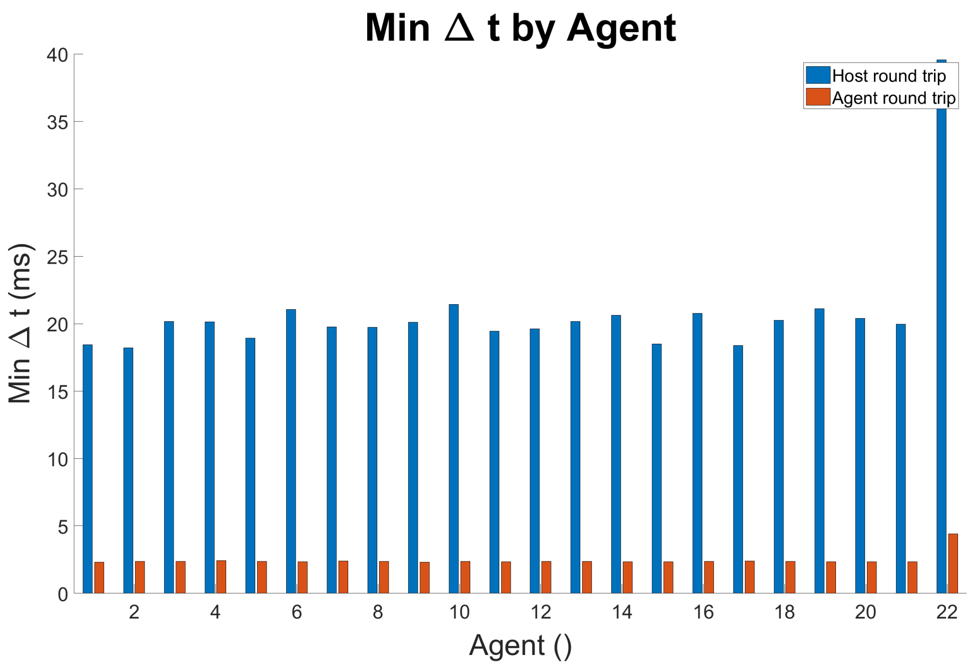


Figure C.7: 8-hour test packet minimum round trip time

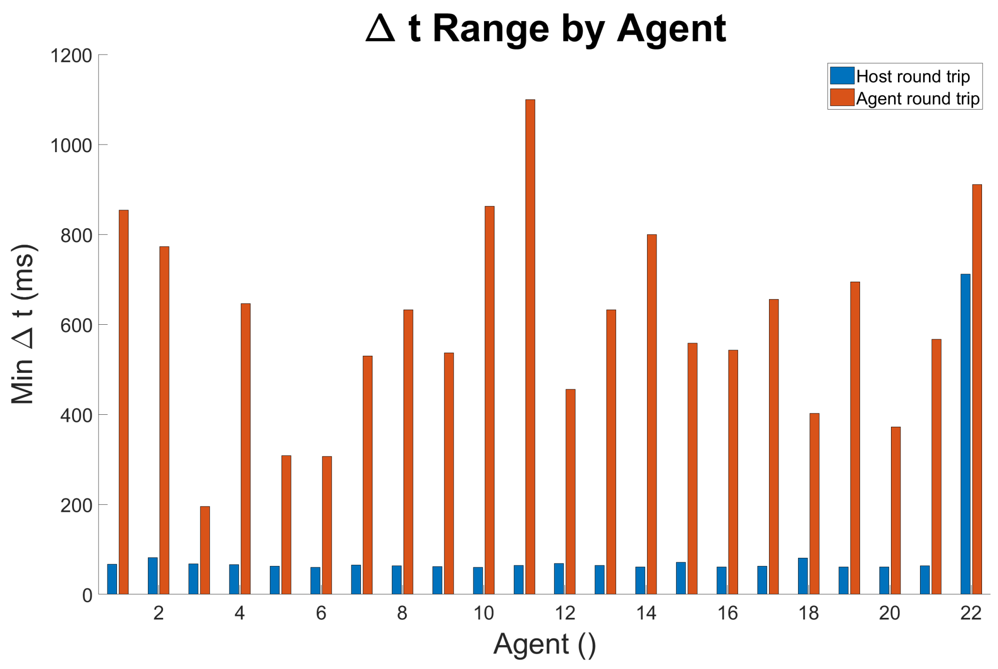


Figure C.8: 8-hour test packet round trip time range

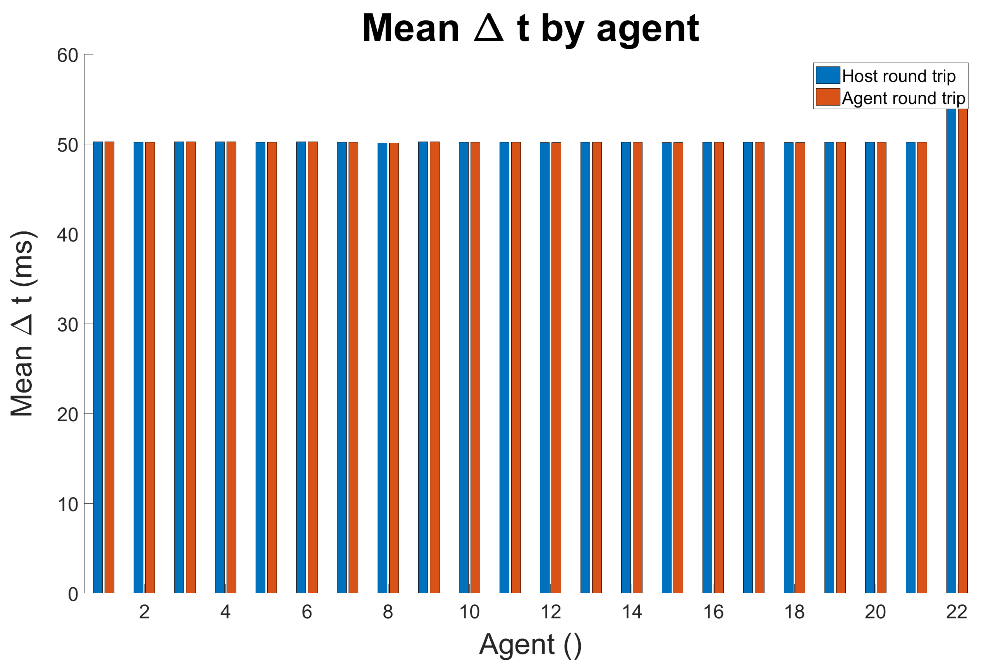


Figure C.9: 8-hour test packet average round trip time

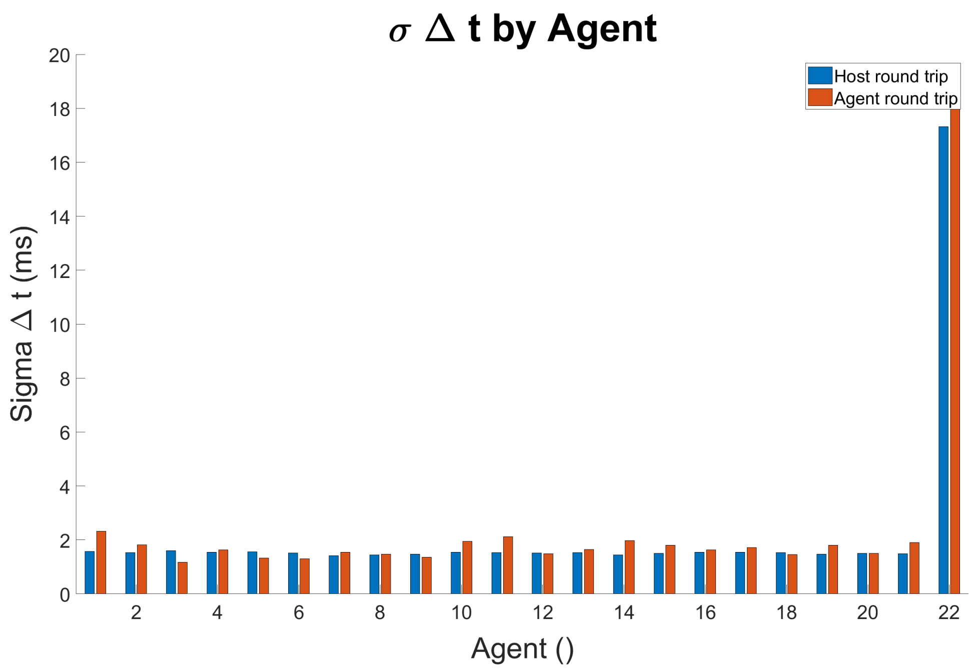


Figure C.10: 8-hour test packet round trip time standard deviation

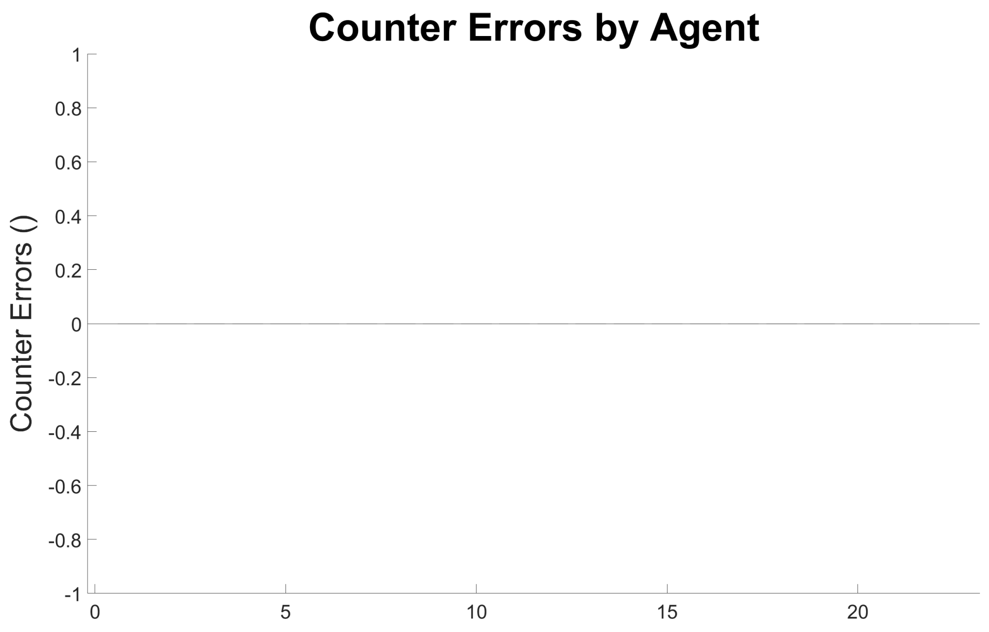


Figure C.11: 8-hour test lost packets

C.2 State of Health Results

C.2.1 Temperature

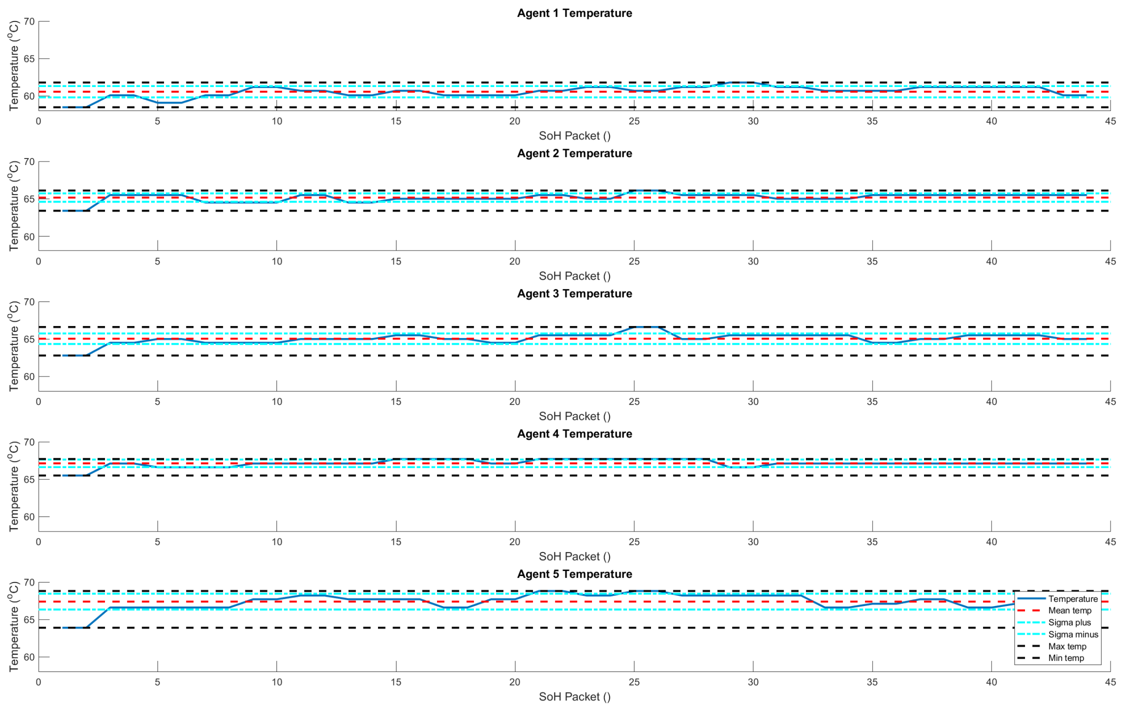


Figure C.12: 8-hour measured temperature agents 1-5

C.2.2 CPU Usage by agent

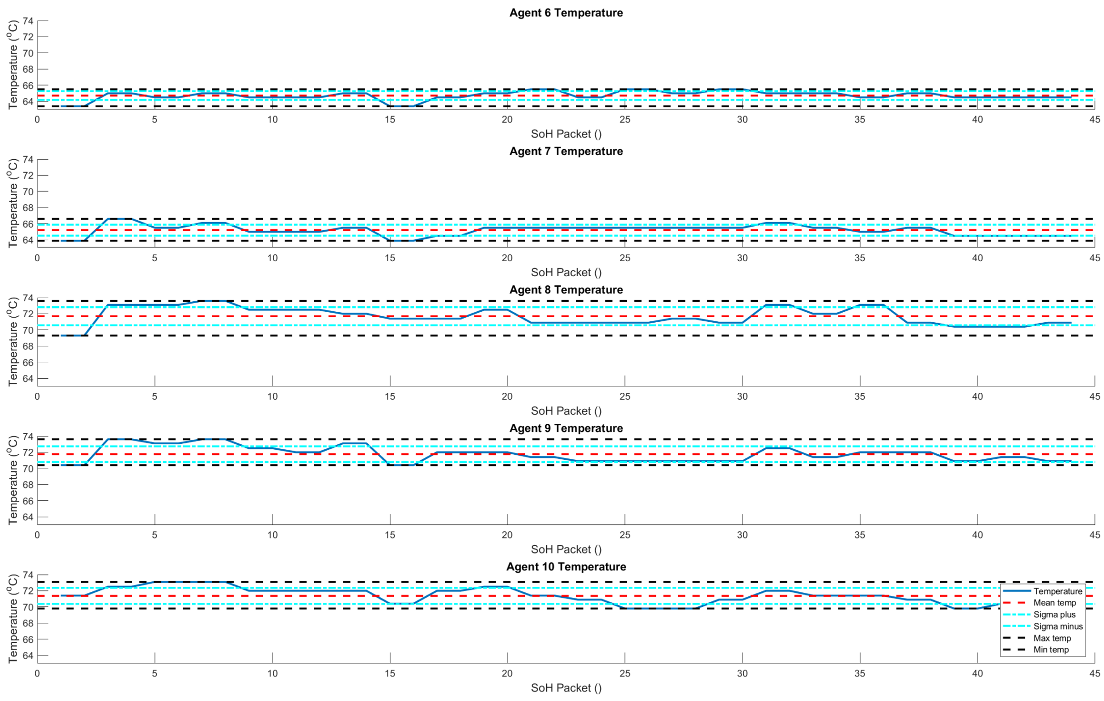


Figure C.13: 8-hour measured temperature agents 6-10

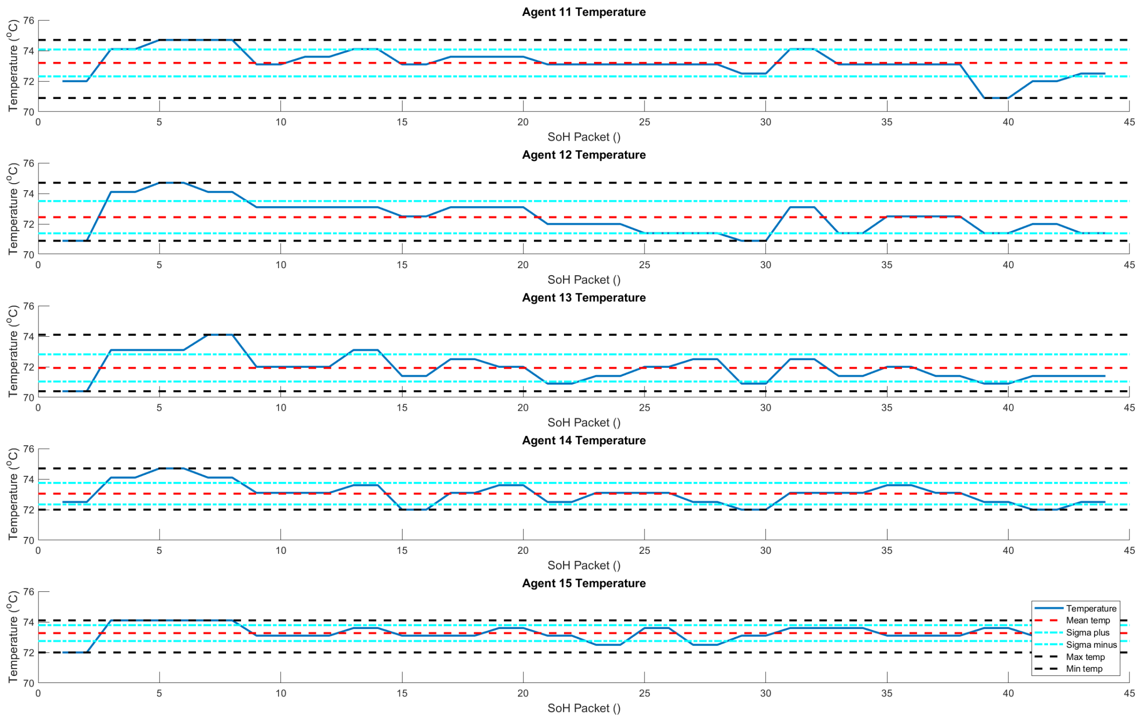


Figure C.14: 8-hour measured temperature agents 11-15

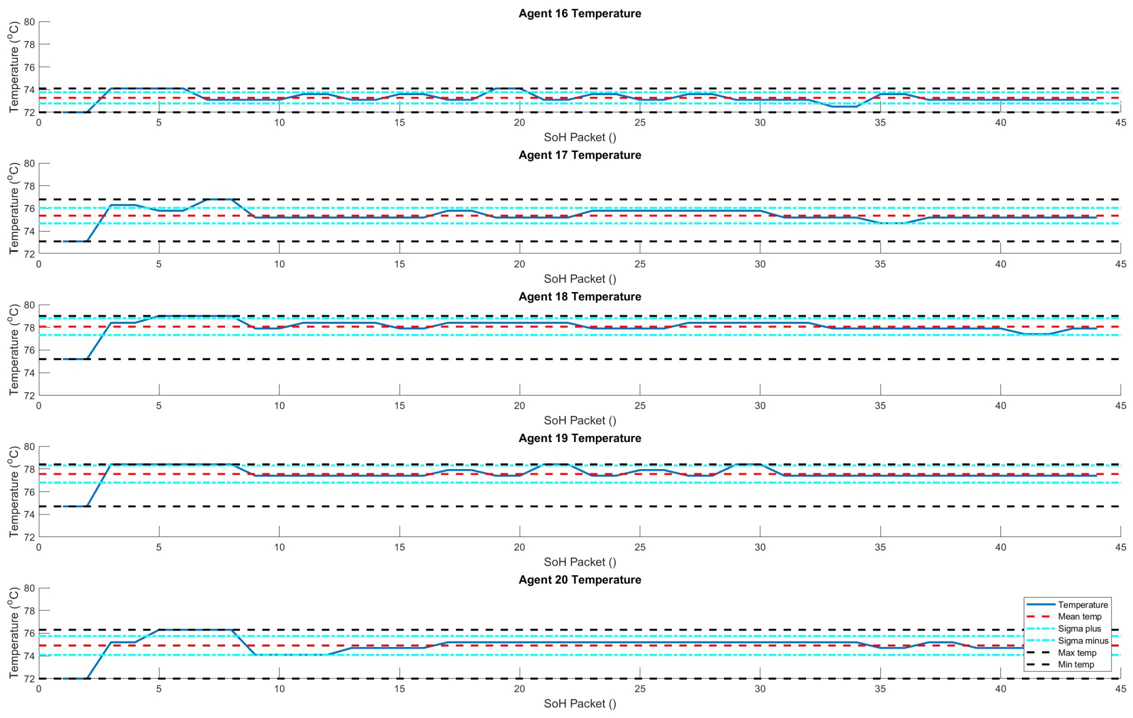


Figure C.15: 8-hour measured temperature agents 16-20

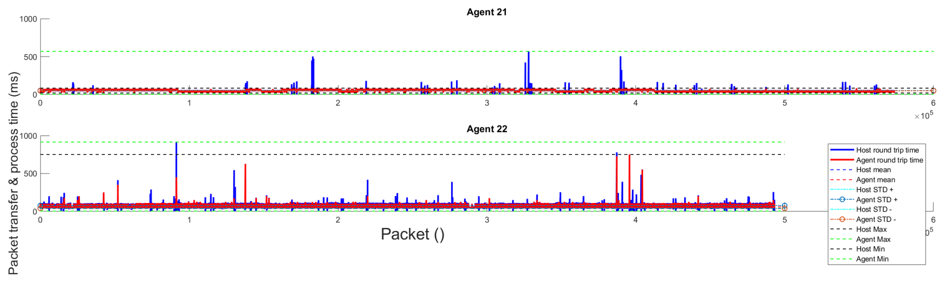


Figure C.16: 8-hour measured temperature agents 21-22

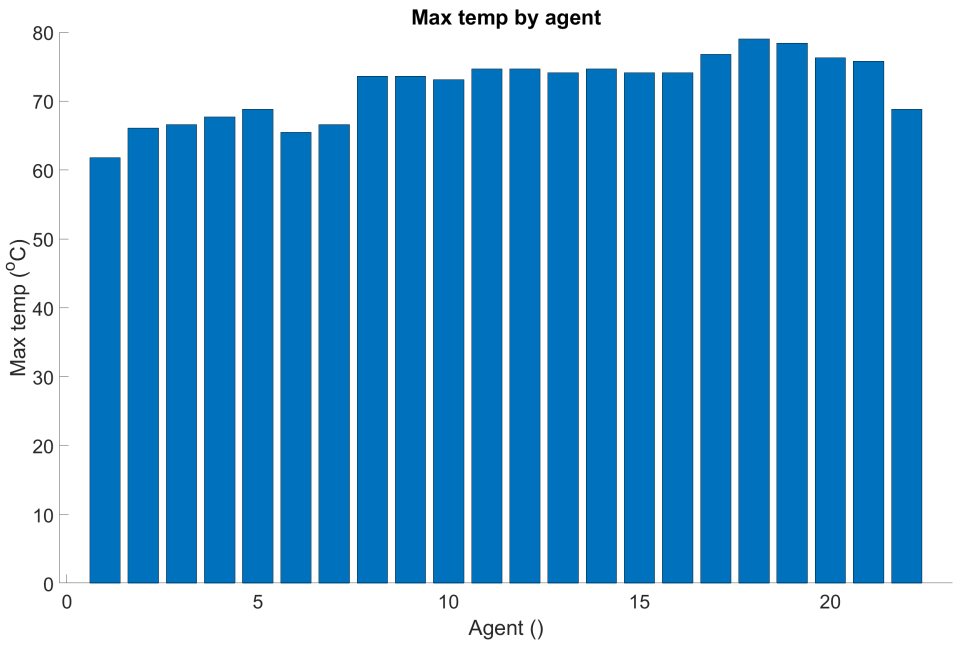


Figure C.17: 8-hour maximum temperature by agent

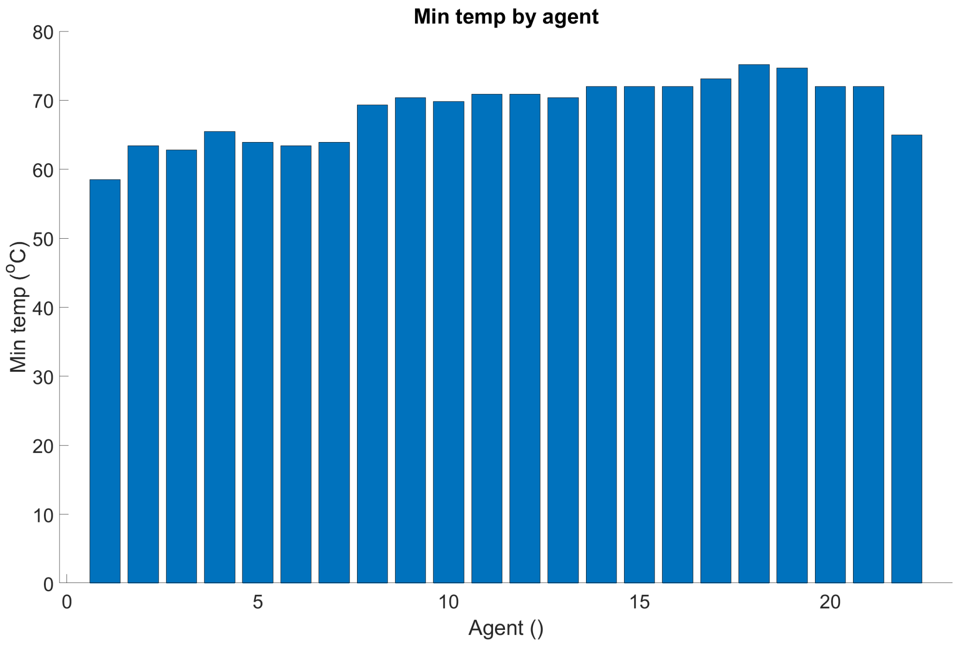


Figure C.18: 8-hour minimum temperature by agent

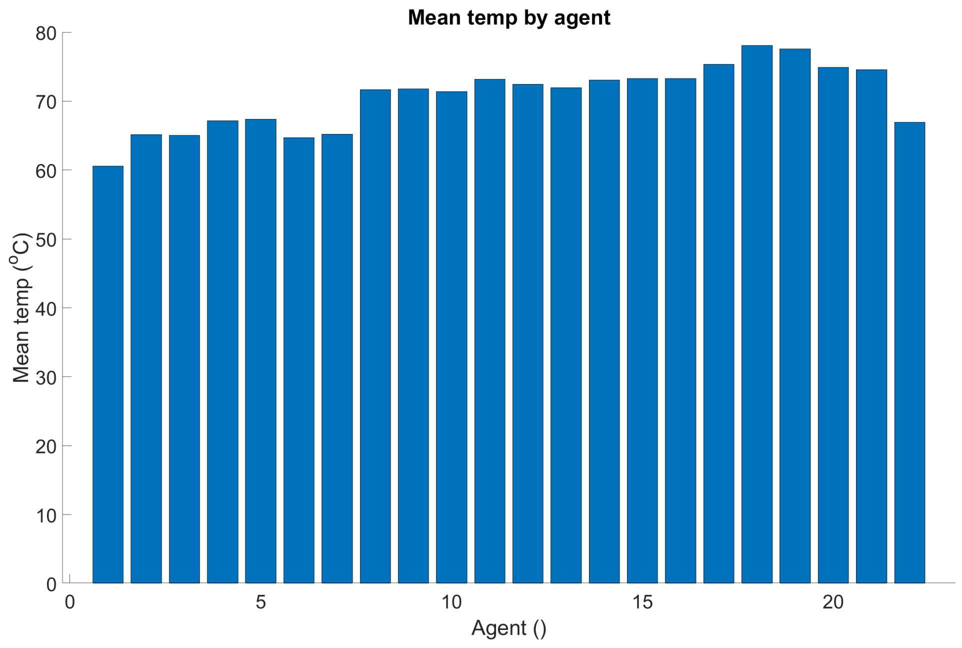


Figure C.19: 8-hour average temperature by agent

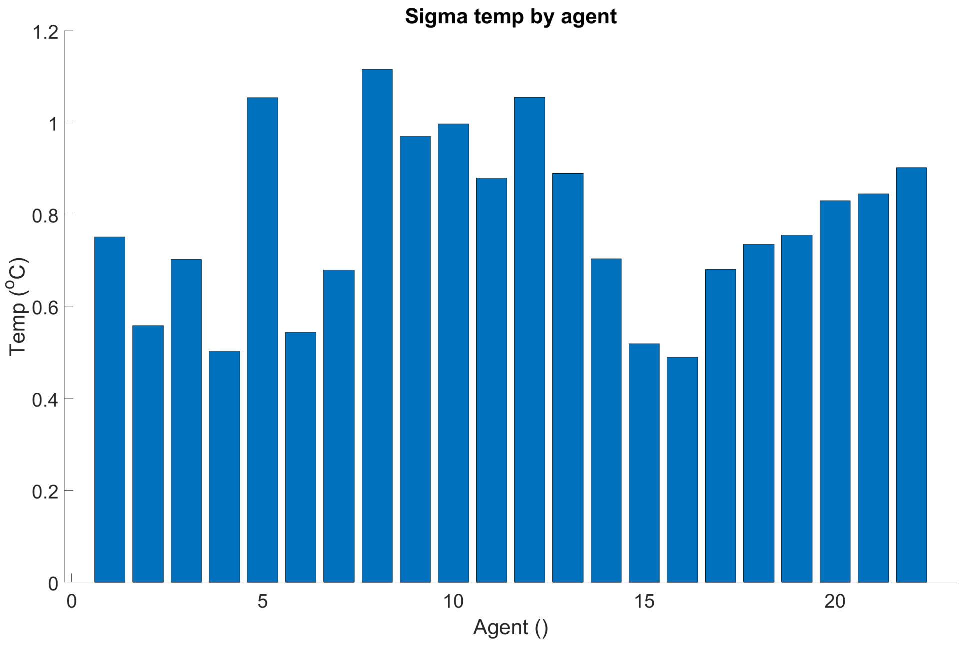


Figure C.20: 8-hour temperature standard deviation by agent

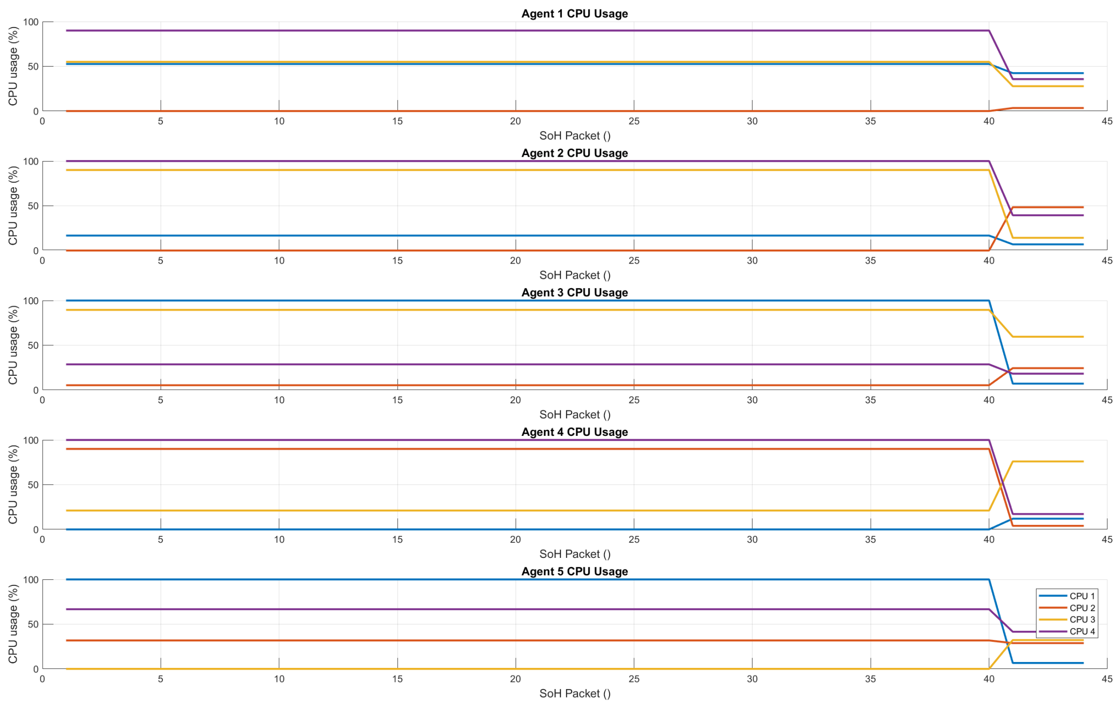


Figure C.21: 8-hour measured CPU usage agents 1-5

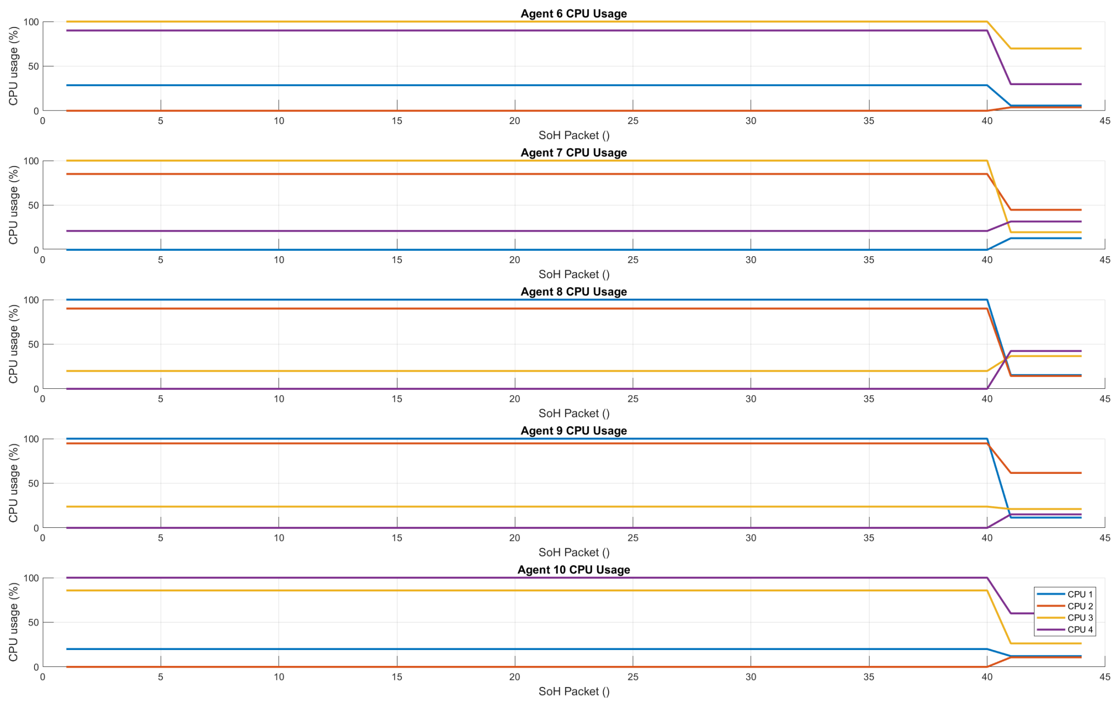


Figure C.22: 8-hour measured CPU usage agents 6-10

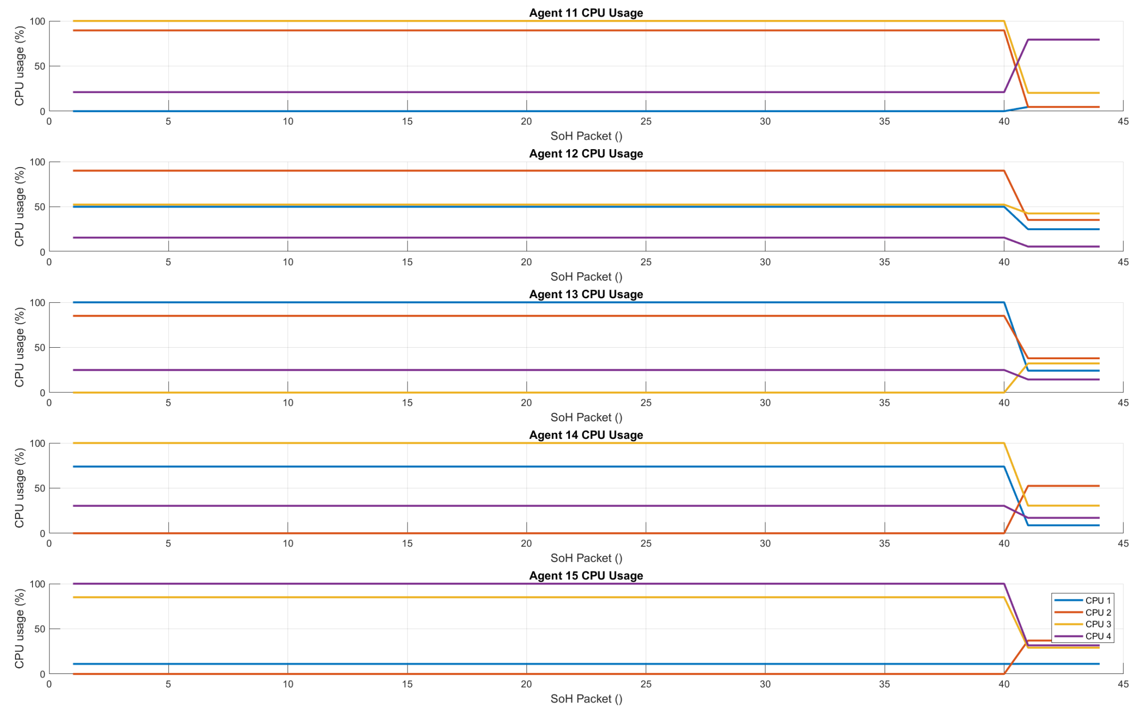


Figure C.23: 8-hour measured CPU usage agents 11-15

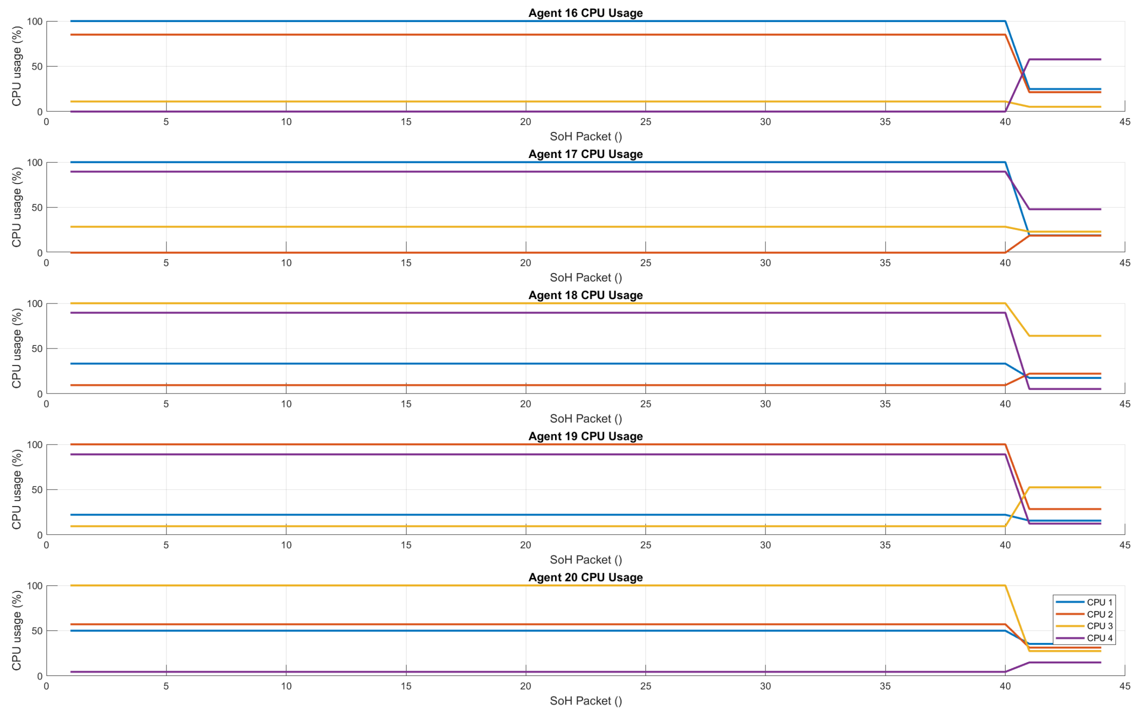


Figure C.24: 8-hour measured CPU usage agents 16-20

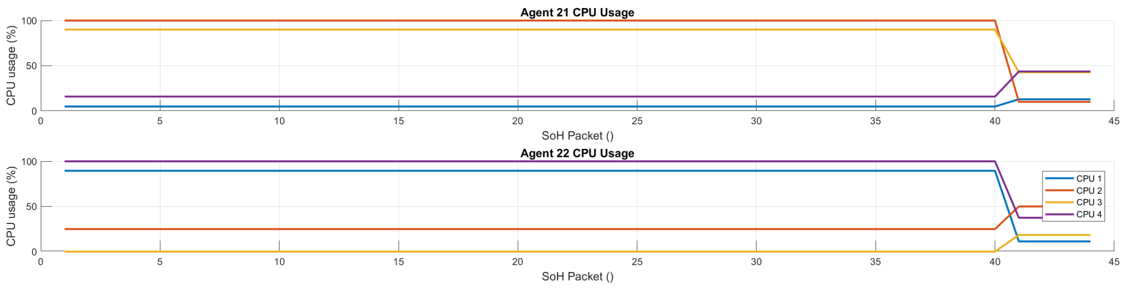


Figure C.25: 8-hour measured CPU usage agents 21-22

C.3 HWIL Data

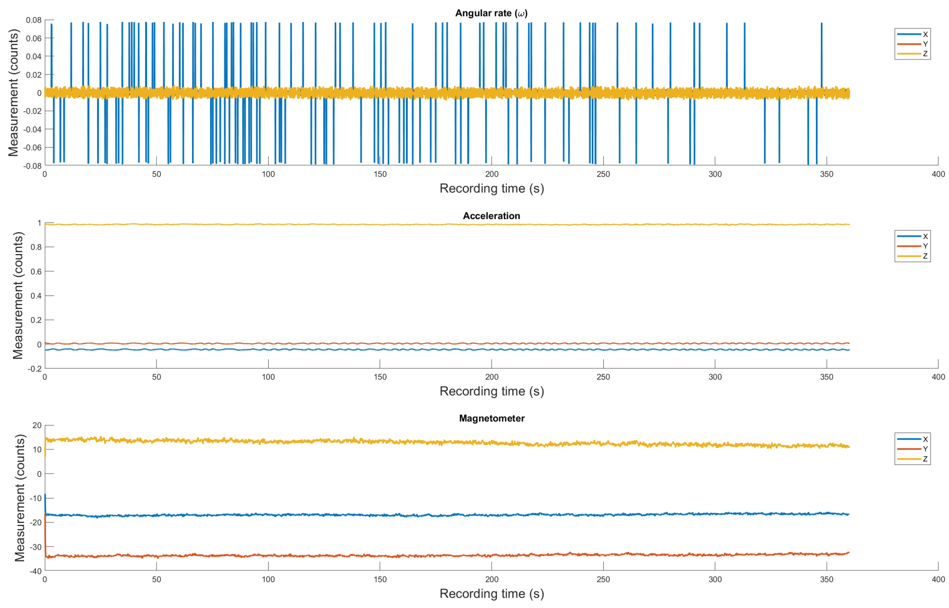


Figure C.26: 8-hour recorded HWIL gyro, accelerometer, and magnetometer

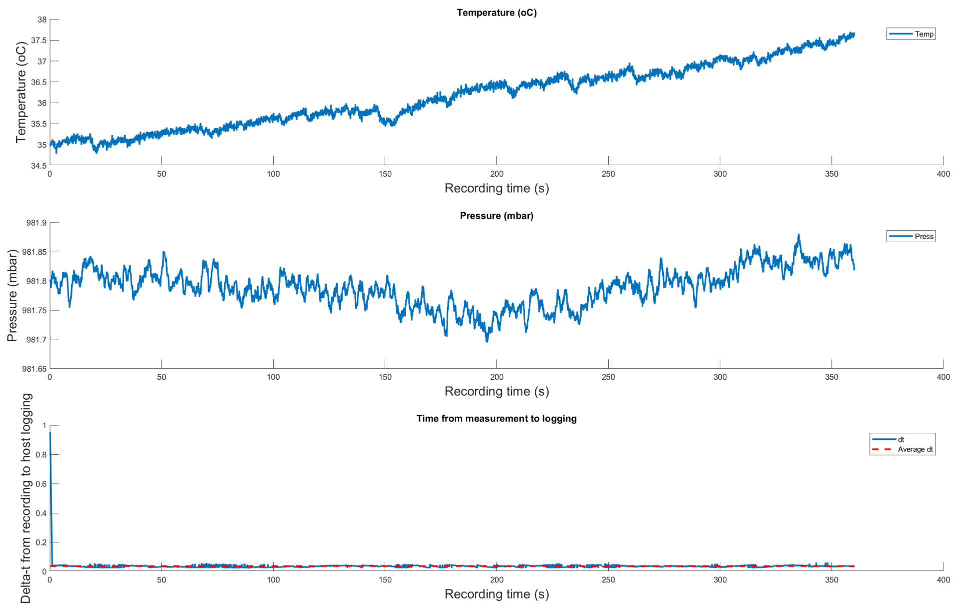


Figure C.27: 8-hour recorded temperature, pressure, and calculated packet Δt

APPENDIX D

COSMOS SIMULATION RESULTS

D.1 Base Simulation Results

The following results were obtained from a 1-hour COSMoS HWIL simulation without any guidance commands. Note: There was an error in the algorithm used to calculate specific orbital energy. The system neglected to include mass-specific gravitational potential in the orbital energy, which is what caused the oscillatory pattern in the orbital energy. Therefore, the quantity shown is actually the mass-specific kinetic energy of the spacecraft, not the total mass-specific orbital energy.

D.1.1 Orbital results

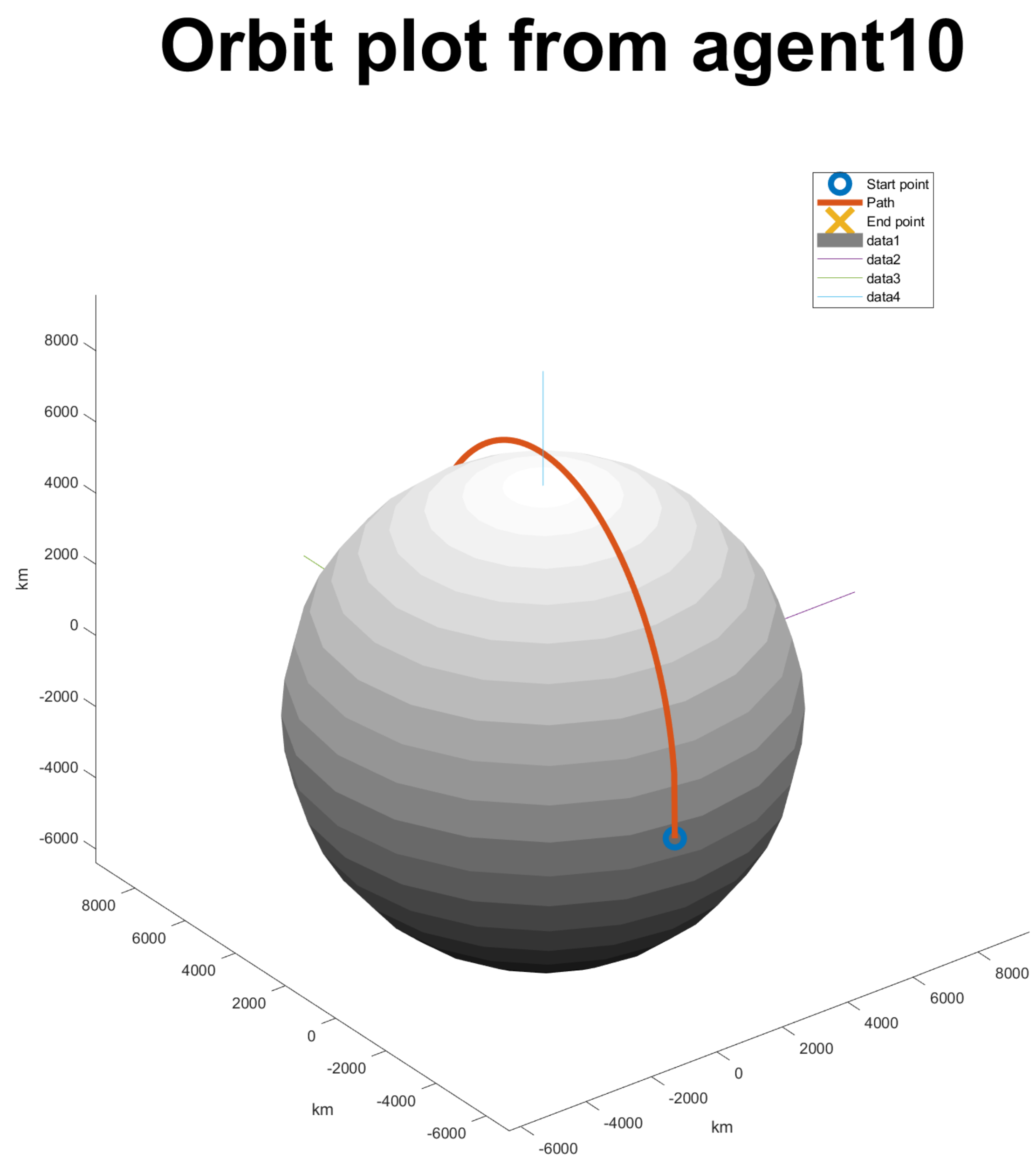


Figure D.1: Orbit plot from agent 10

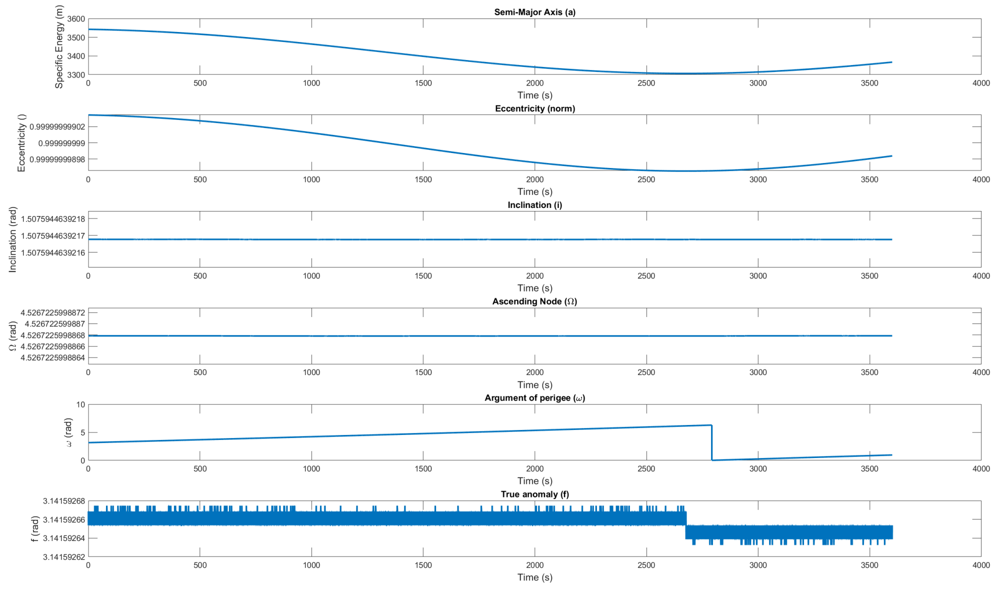


Figure D.2: Classical orbital elements from agent 10

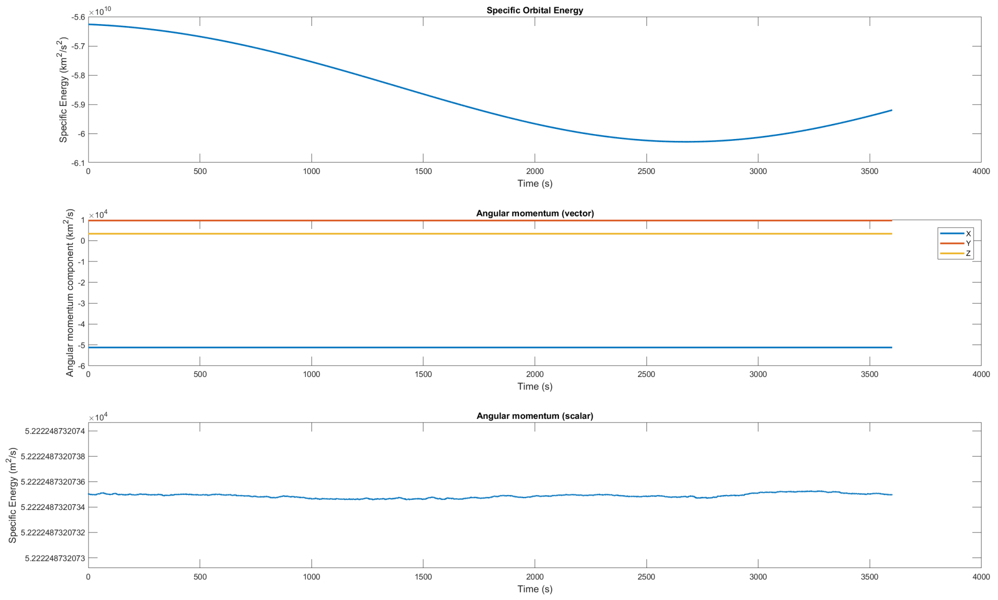


Figure D.3: Angular momentum agent 10

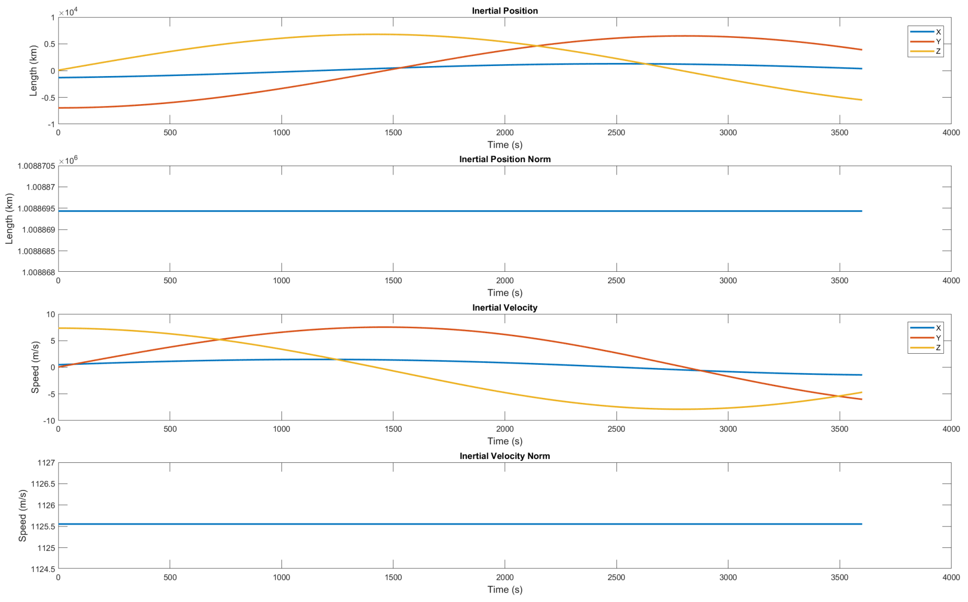


Figure D.4: Inertial position and velocity from agent 10

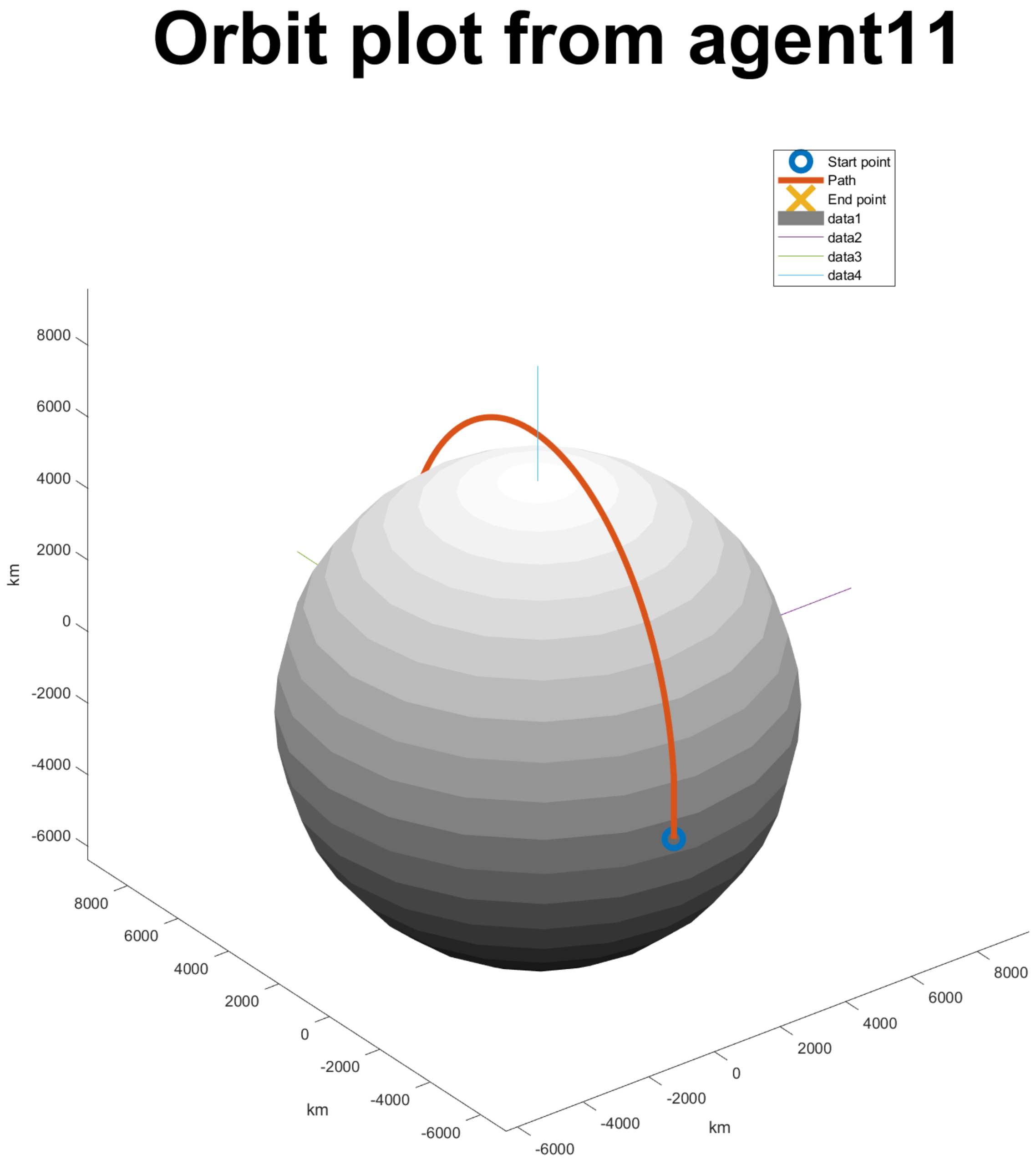


Figure D.5: Orbit plot from agent 11

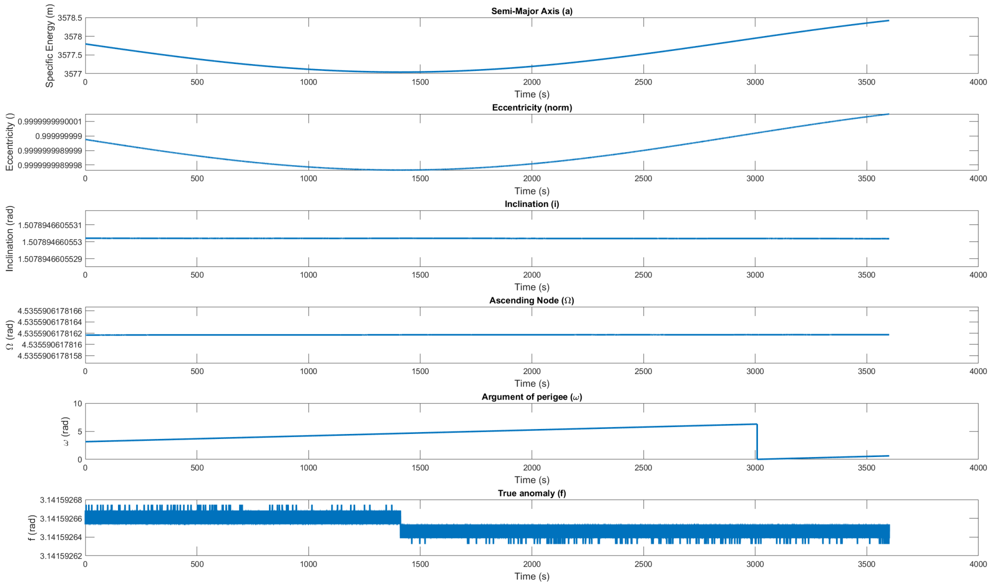


Figure D.6: Classical orbital elements from agent 11

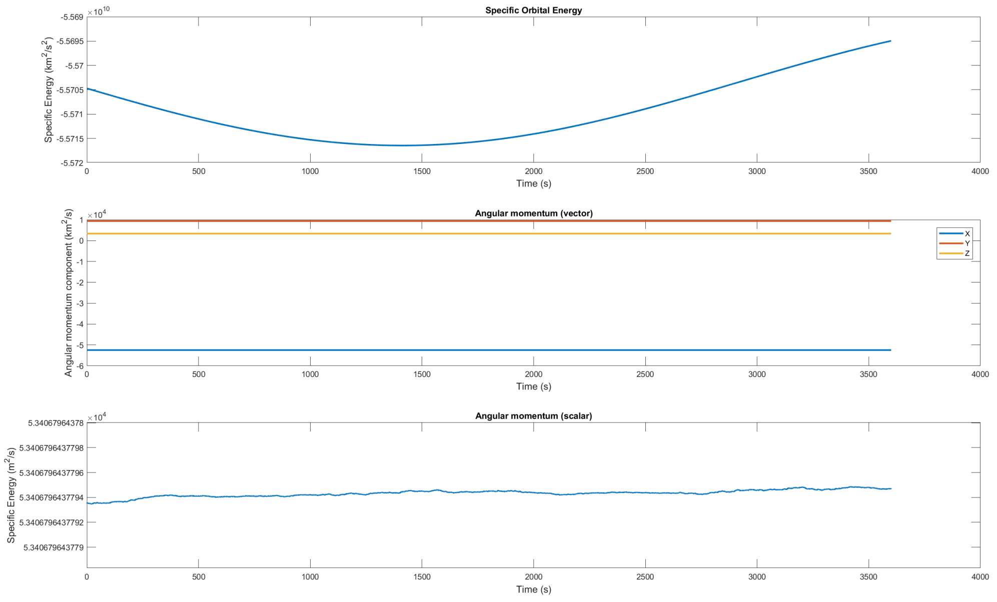


Figure D.7: Angular momentum agent 11

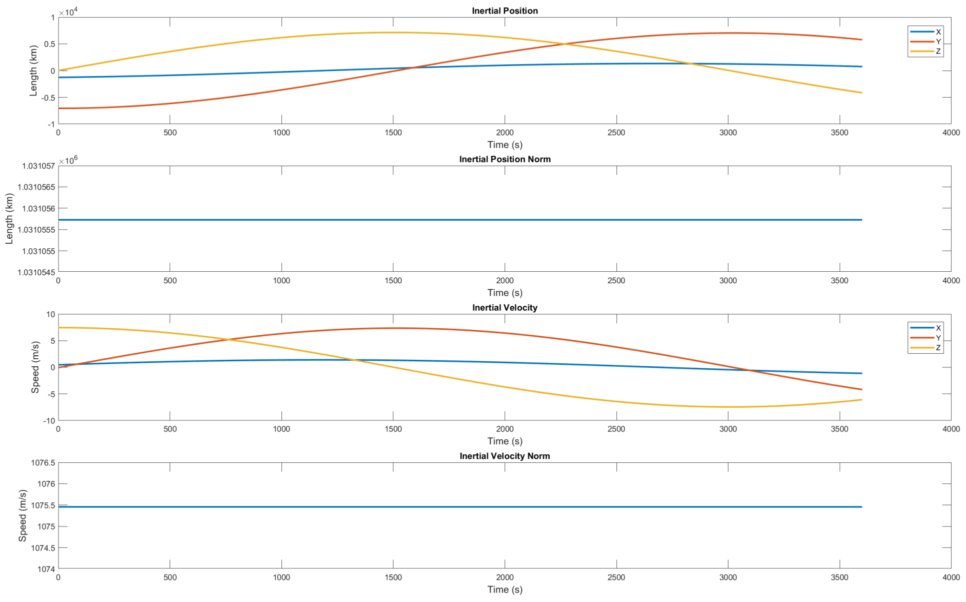


Figure D.8: Inertial position and velocity from agent 11

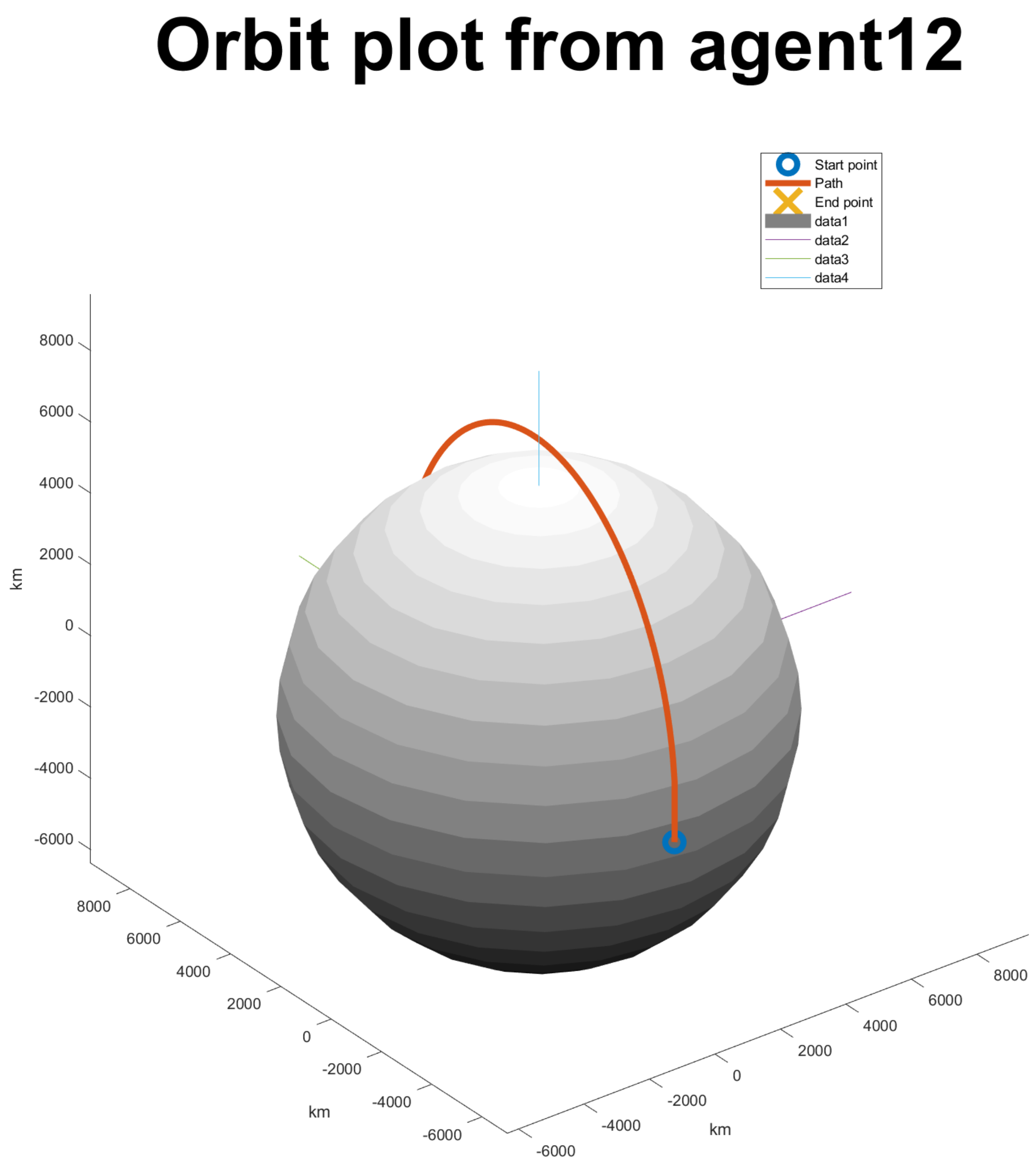


Figure D.9: Orbit plot from agent 12

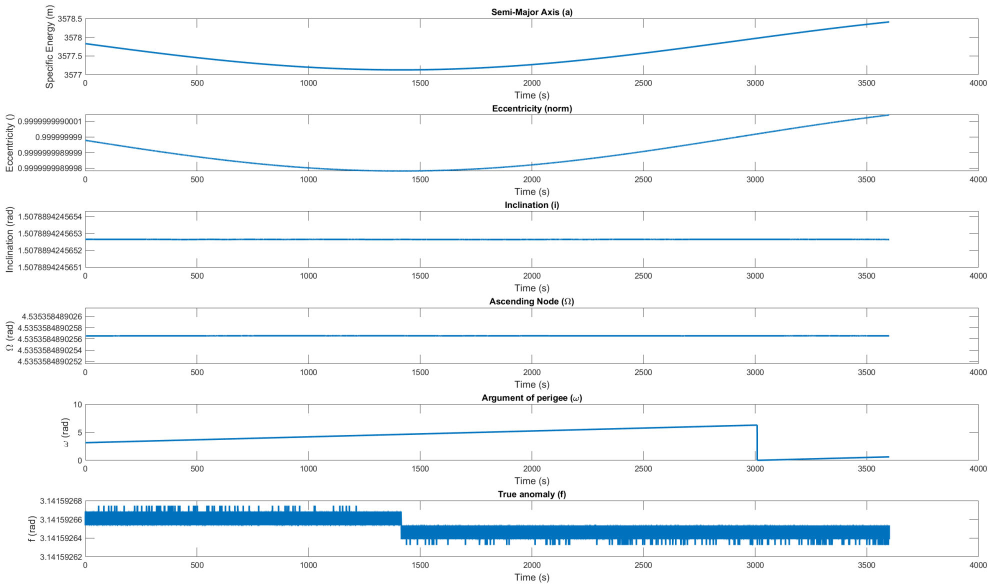


Figure D.10: Classical orbital elements from agent 12

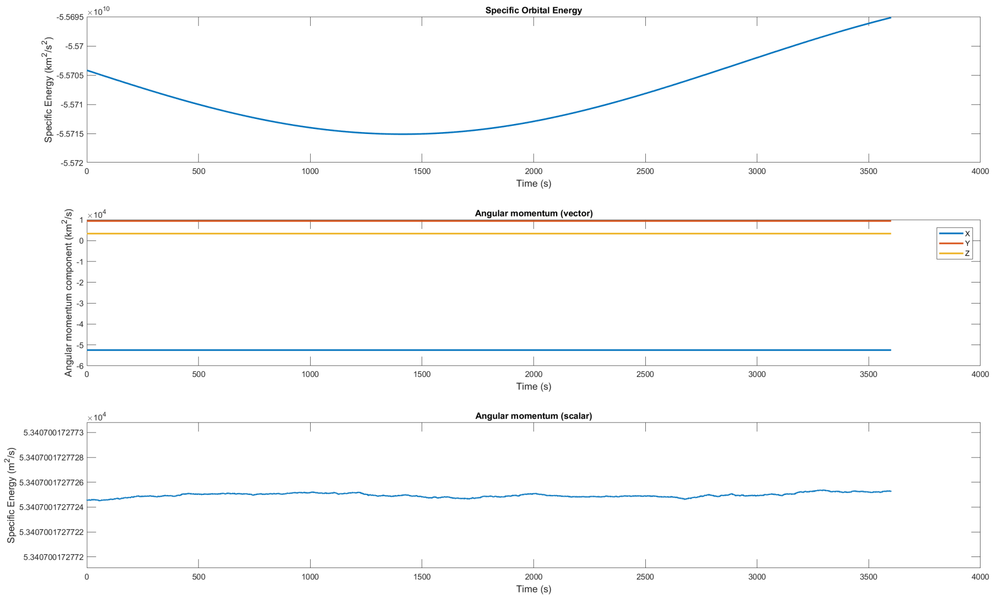


Figure D.11: Angular momentum agent 12

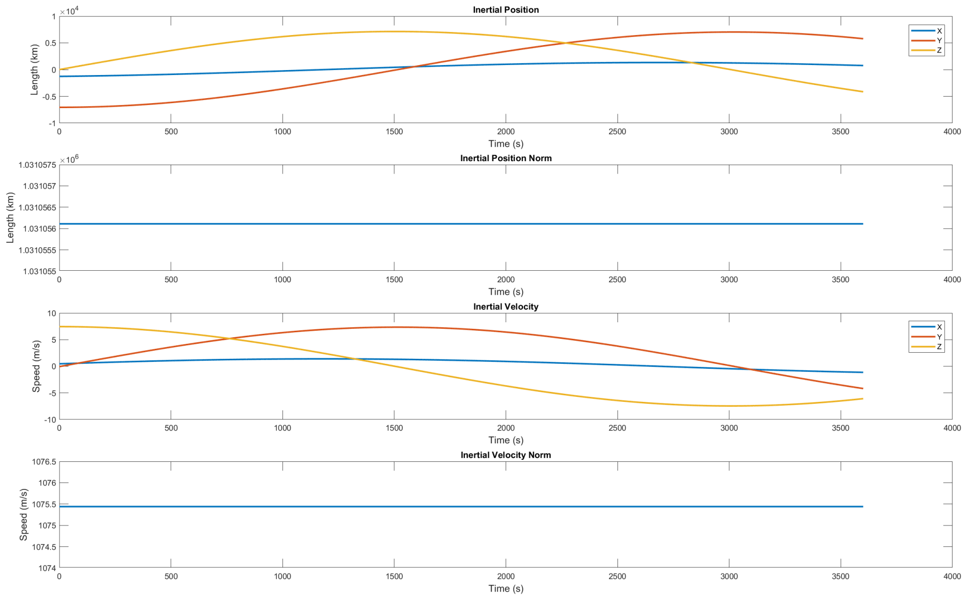


Figure D.12: Inertial position and velocity from agent 12

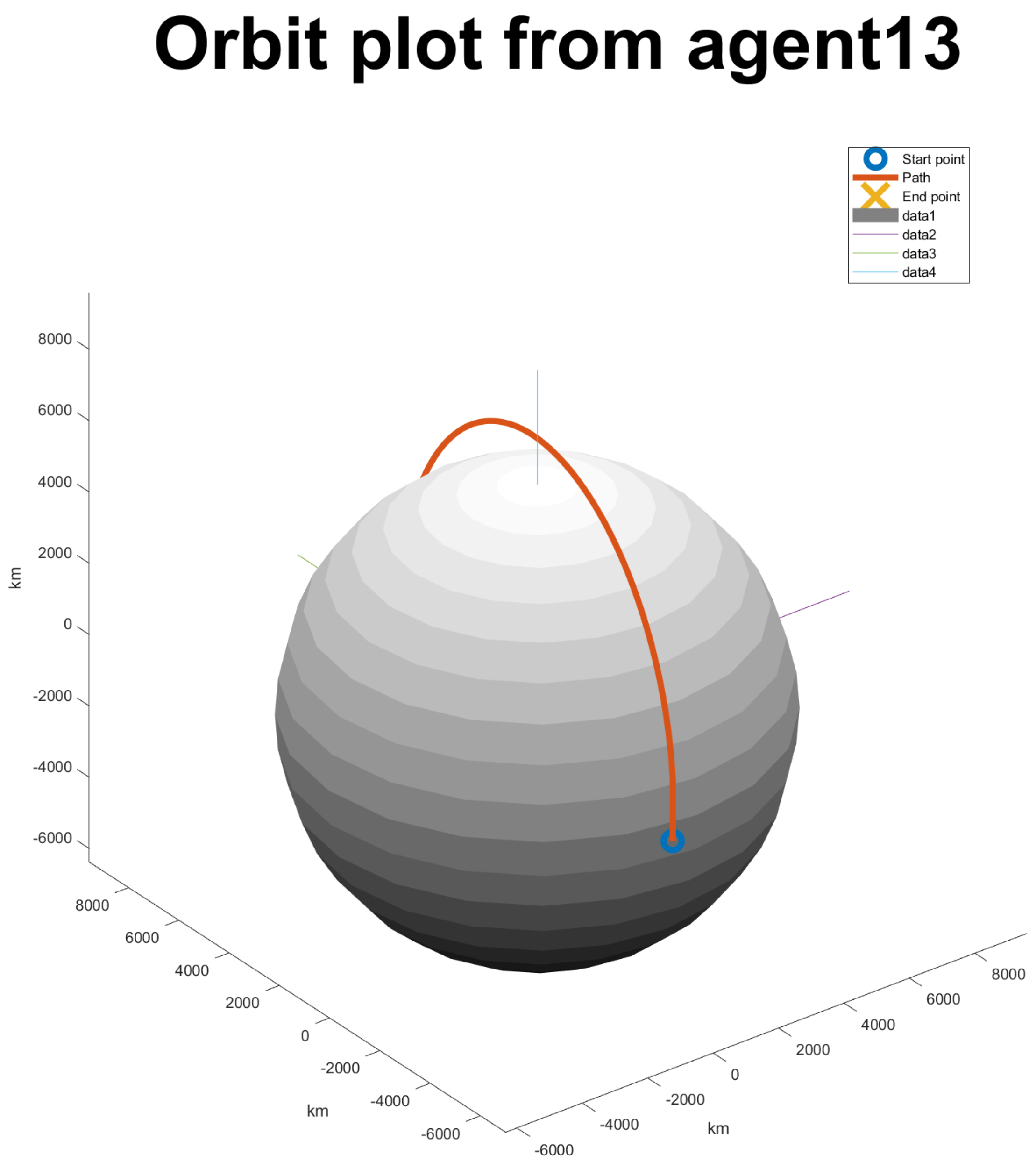


Figure D.13: Orbit plot from agent 13

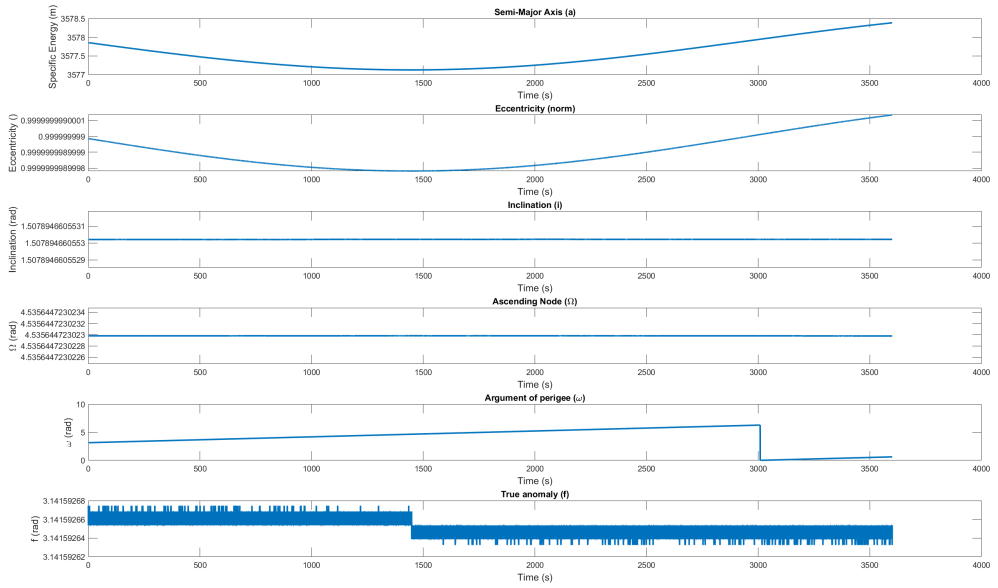


Figure D.14: Classical orbital elements from agent 13

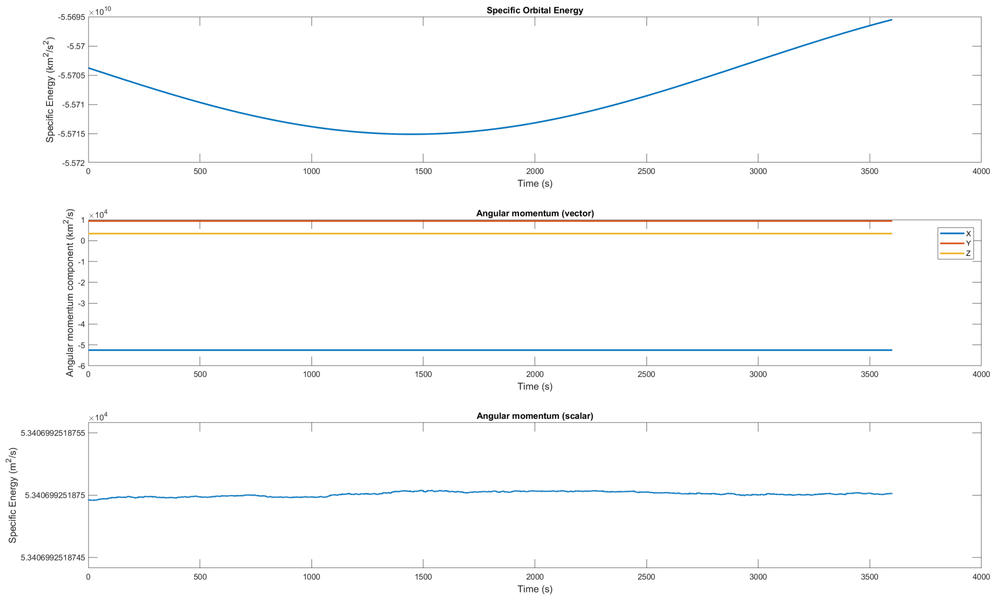


Figure D.15: Angular momentum agent 13

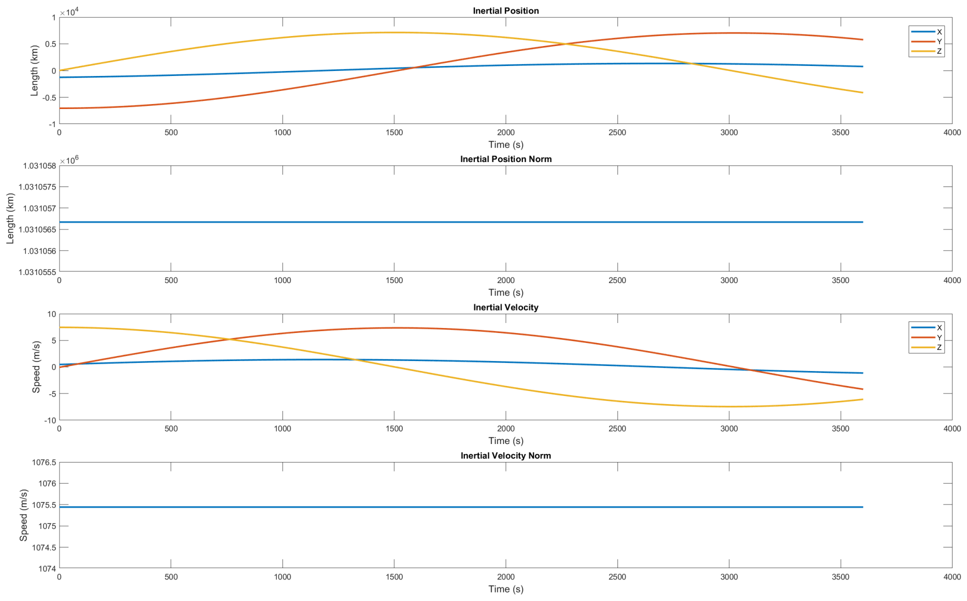


Figure D.16: Inertial position and velocity from agent 13

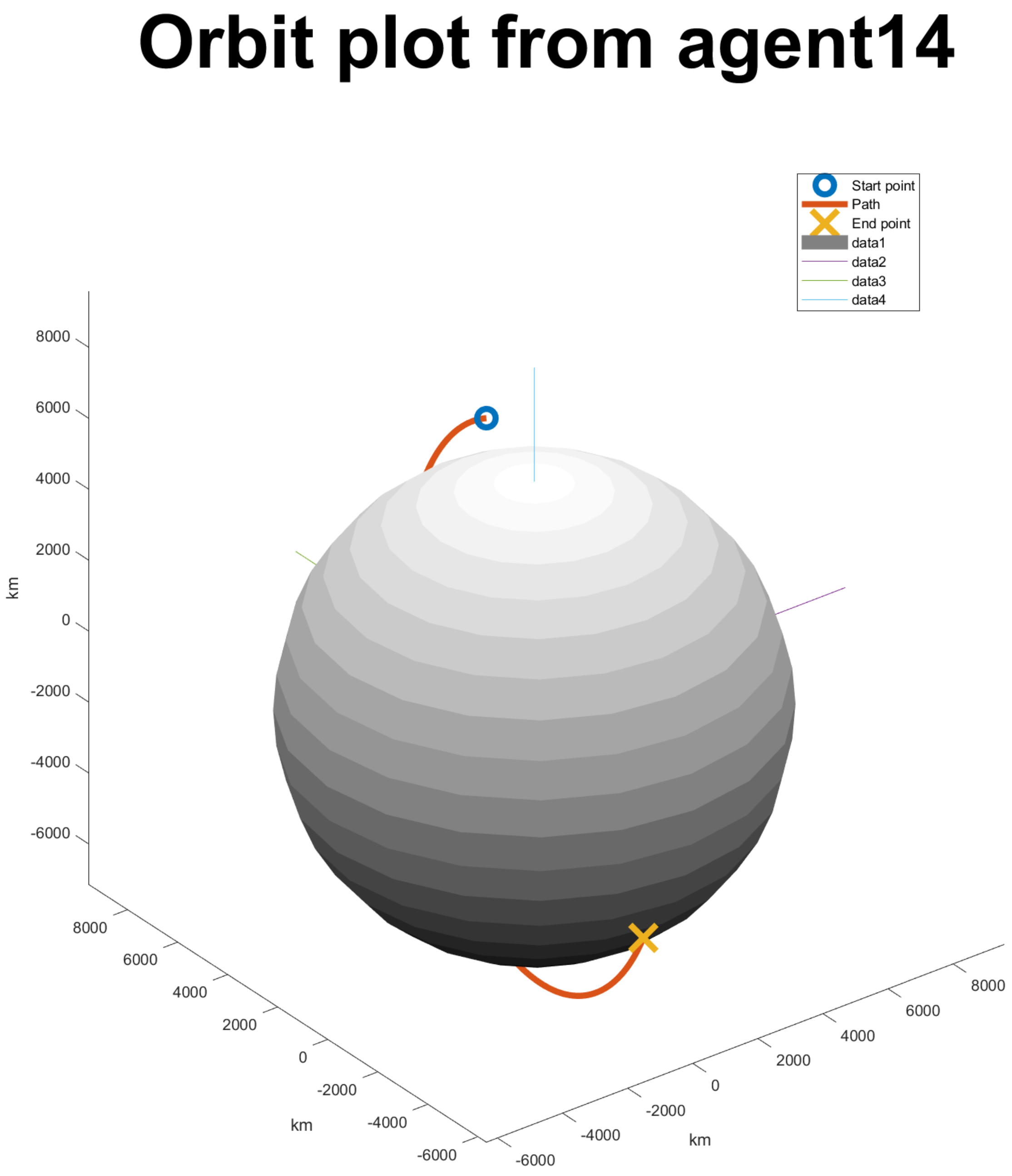


Figure D.17: Orbit plot from agent 14

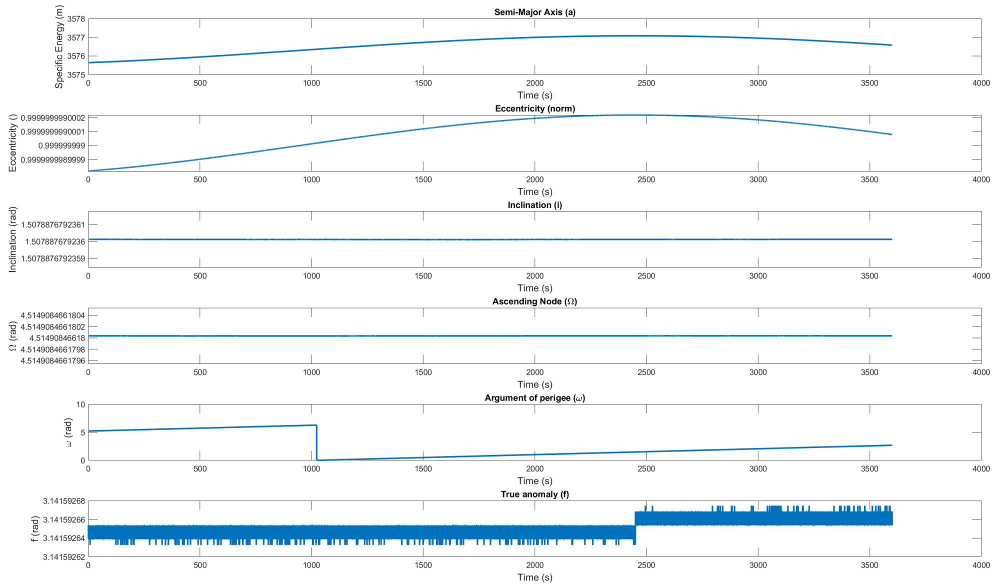


Figure D.18: Classical orbital elements from agent 14

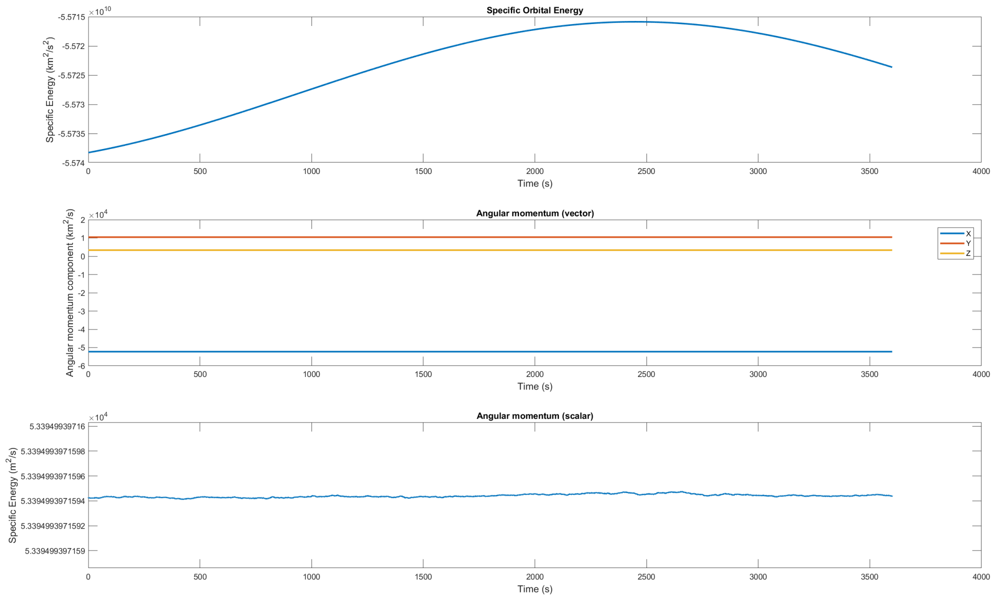


Figure D.19: Angular momentum agent 14

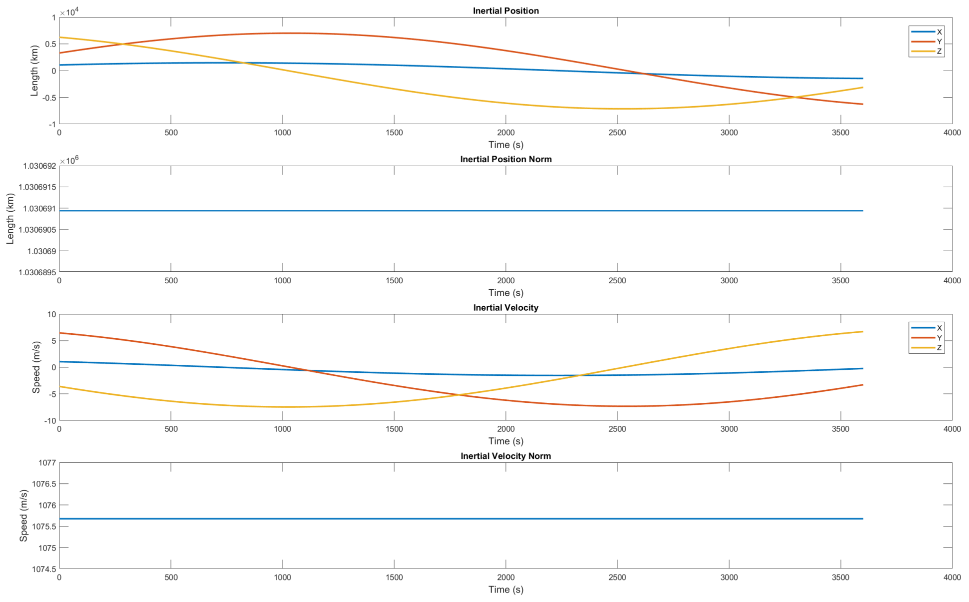


Figure D.20: Inertial position and velocity from agent 14

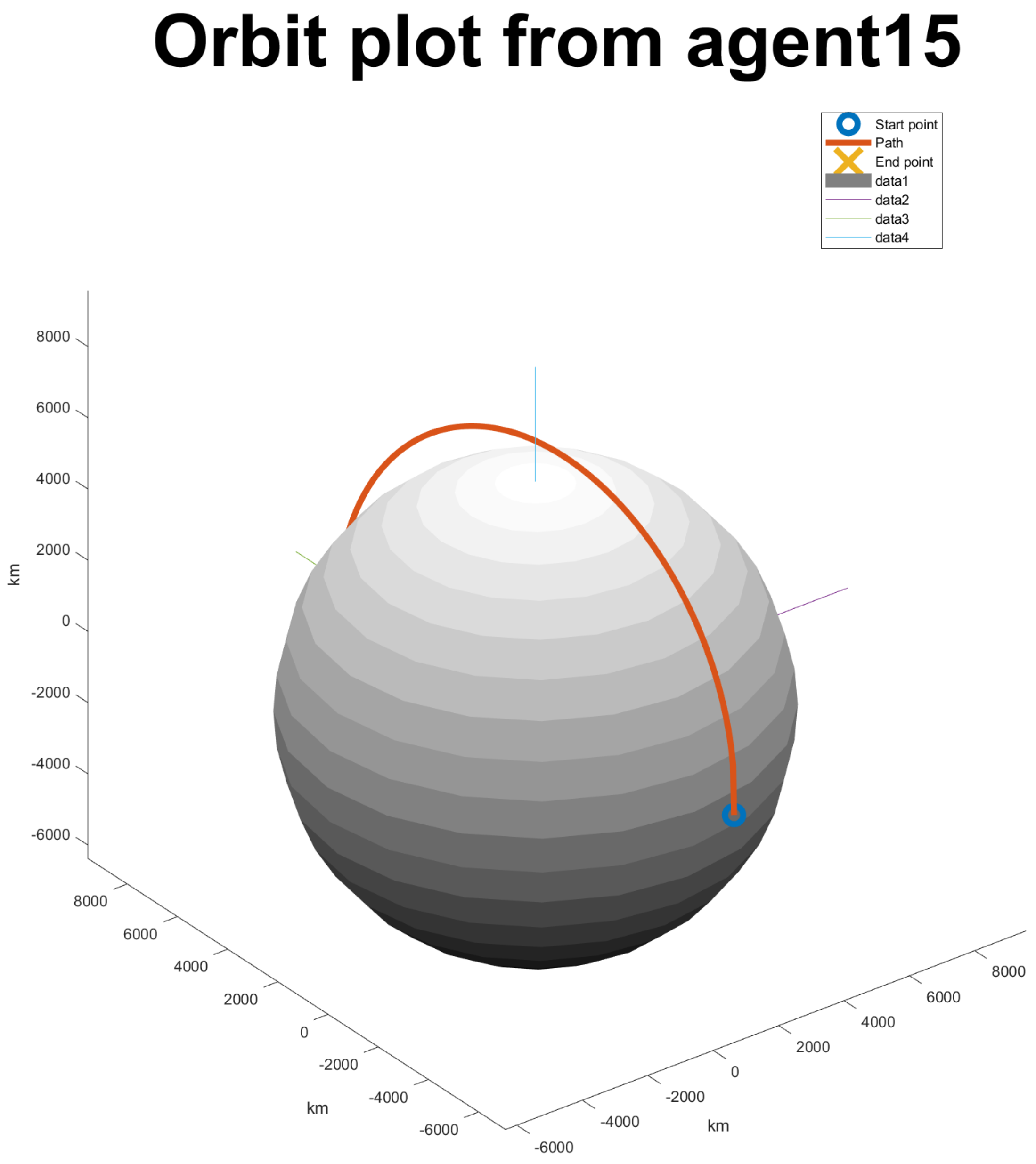


Figure D.21: Orbit plot from agent 15

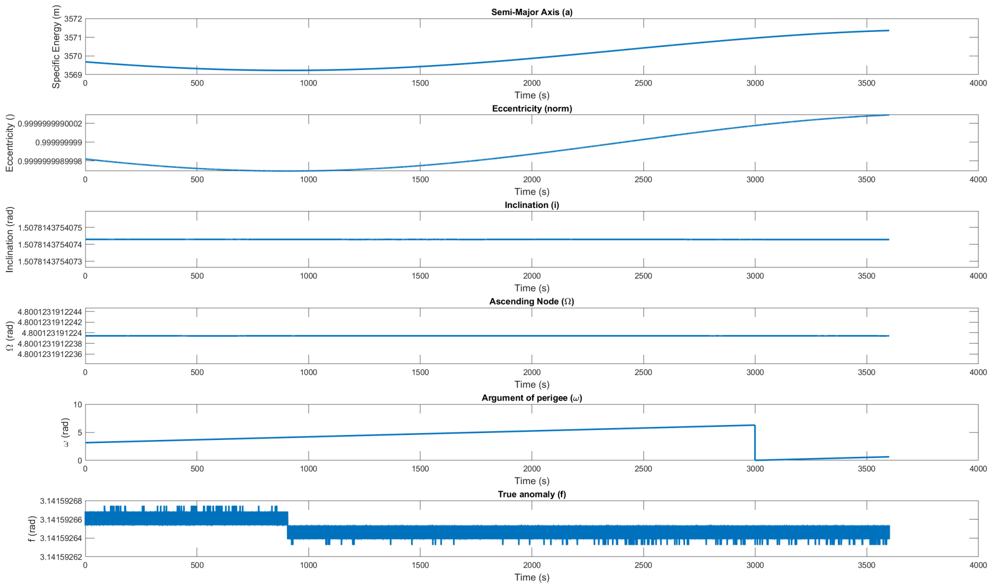


Figure D.22: Classical orbital elements from agent 15

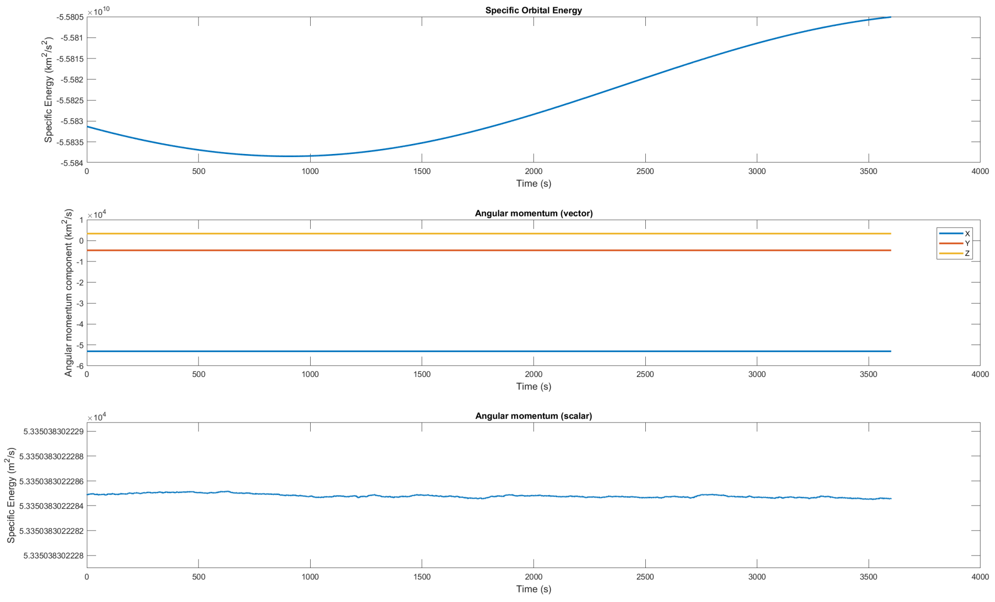


Figure D.23: Angular momentum agent 15

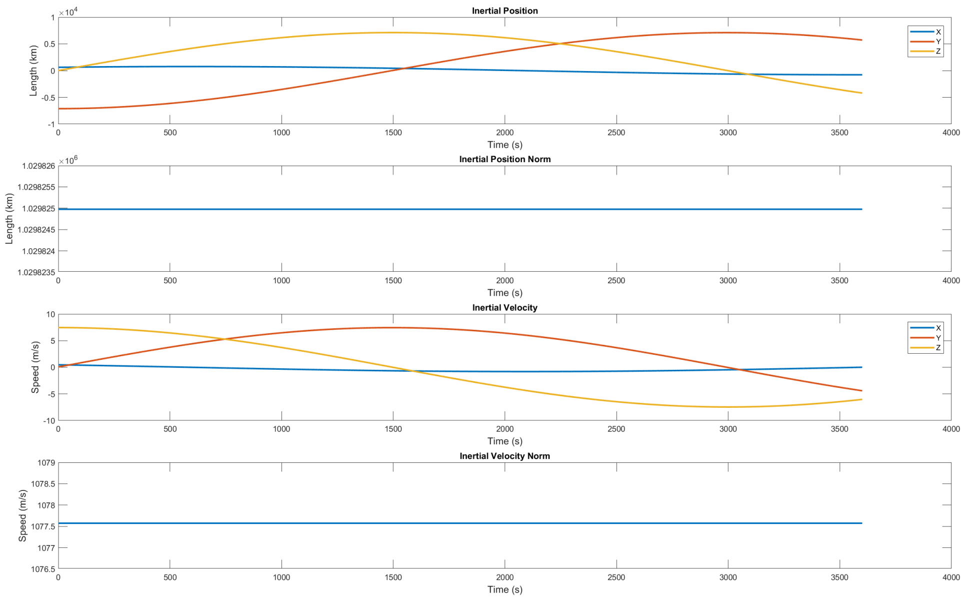


Figure D.24: Inertial position and velocity from agent 15

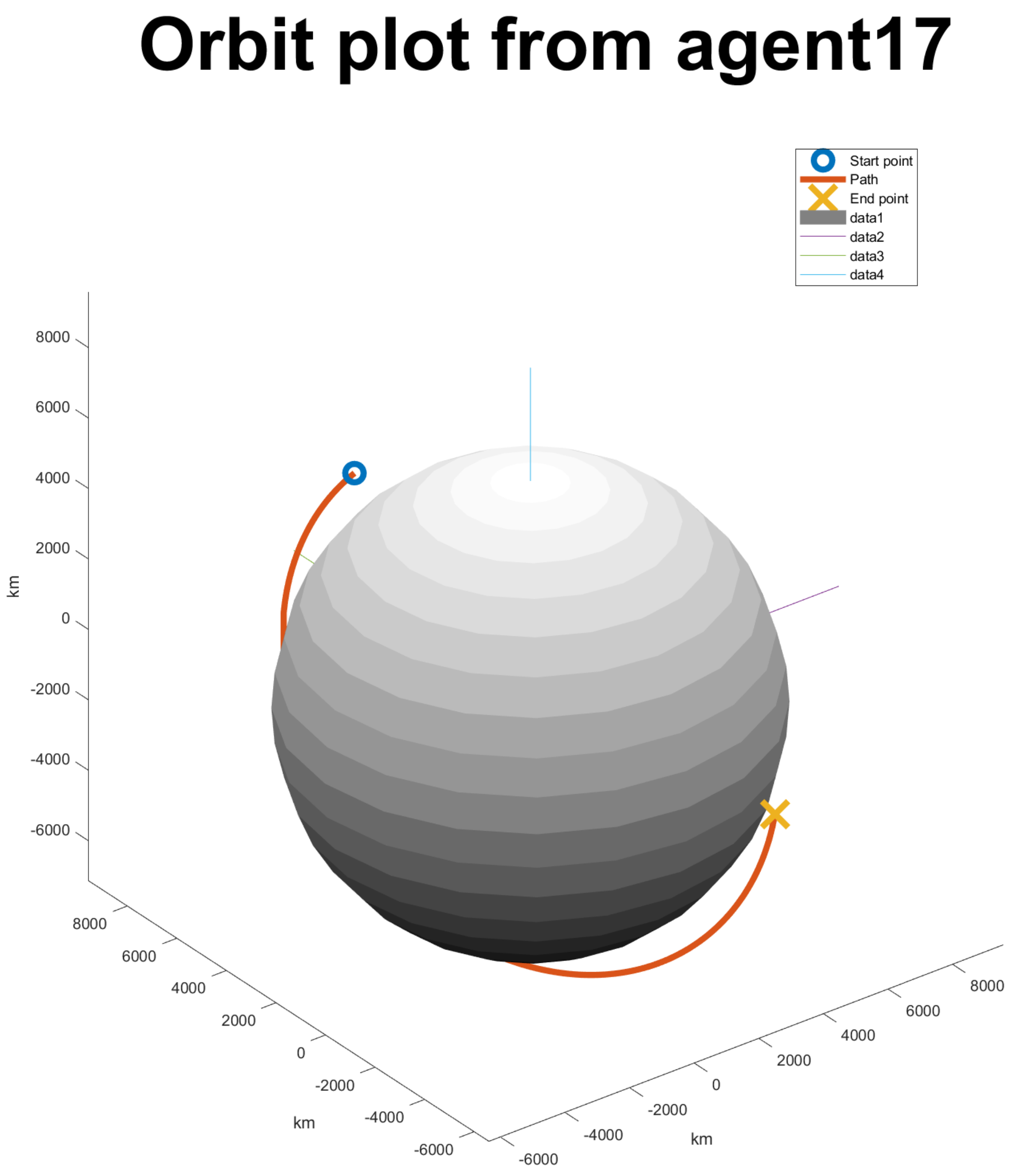


Figure D.25: Orbit plot from agent 17

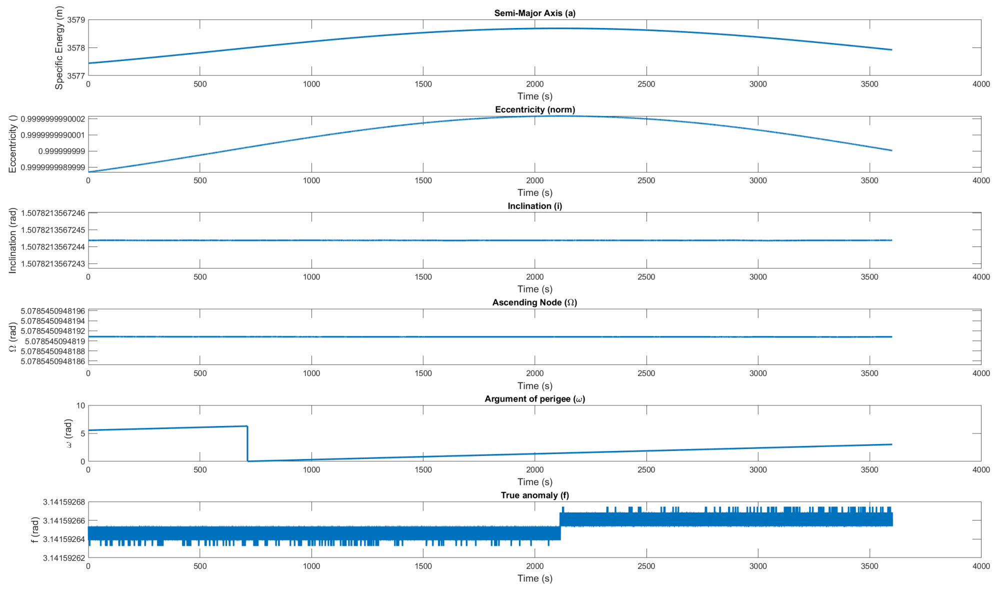


Figure D.26: Classical orbital elements from agent 17

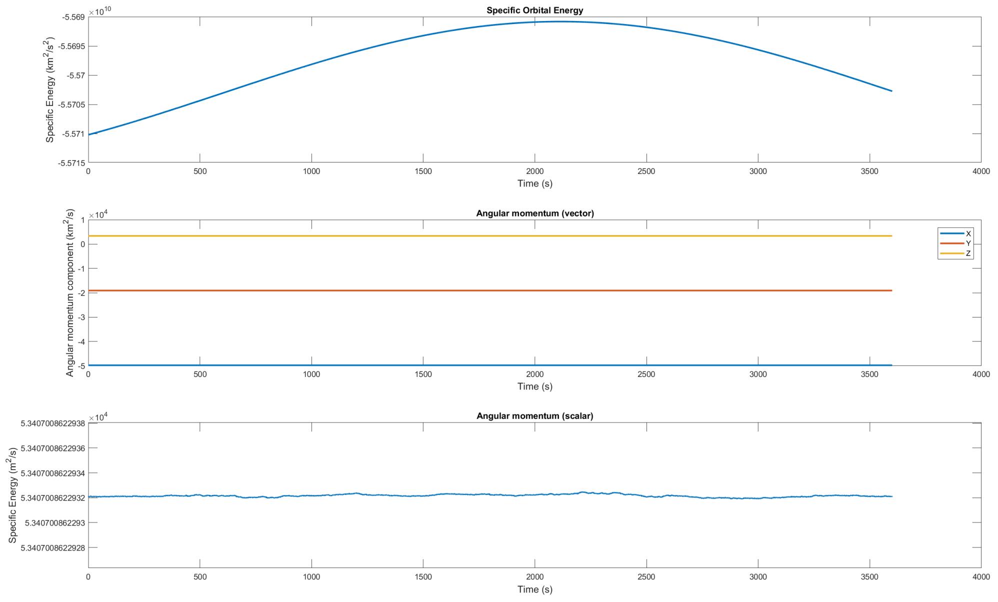


Figure D.27: Angular momentum agent 17

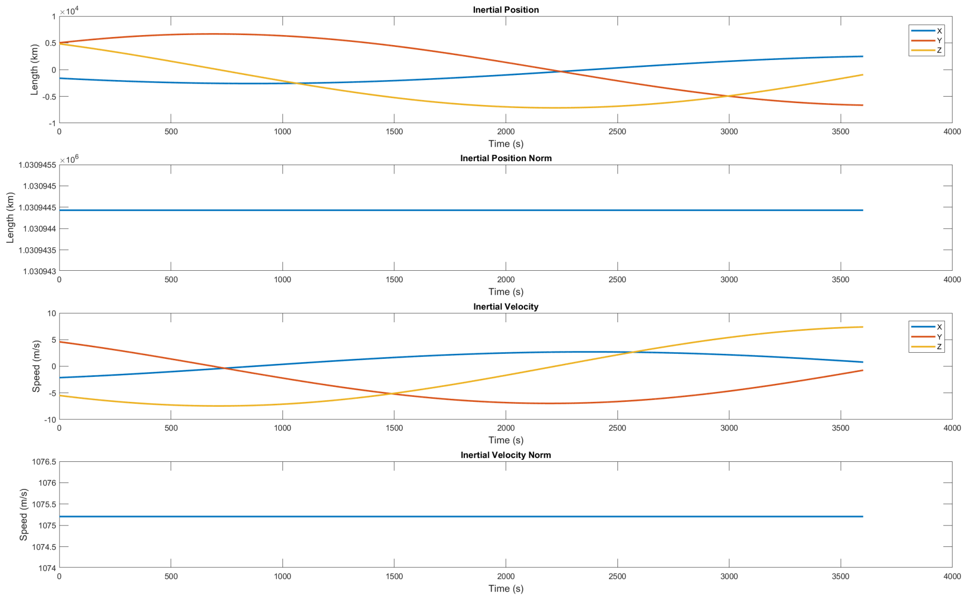


Figure D.28: Inertial position and velocity from agent 17

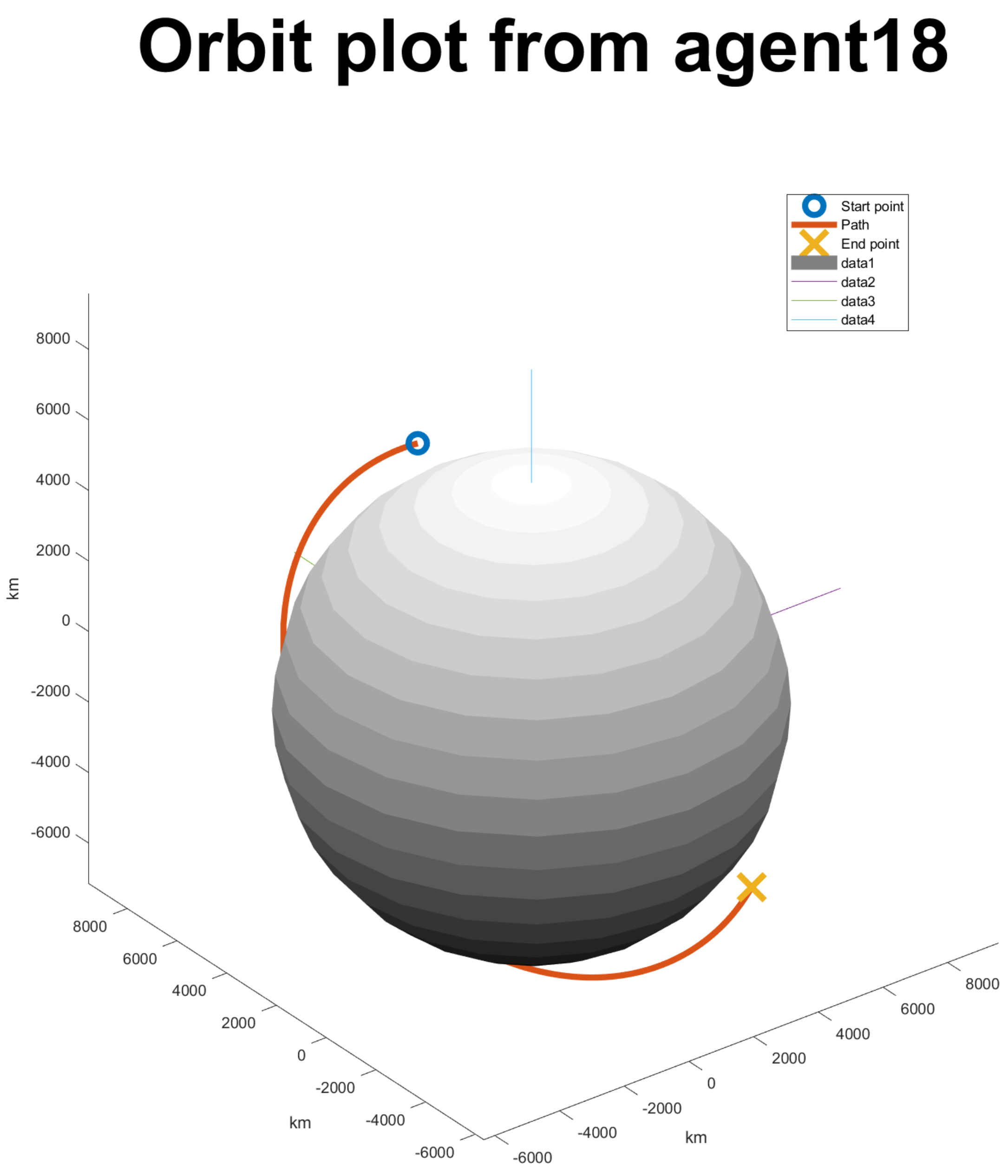


Figure D.29: Orbit plot from agent 18

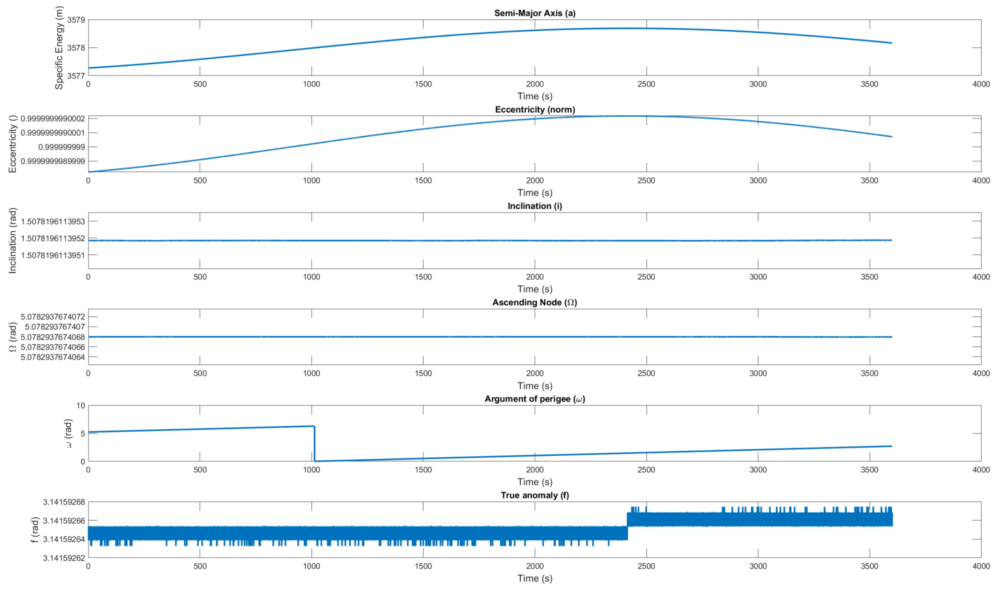


Figure D.30: Classical orbital elements from agent 18

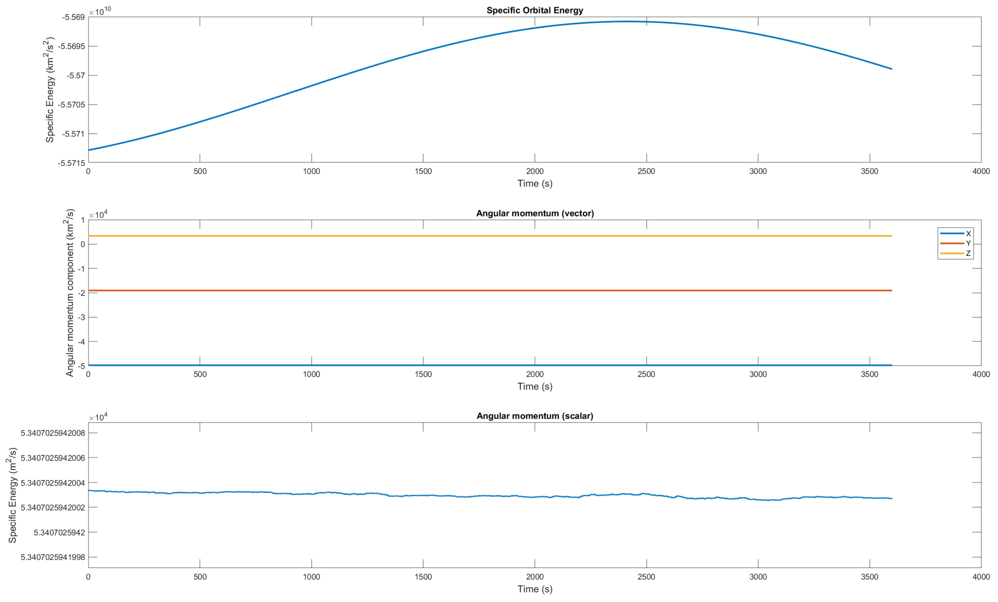


Figure D.31: Angular momentum agent 18

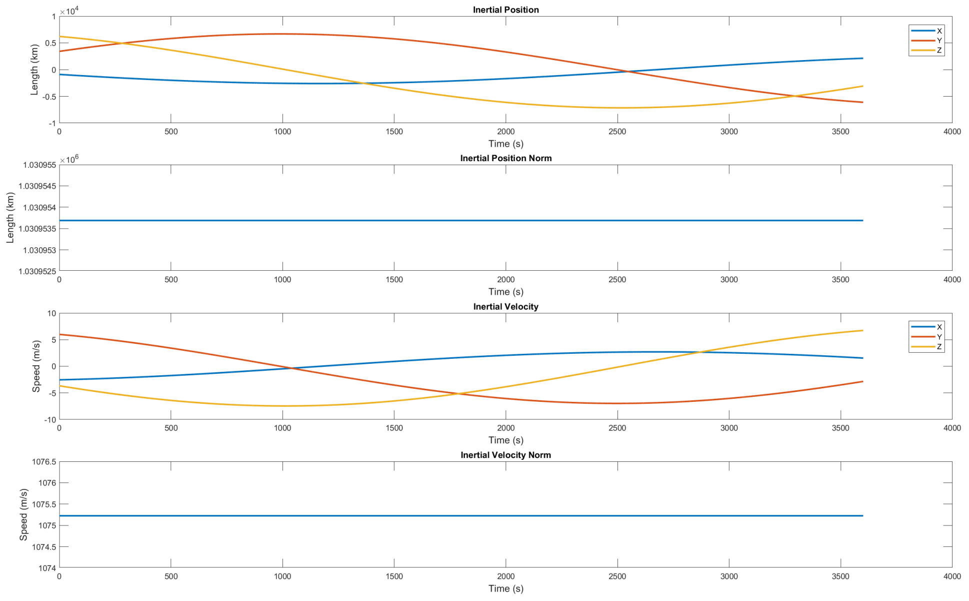


Figure D.32: Inertial position and velocity from agent 18

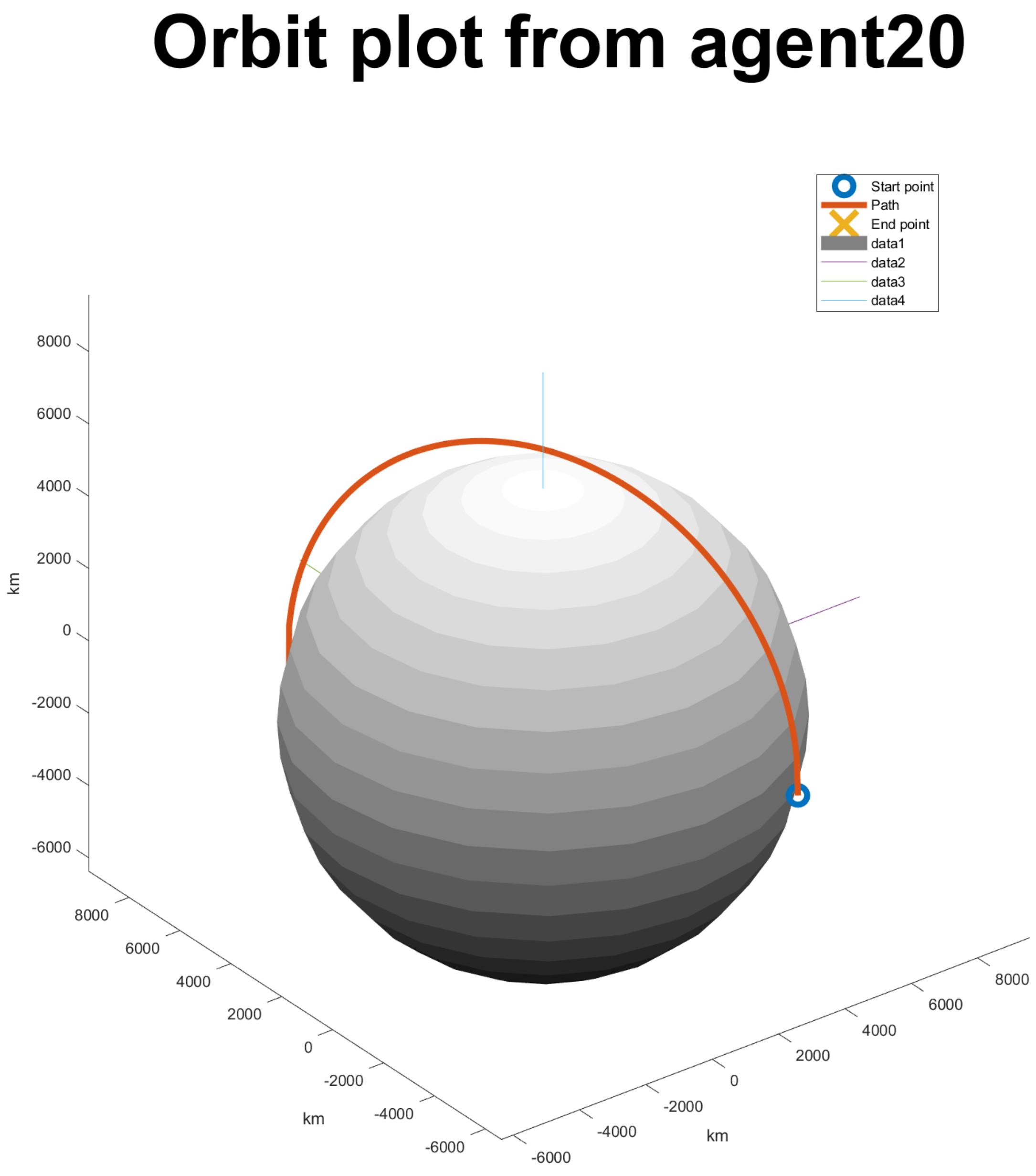


Figure D.33: Orbit plot from agent 20

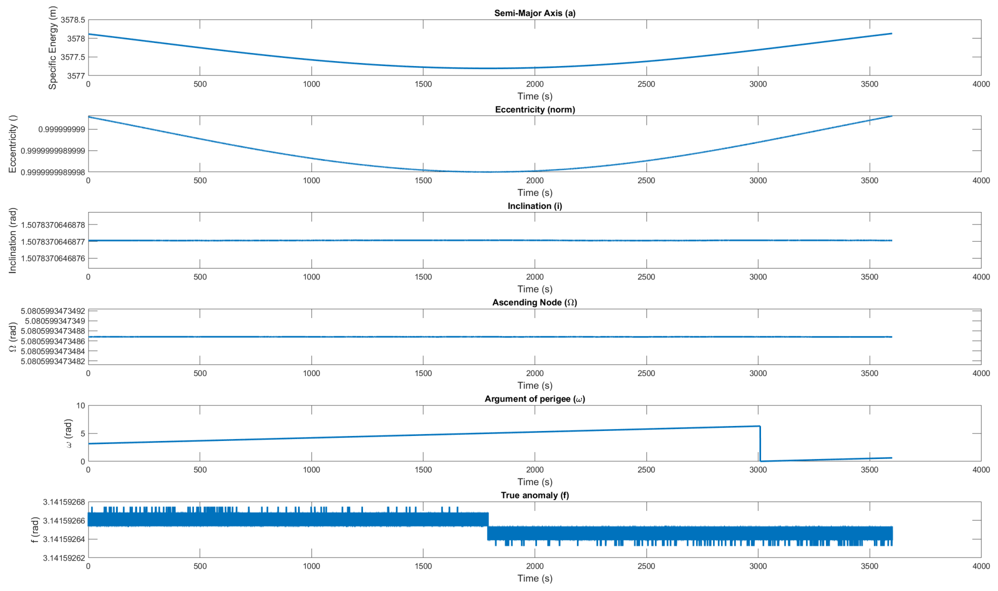


Figure D.34: Classical orbital elements from agent 20

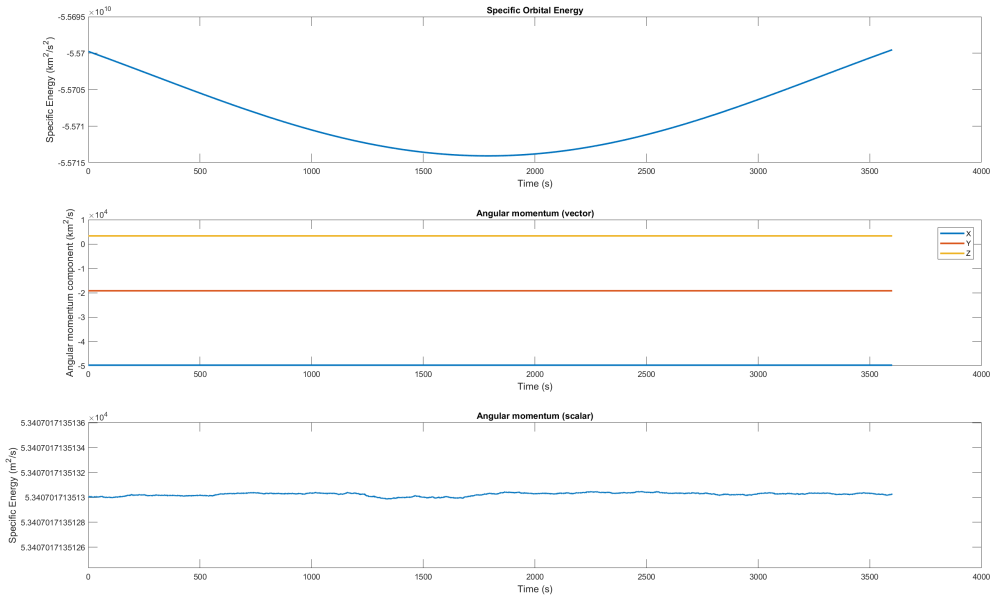


Figure D.35: Angular momentum agent 20

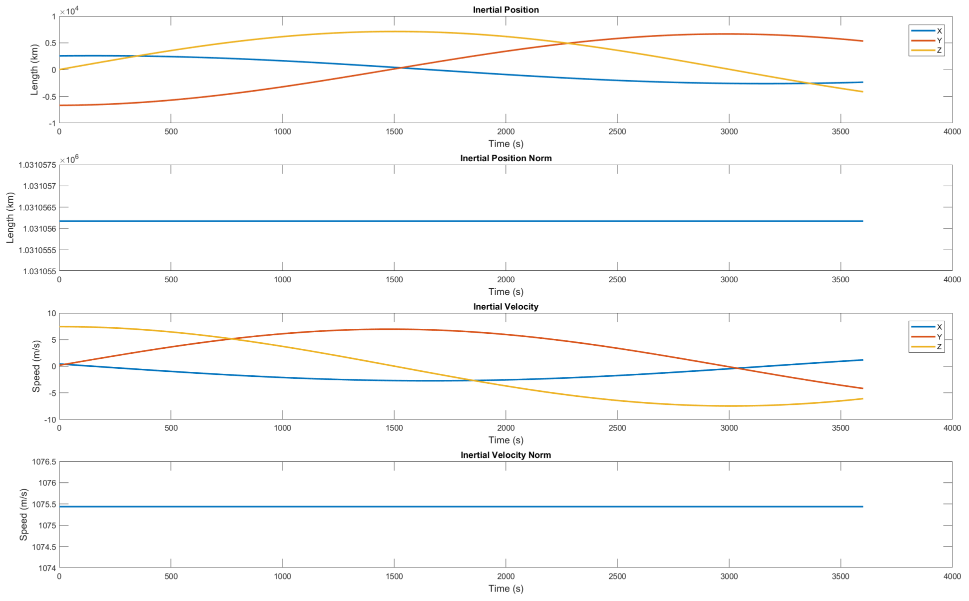


Figure D.36: Inertial position and velocity from agent 20

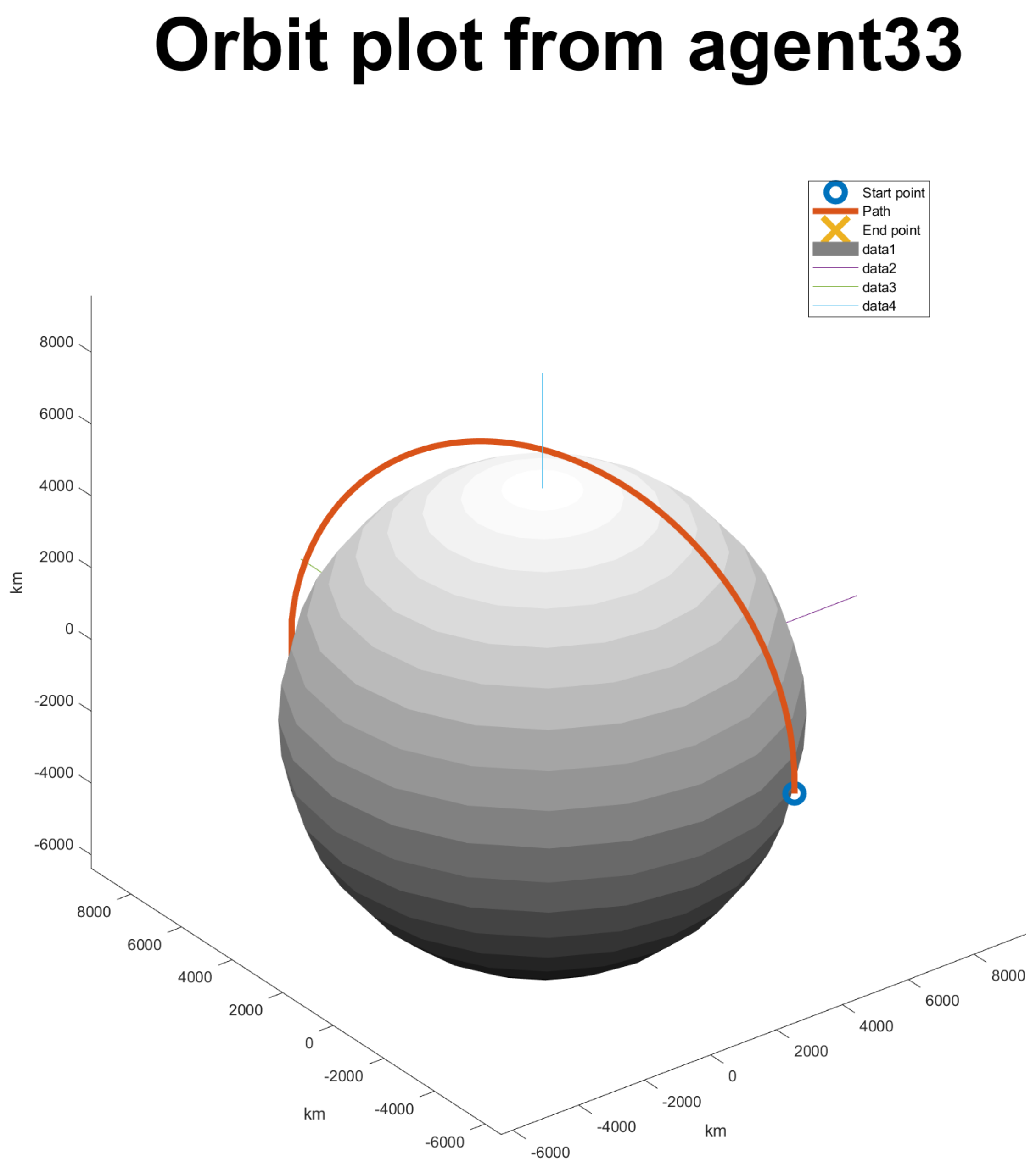


Figure D.37: Orbit plot from agent 33

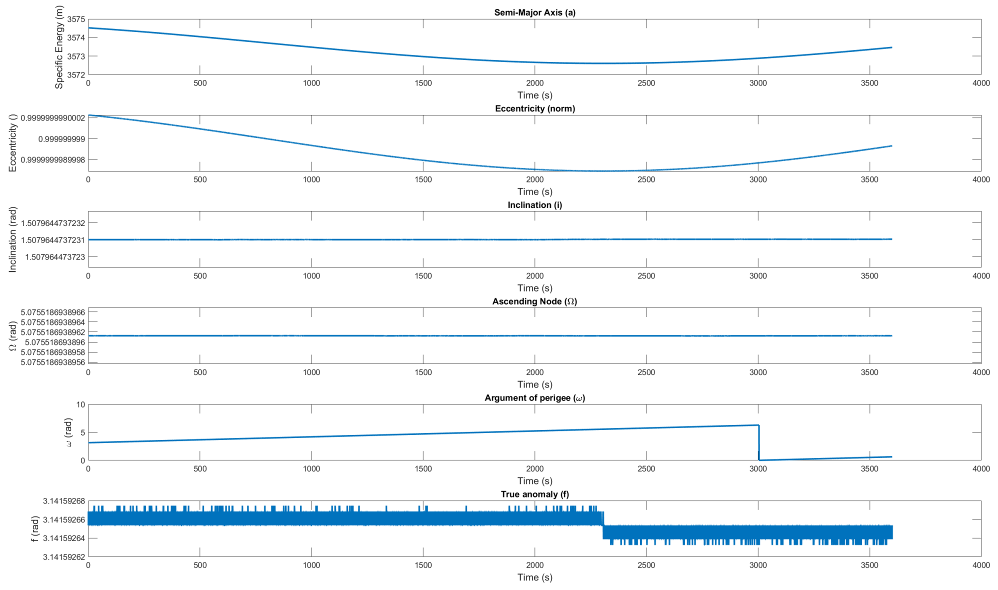


Figure D.38: Classical orbital elements from agent 33

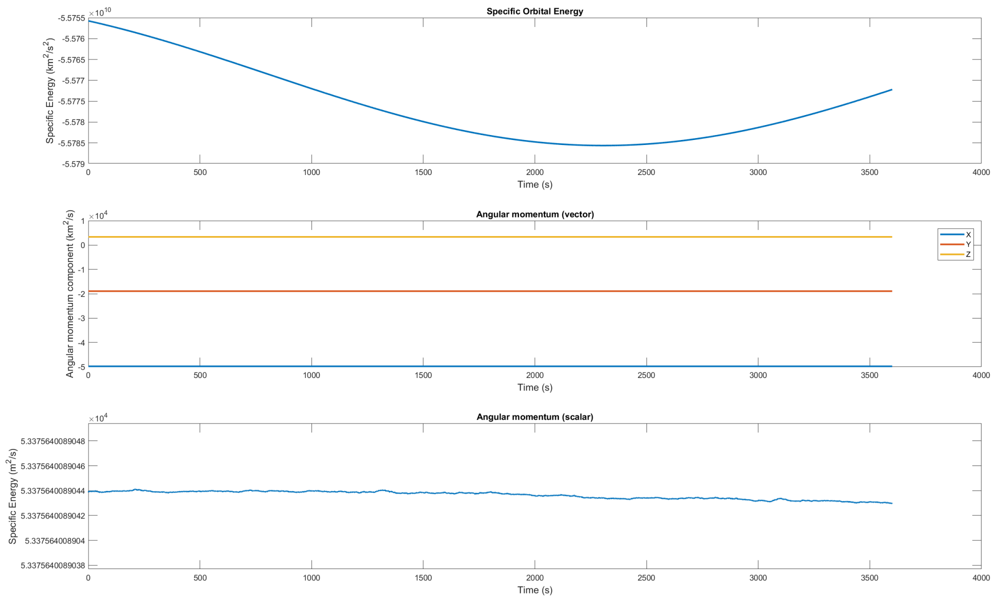


Figure D.39: Angular momentum agent 33

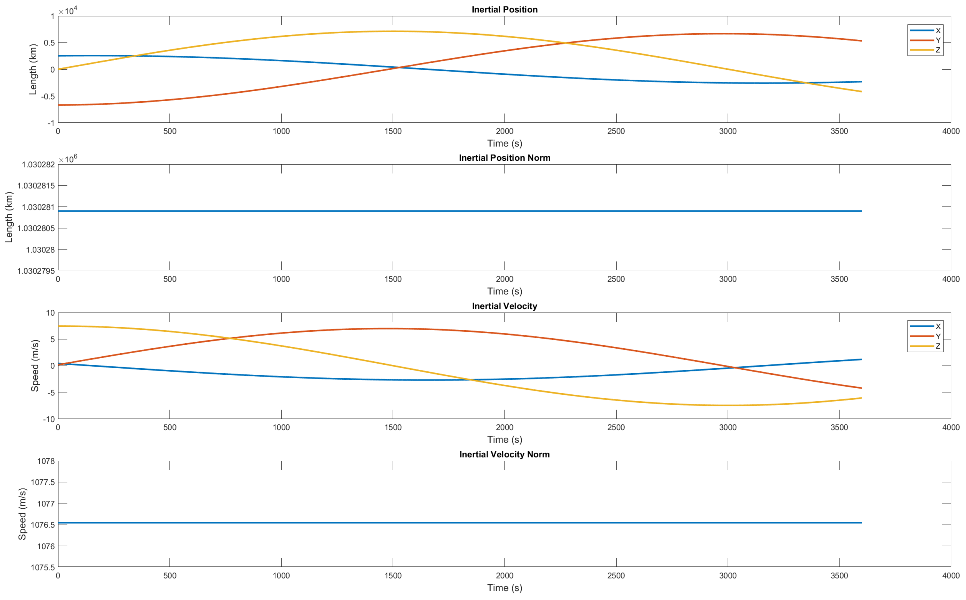


Figure D.40: Inertial position and velocity from agent 33

D.1.2 Temperature results

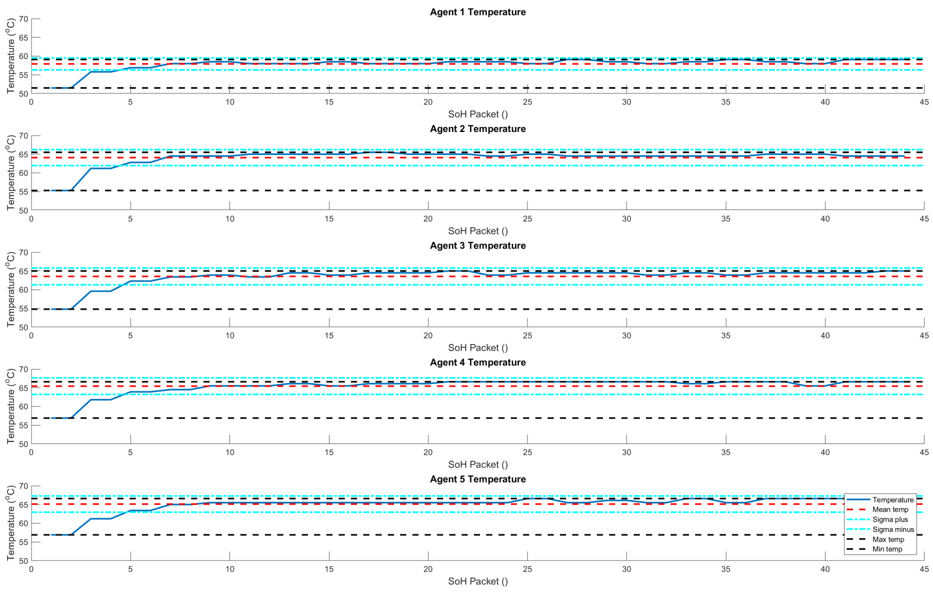


Figure D.41: 1-hour measured temperature agents 1-5

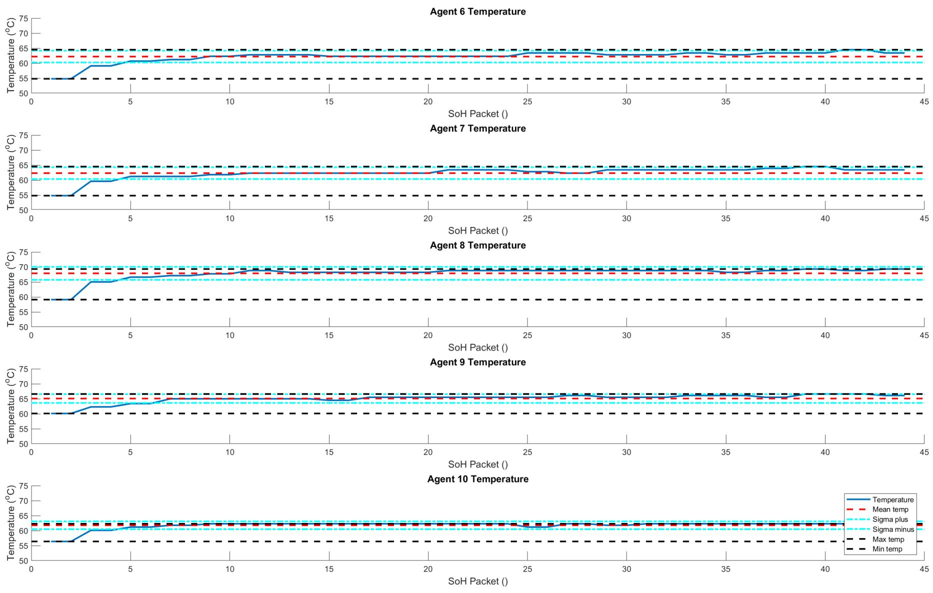


Figure D.42: 1-hour measured temperature agents 6-10

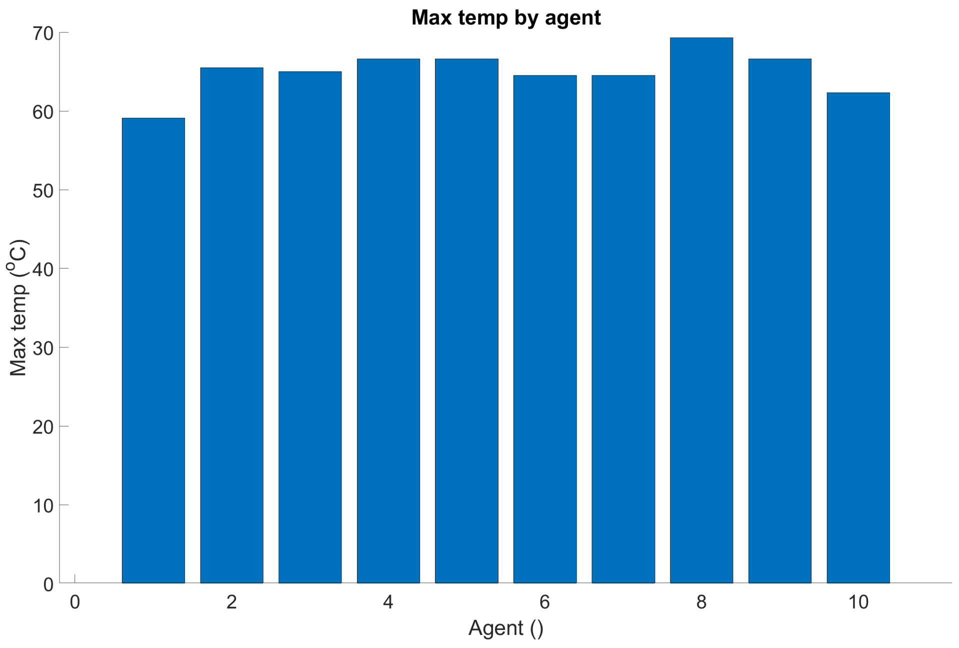


Figure D.43: 1-hour maximum temperature by agent

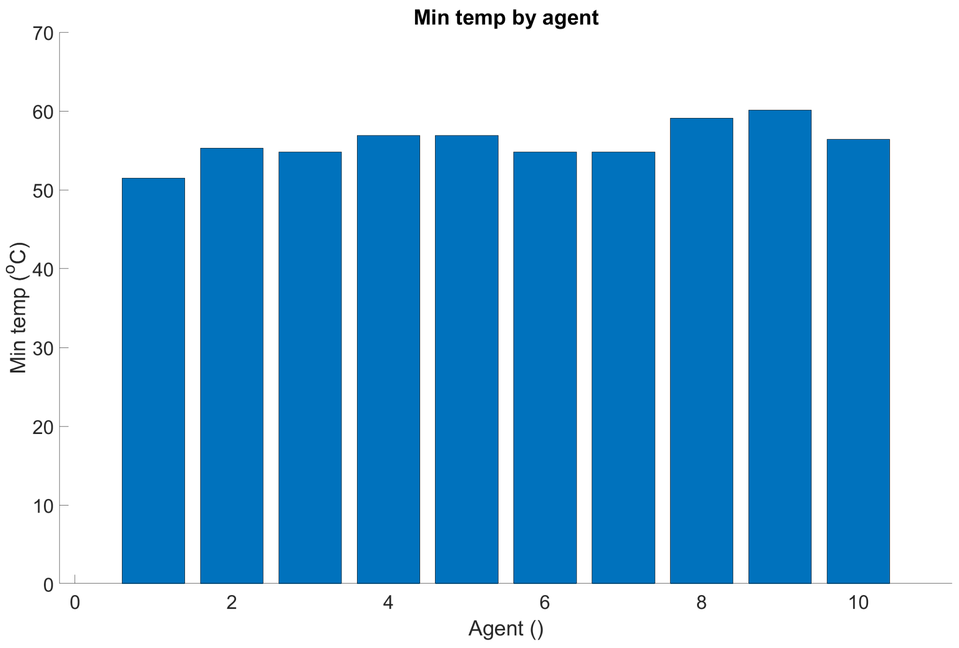


Figure D.44: 1-hour minimum temperature by agent

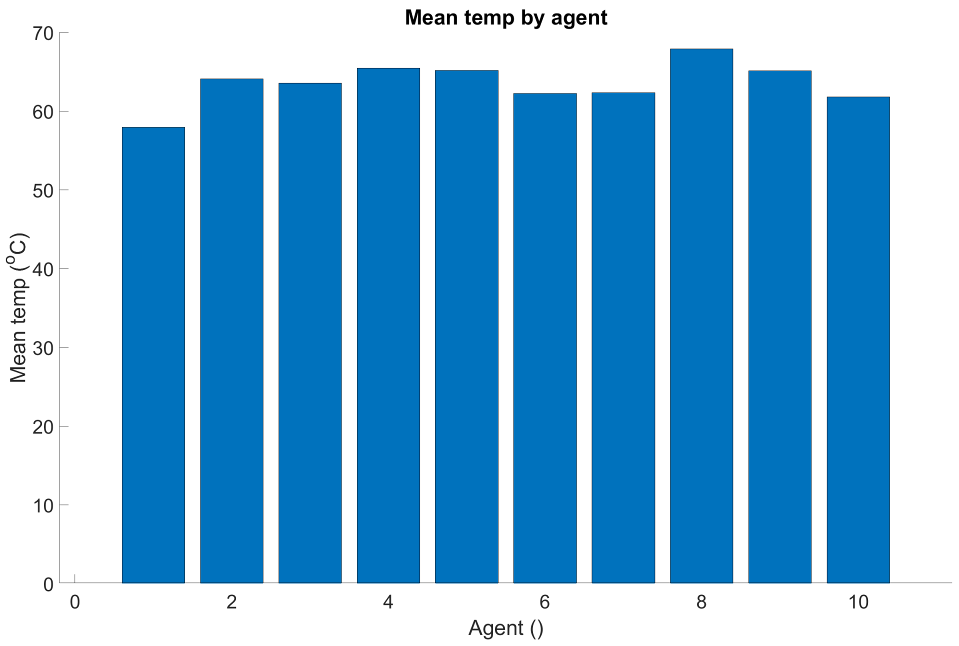


Figure D.45: 1-hour average temperature by agent

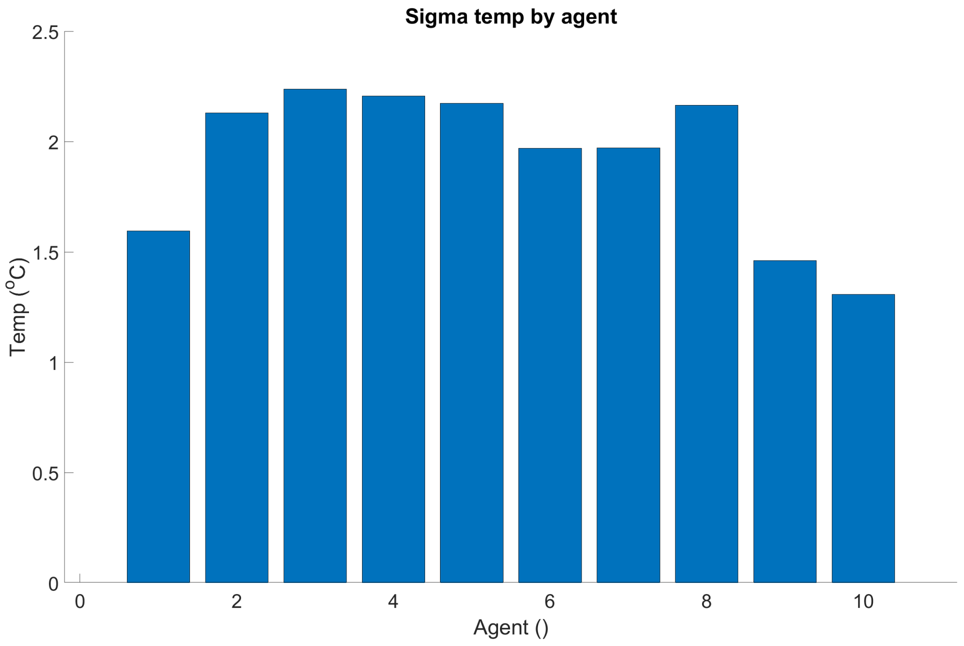


Figure D.46: 1-hour temperature standard deviation by agent

D.1.3 CPU Usage by agent

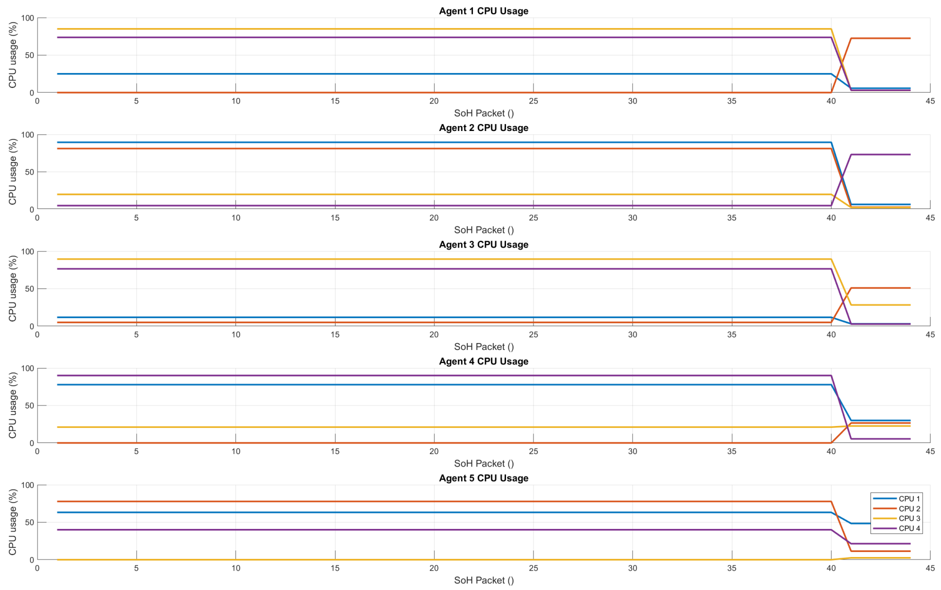


Figure D.47: 1-hour measured CPU usage agents 1-5

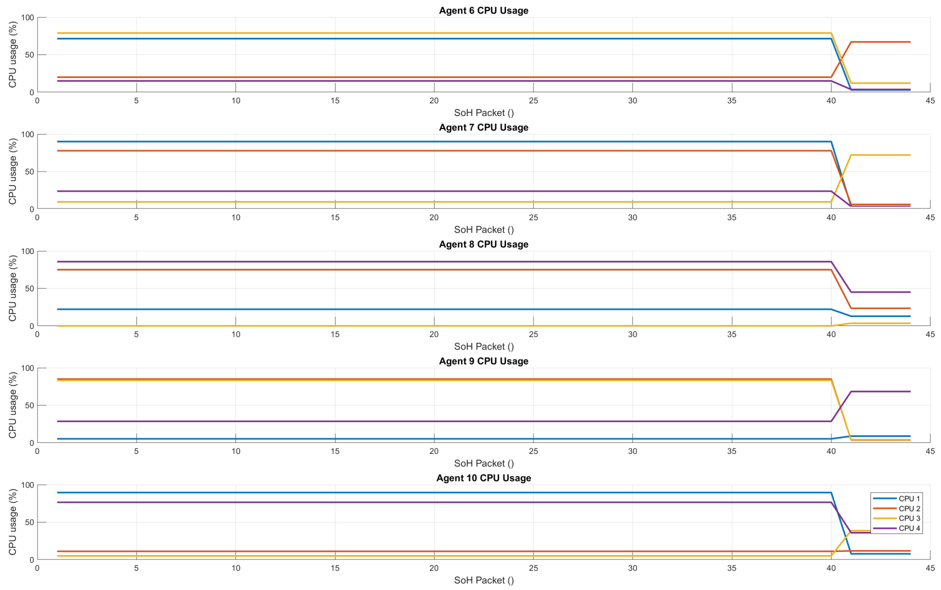


Figure D.48: 1-hour measured CPU usage agents 6-10

D.1.4 HWIL Data

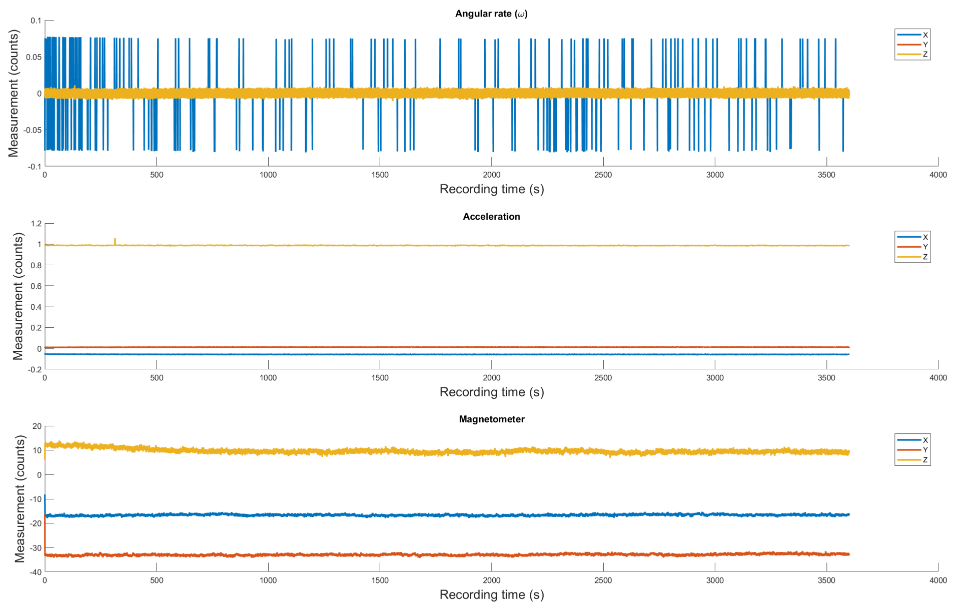


Figure D.49: 1-hour recorded HWIL gyro, accelerometer, and magnetometer

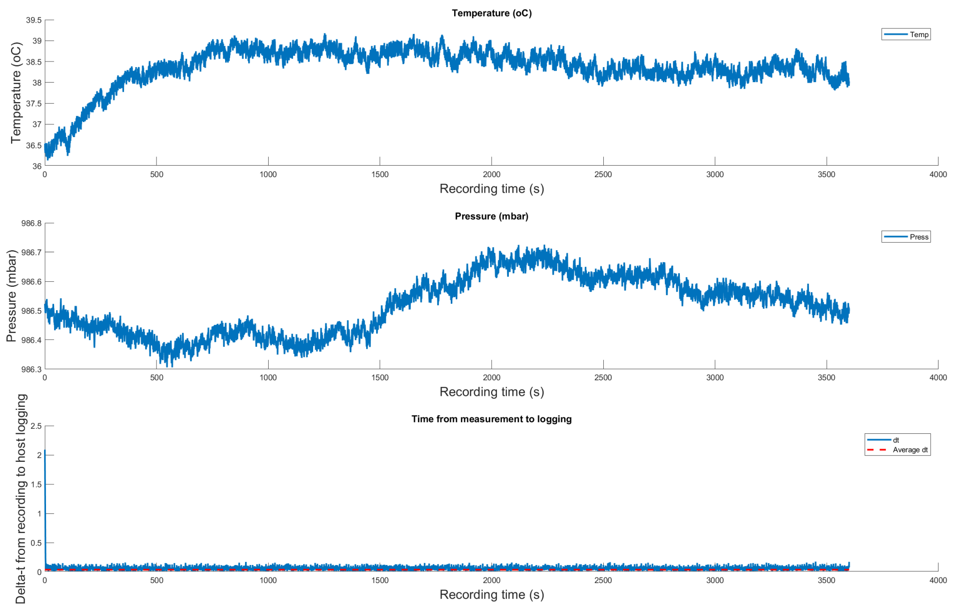


Figure D.50: 1-hour recorded temperature, pressure, and calculated packet Δt

D.1.5 Guidance Algorithm Results

The full set of results from the 4-hour COSMoS HWIL and guidance test is shown below:

Orbital results

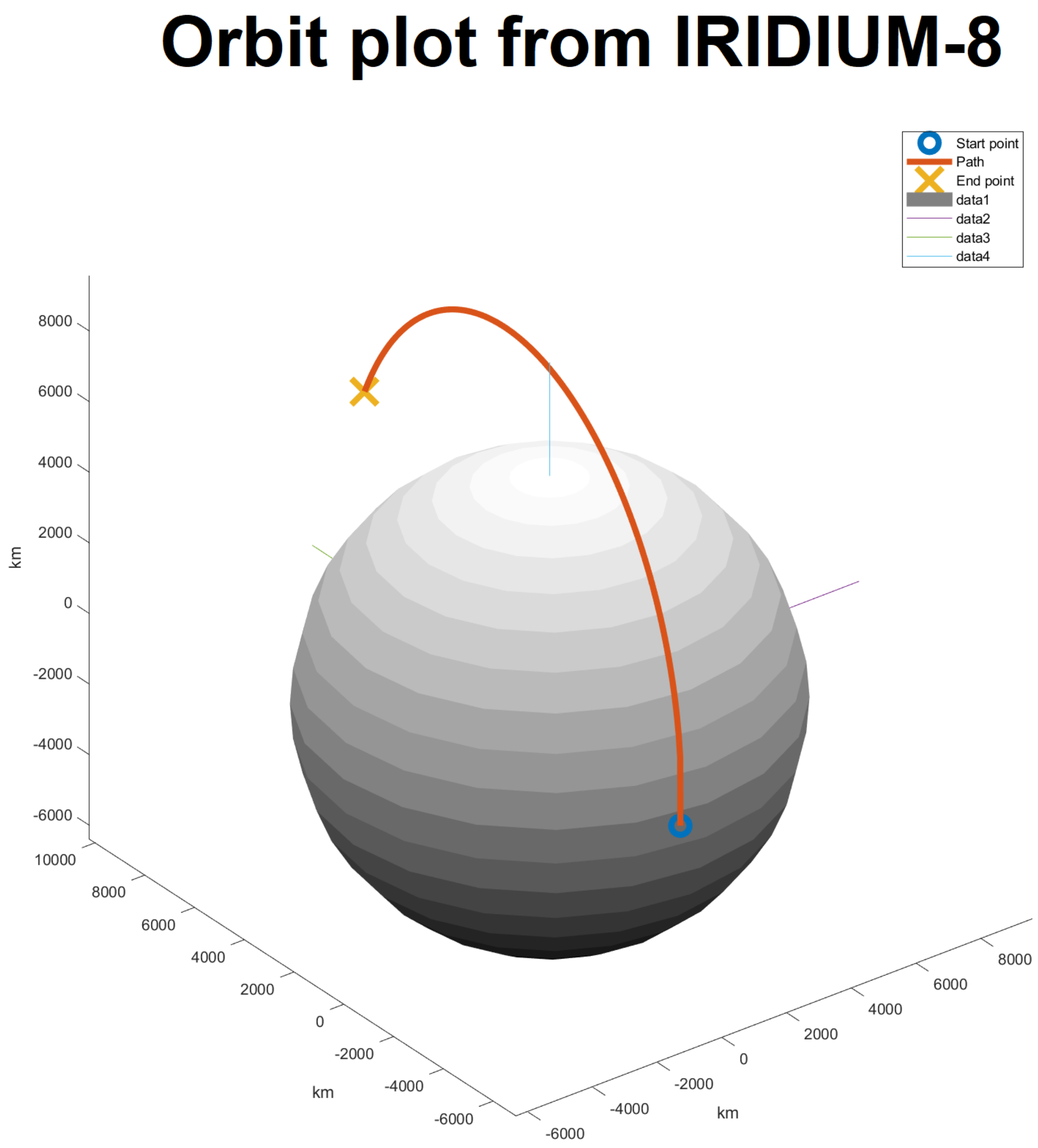


Figure D.51: Orbit plot from agent 10

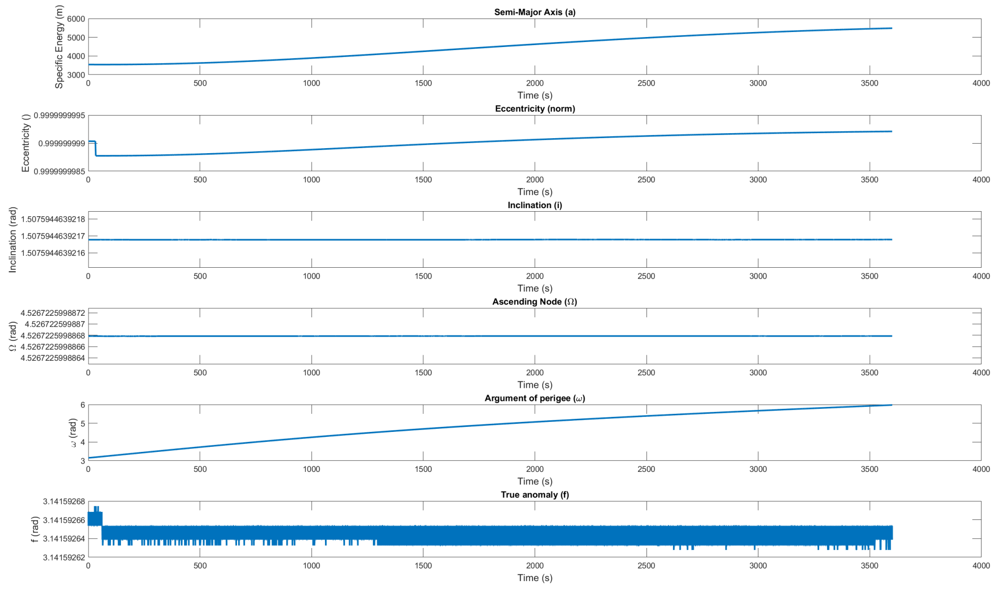


Figure D.52: Classical orbital elements from agent 10

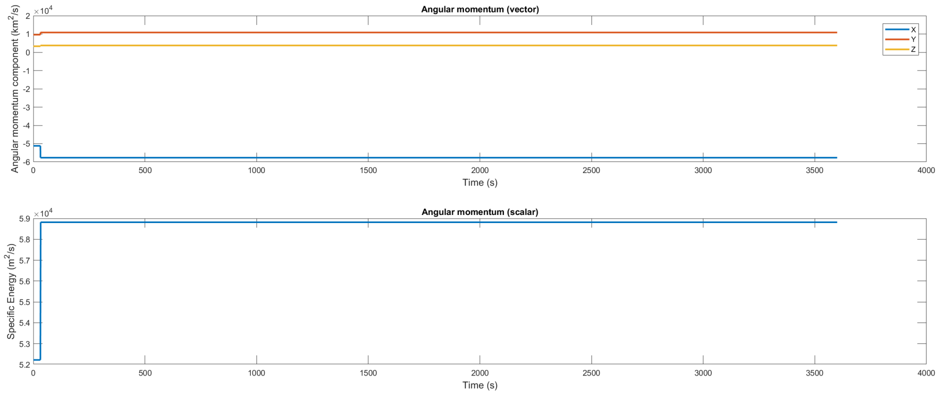


Figure D.53: Angular momentum agent 10

Temperature and CPU results

D.1.6 HWIL Data

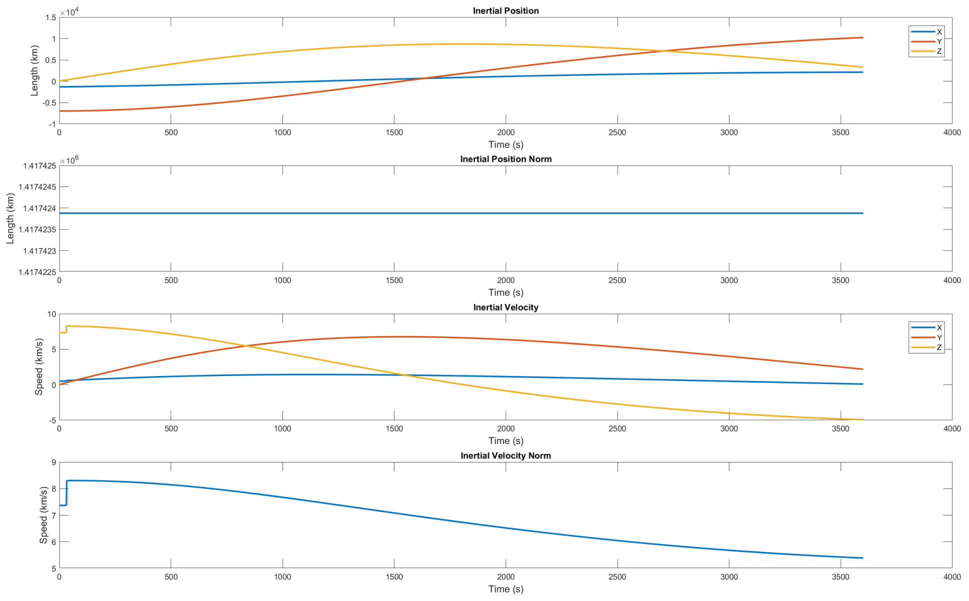


Figure D.54: Inertial position and velocity from agent 10

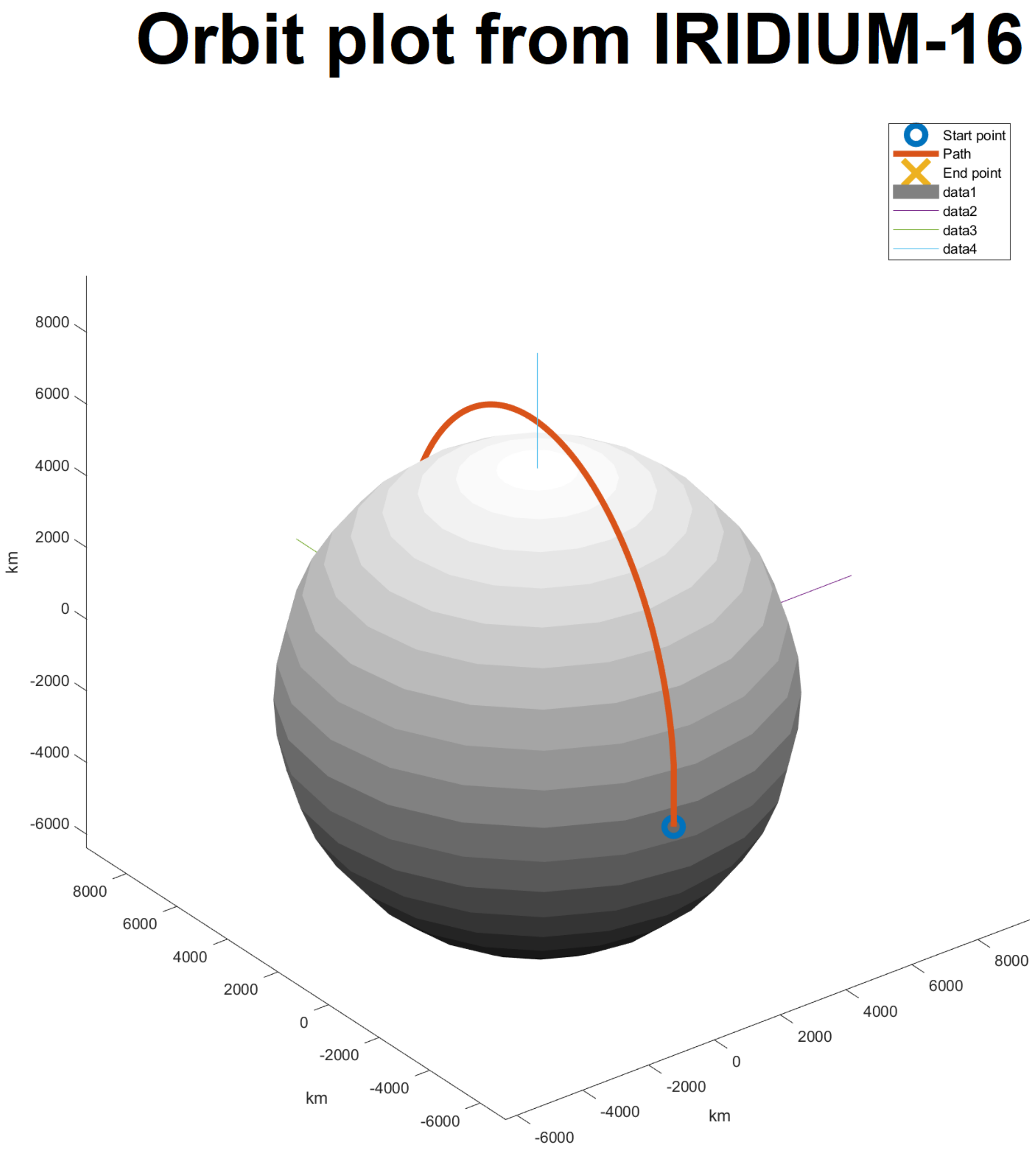


Figure D.55: Orbit plot from agent 33

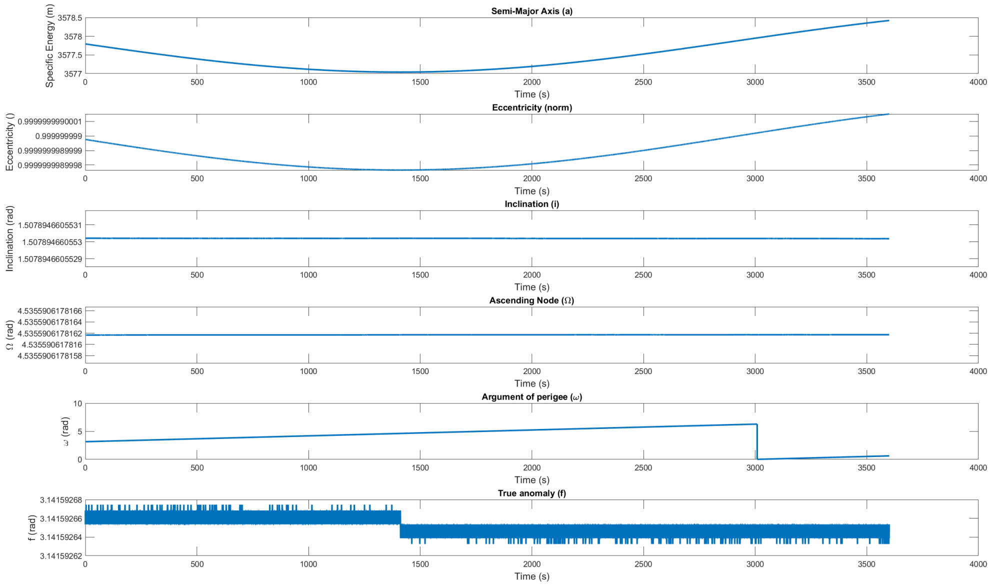


Figure D.56: Classical orbital elements from agent 33

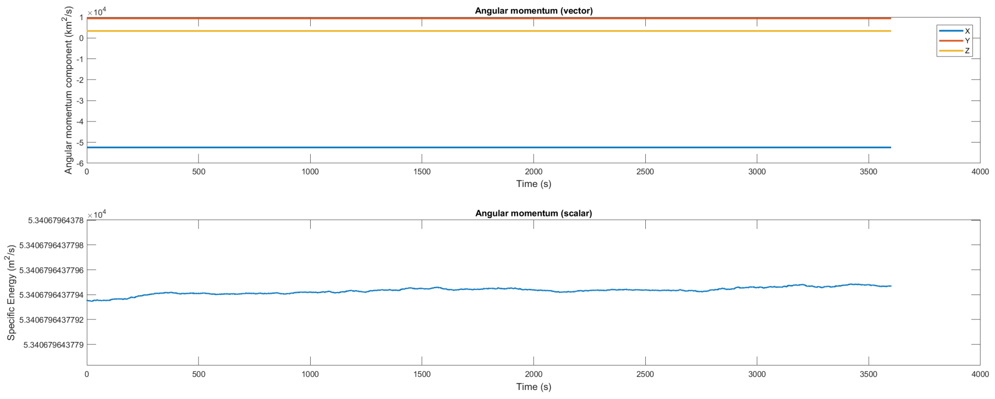


Figure D.57: Angular momentum agent 33

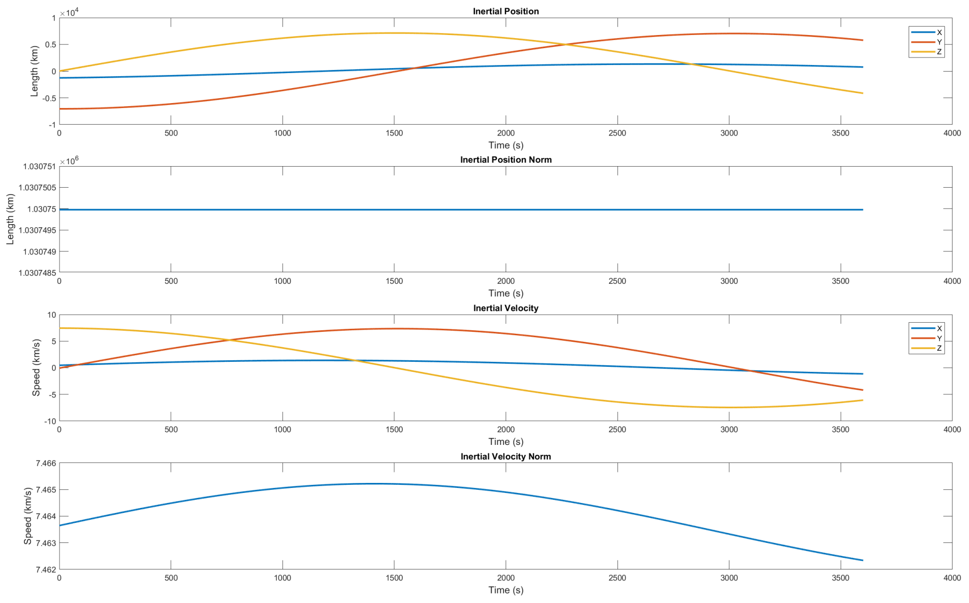


Figure D.58: Inertial position and velocity from agent 33

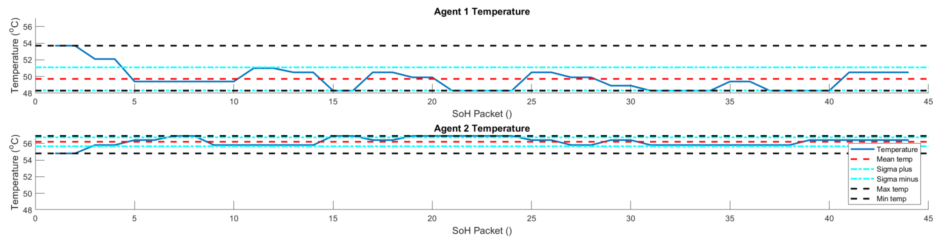


Figure D.59: 1-hour measured temperature agents 1-5

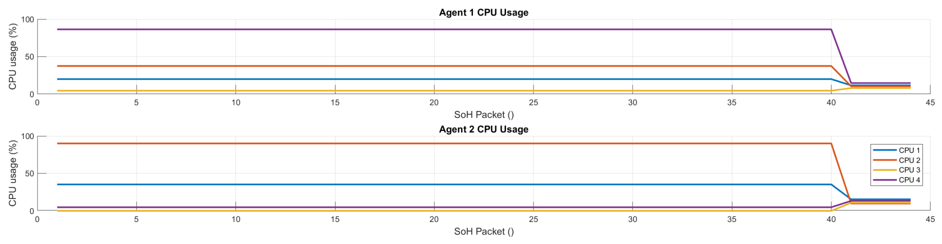


Figure D.60: 1-hour measured CPU usage agents 1-5

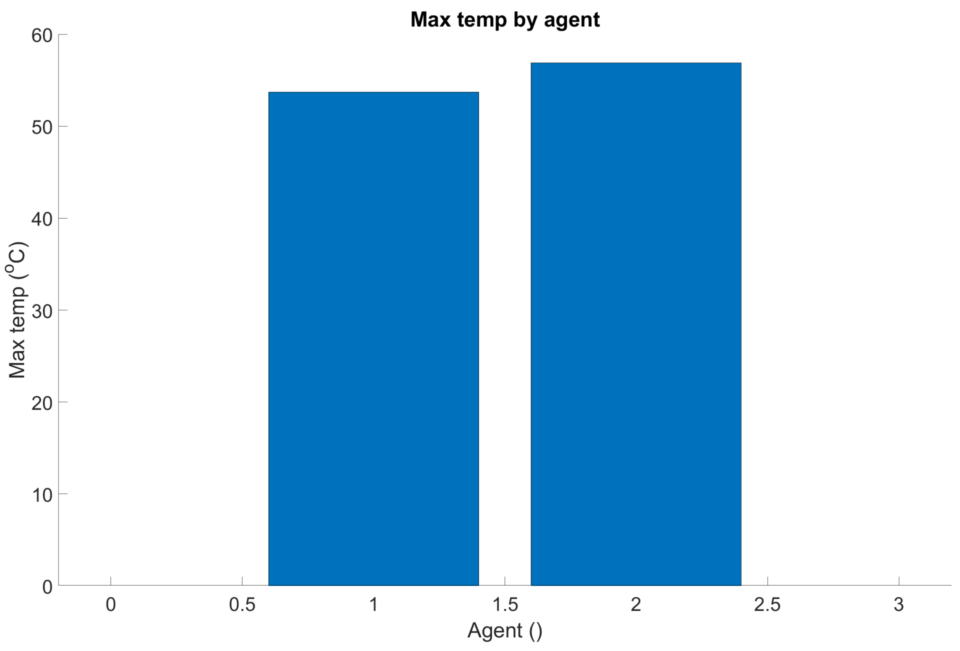


Figure D.61: 1-hour maximum temperature by agent

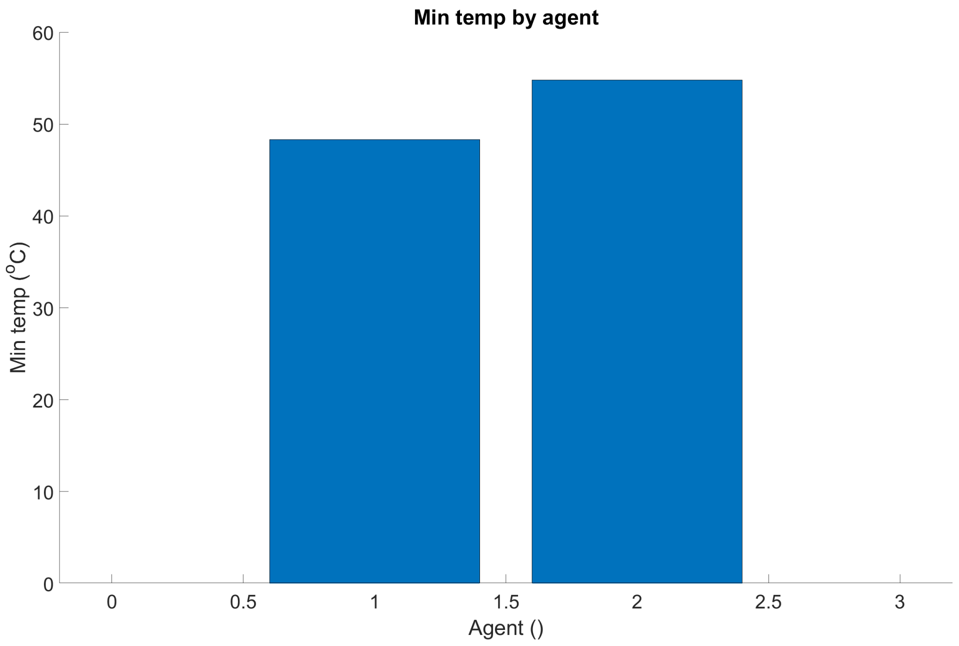


Figure D.62: 1-hour minimum temperature by agent

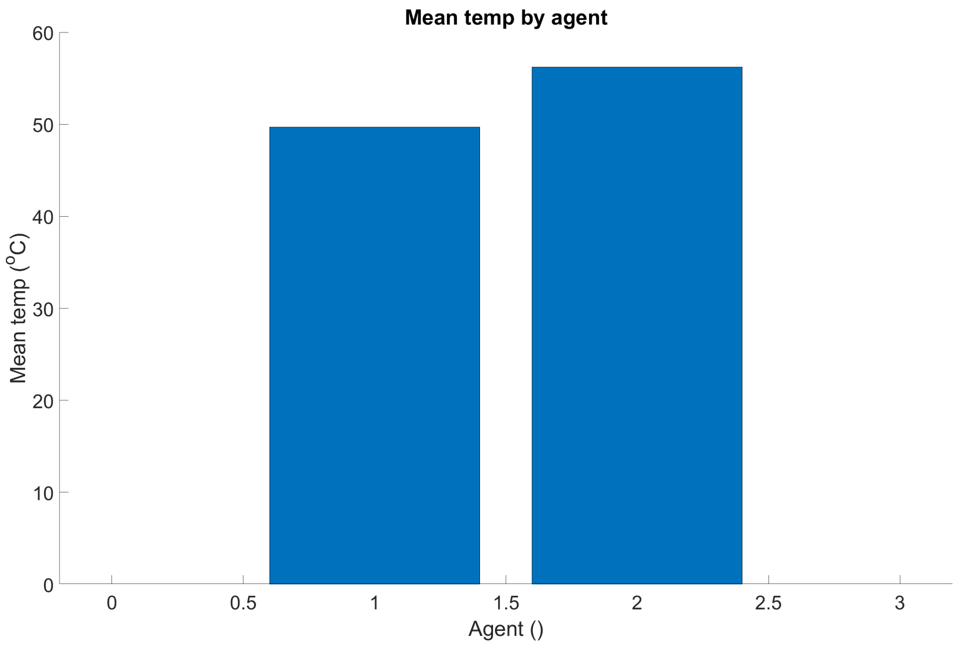


Figure D.63: 1-hour average temperature by agent

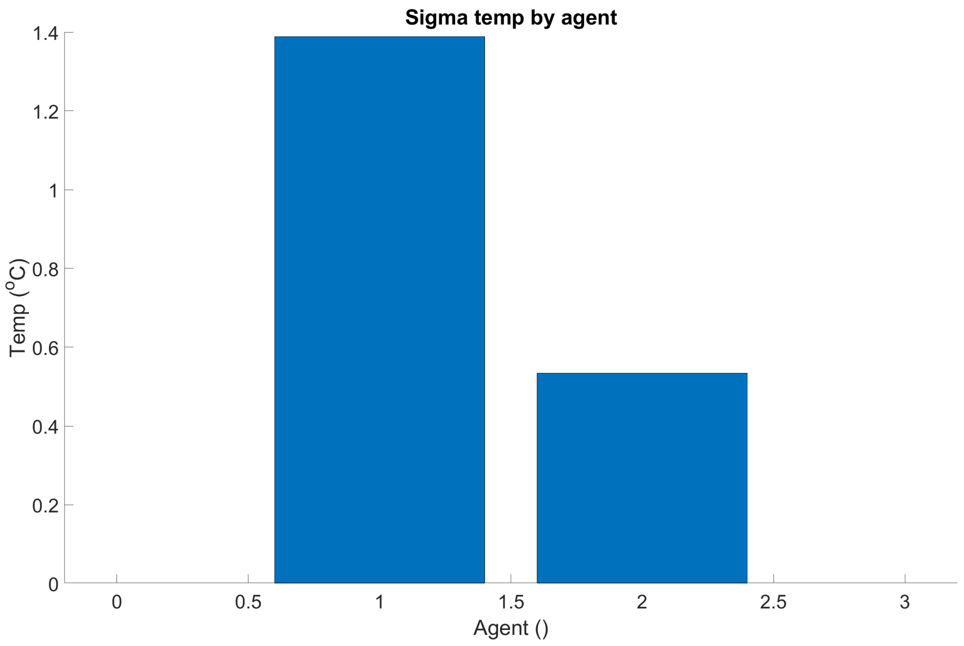


Figure D.64: 1-hour temperature standard deviation by agent

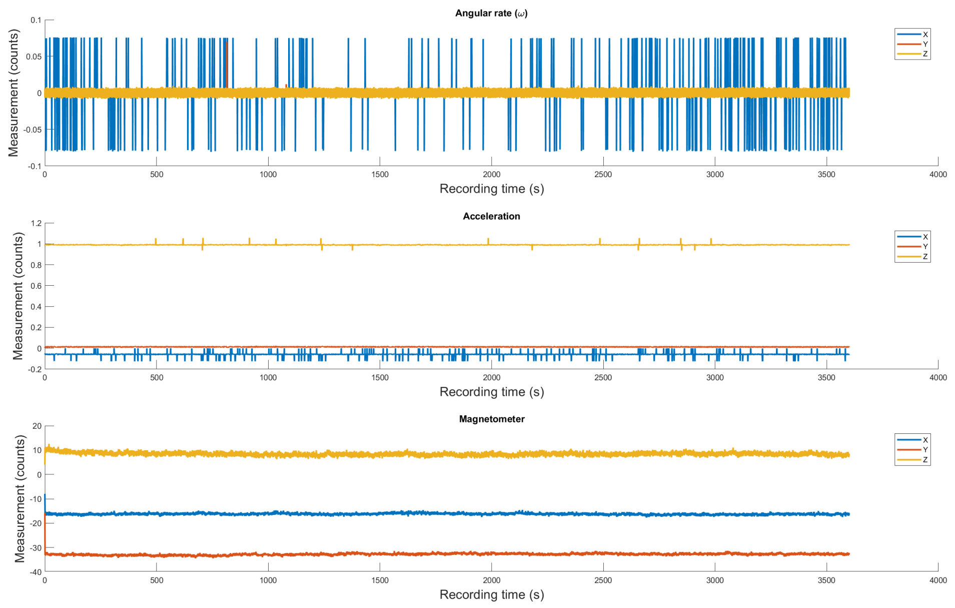


Figure D.65: 1-hour recorded HWIL gyro, accelerometer, and magnetometer

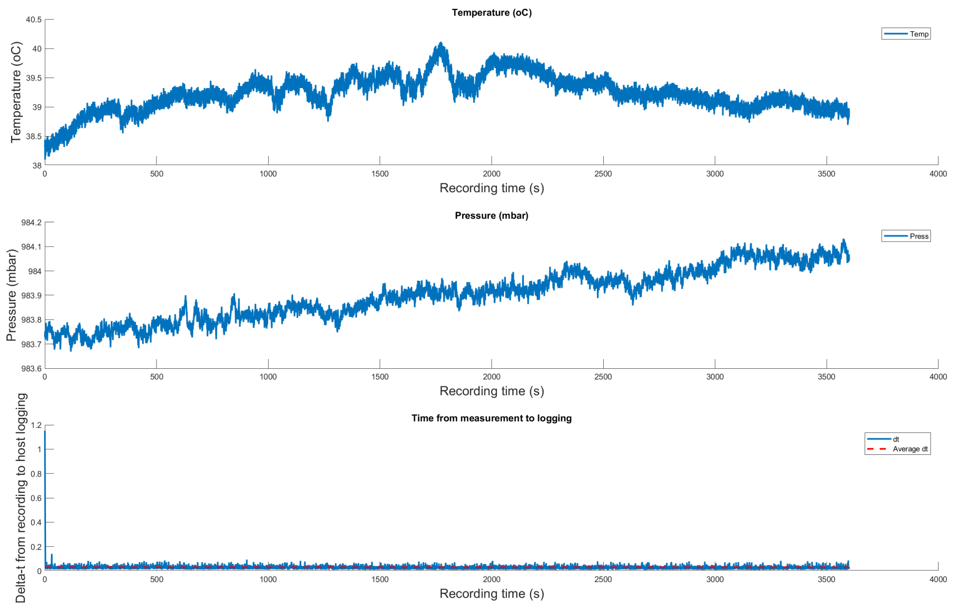


Figure D.66: 1-hour recorded temperature, pressure, and calculated packet Δt

REFERENCES

[1] C. Dillow, "Here's why small satellites are so big right now," *Fortune.com [online]*, Aug. 2015, Retrieved on 10/4/2016.

[2] K. J. Schilling, "Robotics for efficient production fo satellite constellations," 2017.

[3] M. Safyan, "Overview of the planet labs constellation of earth imaging satellites," ITU Symposium and Workshop on small satellite regulation and communication systems, Prague, Czech Republic, 2-4 March 2015, 2015.

[4] NASA. (2016). NanoRacks-Planet Labs-Dove. `http://www.nasa.gov/mission_pages/station/research/experiments/1326.html`, (visited on 10/02/2016).

[5] NASA. (2017). The afternoon constellation. `https://atrain.nasa.gov/`, (visited on 09/07/2017).

[6] Wikipedia, *Sirius Satellite Radio — Wikipedia The Free Encyclopedia*, `https://en.wikipedia.org/w/index.php?title=Sirius_Satellite_Radio&oldid=737121064`, 2016.

[7] C. Kang and C. Davenport, "Spacex founder files with government to provide internet service from space," *The Washington Post [online]*, Jul. 2015, `https://www.washingtonpost.com/business/economy/spacex-founder-files-with-government-to-provide-internet-service-from-space/2015/06/09/db8d8d02-0eb7-11e5-a0dc-2b6f404ff5cf_story.html`.

[8] J. Foust, "The return of the satellite constellations," *The Space Review [online]*, Mar. 2015, `http://www.thespacereview.com/article/2716/1`.

[9] J. Radtke, E. Stoll, H. Lewis, and B. B. Virgili, "The impact of the increase in small satellite launch traffic on the long-term evolution of the space debris environment," Apr. 2017.

[10] A. Rosengren, D. Amato, J.Daquin, and I. Gkolias, "The dynamical placement of satellite constellations and designing for demise," 2017.

[11] A. Hawkins, J. Carrico, S. Motiwala, and C. MacLachlan, "Flight dynamics operations and collision avoidance for the skysat imaging constellation," 2017.

[12] J. A. Haimerl and G. P. Fondler, "Space fence system overview," 2015.

[13] S. Engelen, S. Eberhard, and C. Verhoeven, "Systems engineering challenges for satellite swarms," *IEEE Aerospace Conference*, 2011.

[14] V. Trianni, R. Gross, T. Labella, E. Sahin, and M. Dorigo, "Evolving aggregation behaviors in a swarm of robots," *Advances in Artificial Life. Ecal 2003. Lecture Notes in Computer Science.*, vol. 2801, 2003.

[15] T. Schetter, m. Campbell, and D. Surka, "Multiple agent-based autonomy for satellite constellations," *Artificial Intelligence*, vol. 145, pp. 147–180, Apr. 2003.

[16] NASA, "NASA and DARPA sponsor international debris removal conference," *Orbital Debris Quarterly News*, vol. 14, pp. 1–2, 1 Jan. 2010.

[17] SpaceWorks Enterprises Inc. (2017). Spaceworks releases 2017 nano/microsatellite market asessment. `http://spaceworksforecast.com/2017-market-forecast/`, (visited on 09/17/2017).

[18] J. Foust, "Managing a flock of satellites," *Space News*, pp. 17–19, Aug. 2017.

[19] J. Luft and H. Ingham, "The Johari Window," *Human relations Training News*, vol. 5, pp. 6–7, 1 1961.

[20] C. Foster, J. Mason, V. Vittaldev, L. Leung, V. Beukelaers, L. Stepan, and R. Zimmerman, "On hardware-in-the-loop simulation," *44th IEEE Conference on Decision and Control and the European Control Conference 2005*, Dec. 2005.

[21] P. J. Teunissen and O. Montenbruck, Eds., *Springer Handbook of Global Navigation Satellite Systems*. Gewerbestrasse 11, 6330 Cham, Switzerland: Springer International publishing, 2017, ISBN: 978-3-319-42926-7.

[22] W. J. Perry, B. Scowcroft, J. Nye, and J. Schear, "Space traffic control the culmination of improved space operations subject and problem statement," 2005.

[23] D. Werner, "Hazardous intersection," *Space News*, pp. 17–20, Sep. 2017.

[24] K. Sobh, K. El-Ayat, F Morcos, and A. El-Kadi, "Scalable cloud-based leo satellite constellation simulator," *World Academy of Science, Engineering and Technology International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 9, no. 6, pp. 1467–1478, 2015.

[25] M Bacic, "On hardware-in-the-loop simulation," *44th IEEE Conference on Decision and Control and the European Control Conference 2005*, Dec. 2005.

[26] US Department of Defense. (2011). Technology readiness assessment (TRA) guidance. http://www.acq.osd.mil/chieftechnologist/publications/docs/TRA2011.pdf, (visited on 09/07/2017).

[27] US Department of Energy. (2011). Technology readiness assessment guide. https://www.directives.doe.gov/directives-documents/400-series/0413.3-EGuide-04-admchg1, (visited on 09/07/2017).

[28] NASA. (2012). Technology readiness level. https://www.nasa.gov/directorates/heo/scan/engineering/technology/txt_accordion1.html, (visited on 09/07/2017).

[29] I. Kant, *The Critique of Pure Reason*, trans. by J. Tkeiklejohn. 1781, Retrieved from Project Gutenberg.

[30] MPICH. (2018). MPICH: High-Perforomance Portable MPI. https://www.mpich.org/.

[31] Software in the Public Interest. (2018). Open MPI: Open Source High Performance Computing. https://www.open-mpi.org/.

[32] C. DeGraw and M. J. Holzinger, "A Massive-Scale Satellite Constellation Hardware-in-the-Loop Simulatr and Its Applications," 2017.

[33] N. Elhage. (2011). Made of bugs: Exploiting misuse of python's "pickle". https://blog.nelhage.com/2011/03/exploiting-pickle/.

[34] Raspberry Pi Foundation. (2018). Sense HAT. https://www.raspberrypi.org/blog/astro-pi-tech-specs/.

[35] ——, (2018). Astro Pi: Flight Hardware Tech Specs. https://www.raspberrypi.org/products/sense-hat/.

[36] C.-F. Natali. (2013). Python bugs: Msg180663. https://bugs.python.org/issue17038#msg180663.

[37] N. Slatt, "FCC approves SpaceX's ambitious satellite internet plans," *The Verge*, Mar. 2018, https://www.theverge.com/2018/3/29/17178126/spacex-satellite-broadband-internet-fcc-approval-license-starlink-spectrum.

[38] M. Harris, "FCC accuses stealthy startup of launching rogue satellites," *IEE Spectrum*, Mar. 2018, https://spectrum.ieee.org/tech-talk/aerospace/satellites/fcc-accuses-stealthy-startup-of-launching-rogue-satellites.

[39] Staff, "BMW faces u.s. lawsuit claiming emissions cheating," *Reuters*, Mar. 2018, `https://www.reuters.com/article/us-bmw-emissions-lawsuit/bmw-faces-u-s-class-action-lawsuit-over-emissions-idUSKBN1H326I`.

[40] J. Knupp. (2013). Python's hardest problem, revisited. `https://jeffknupp.com/blog/2013/06/30/pythons-hardest-problem-revisited/`.

[41] Various. (2017). IOError: [Errno 24] Too many open files: `https://stackoverflow.com/questions/18280612/ioerror-errno-24-too-many-open-files`.

[42] Various. (2010). How accurate is python's time.sleep()? `https://stackoverflow.com/questions/1133857/how-accurate-is-pythons-time-sleep`.

[43] Ubuntu. (2015). UbuntuStudio/RealTimeKernel. `https://help.ubuntu.com/community/UbuntuStudio/RealTimeKernel`.

[44] E. Douglass, M. J. Holzinger, J. W. McMahon, and A. Jaunzemis, "Formation control problems for decentralized spacecraft systems," Aug. 2013.

[45] M. J. Holzinger and J. W. McMahon, "Decentralized mean orbit-element formation guidance, navigation, and control: Part 1," Aug. 2012.