A REAL TIME THERMAL SIMULATOR FOR SMALL SPACECRAFT IN AN ORBITAL ENVIRONMENT

BY

QI LIM

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Aerospace Engineering
in the Graduate College of the
University of Illinois Urbana-Champaign, 2024

Urbana, Illinois

Adviser:

Clinical Associate Professor Michael F. Lembeck

ABSTRACT

This thesis presents a thermal solver for small spacecraft written in MATLAB that interfaces with a.i. Solutions' FreeFlyer mission design software. Thermal control on small spacecraft is a dynamic challenge, influenced by the ever-changing environment. Heat is sourced from both within and outside the spacecraft. Powered components impose heat loads, radiating and conducting energy into their surroundings. The sun and other celestial bodies contribute a varying flux upon the exterior of the spacecraft depending on the spacecraft's relative position. The thermal solver takes all these factors into account, simulating the spacecraft's environment, and produces interpretable results for analysis.

Thermal analysis examines potential temperature gradients inside a spacecraft. The thermal solver written for this thesis is a user-friendly tool for space systems design tasks. Integrating this solver with an industry-level orbital propagator can improve the fidelity of thermal analysis beyond that commercial-grade software packages typically provide. Where standalone analysis software sometimes presents a cumbersome user experience for setting up an orbital environment, this solver solves the issue by interfacing with an external propagator, a.i. Solutions' FreeFlyer™. This allows for the setup of complex missions that may involve orbital maneuvers and changes in spacecraft attitude, providing an integrated analysis mission model.

Verification of the thermal solver was demonstrated by correlating results with the available Siemens Simcenter 3D commercial software. Future improvements to the user interface are possible, but the basic solver is now ready for use in analyzing Laboratory for Advanced Space Systems at Illinois (LASSI) missions.

For my family.

ACKNOWLEDGMENTS

My unending thanks goes out to the many people who have helped me through this thesis. Primarily, my advisor, Dr. Michael Lembeck, has been of greatest help and encouragement, always excited to see me succeed. Though I wade through rivers of red ink, I would not have it any other way.

I must thank my friends and peers, Eric Alpine, Rick Eason, Michael Harrigan, Chris Young, James Helmich, Hongrui Zhao, Murphy Stratton, Courtney Trom, and Dave Gable. Every day is a treat in your company, and you have taught me half of everything I know.

I would also like to thank Greg Mathy, Damien Vanderpool, Lorraine Guerin, Lina Maricic, and Eoghan O'Neill. Your gentle mentoring and patience raised me up out of ignorance.

And of course, my family. You give and give without expectation. As proud of me as you are, it can never match up to how proud I am to be your son and brother.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

From July 16th to July 18th, 1969, the Apollo 11 command and service module carried three American astronauts from Earth to Lunar orbit. During this journey, it performed what is known as the "barbeque roll," a rotation about the lengthwise axis, to prevent overheating on one side from constant exposure to the Sun's radiation [1]. The success of this maneuver ensured the safe voyage of the astronauts into the Moon's orbit as well as their return home. The aerospace engineers of that era knew that thermal control would be a critical challenge to face for their spacecraft and, five and a half decades later, engineers of the current day face similar challenges in this timeless arena.

Today's spacecraft, large and small, crewed and uncrewed, take advantage of thermal control systems not available during the Apollo era. Thermal engineers use modern tools to assess the performance of passive and active systems to control the spacecraft thermal environment. Through analysis, the spacecraft can be designed with the correct systems for the job, ensuring that it succeeds with precision. Passive control systems have been improved using advanced materials for insulation, reflection, and emission of heat energy. These systems help to maintain a stable thermal environment within the spacecraft without expense of power. Active control systems have also been modernized, enabling the transfer of heat through and out of the spacecraft with high reliability and efficiency.

Thermal analysis software, employing finite element or finite difference methods, is used to calculate temperature distributions across various systems of interest. Such software includes Thermal Desktop, COMSOL, and Simcenter 3D, which are widely used in industry and in academia. In the Laboratory for Advanced Space Systems at Illinois (LASSI), Simcenter 3D has been extensively utilized to analyze temperature profiles of small spacecraft.

Thermal simulation of spacecraft carries with it another challenge not evident in simulation of ground-based systems. The orbital environment is a highly variable, transient environment in which external sources of heating may appear, disappear, increase, decrease, and translate across the spacecraft [2]. Many thermal analysis software tools incorporate a rudimentary orbital propagator to provide an approximation of this environment, however these are vastly limited in their capabilities. Such solvers are primarily focused upon simulating a thermal system, not solving an N-body problem. Fidelity is lost in orbital propagation, which impacts the overall simulation accuracy. In such cases the problem may be redefined by examining only the worst-case hot and cold conditions along with the nominal operating conditions. This approach neglects transient cases, solving only for steady-state temperatures at fixed moments in time. While useful for finding the margin between expected results and analytical results, this fails to provide the engineer with information regarding the spacecraft's daily cycling.

Integration of a thermal analysis tool with an orbital propagator can resolve these issues. Mission analysis tools, such as the ANSYS Systems Tool Kit (STK), the NASA General Mission Analysis Tool (GMAT), and a.i. Solutions' FreeFlyer provide an accurate representation of the orbital environment. C&R Technologies' Thermal Desktop and ANSYS' STK exemplify the integrated approach. Unfortunately, such commercial software can be expensive and complex. This provides motivation for an alternate solution for academic mission analysis. This thesis describes the marriage of MATLAB code (to efficiently solve thermal problems for small spacecraft in real time) with the FreeFlyer mission analysis tool.

CHAPTER 2: BACKGROUND AND THEORY

Spacecraft are systems composed of several different subsystems, of which many may have critical components sensitive to temperature extremes. These components have operational temperature ranges within which they will perform optimally and survival temperature ranges outside of which the component may become damaged to the point of becoming inoperable. In the space environment, temperature can swing between extremes throughout an orbit, and therefore other systems must be employed to control the temperatures of sensitive components to prevent them from failing.

A range of factors, both external and internal, exert influence on a spacecraft's thermal equilibrium. This section delves into the theory behind on-orbit heating analysis of spacecraft and encompasses factors such as heating sources and strategies for maintaining thermal balance. Understanding why a spacecraft heats up and cools down, and how heat moves through the system, is crucial not only for effective control of the system but also for accurate simulations.

Thermodynamics

Thermal analysis is based upon the principles of thermodynamics which will be covered here in brief. There are four basic laws of thermodynamics, numbered zero to three, that describe the natural world of heat energy, work, and temperature. The laws explain why temperature changes and how heat moves, allowing engineers to mathematically represent these physical laws in solvable equations.

The history of thermodynamics dates to the 1600s, when early scientists postulated that heat was formed by the kinetic motion of matter. This was ultimately the correct notion, however shortly after, during the 1700s, a popular idea was that heat was a fluid, distinct from matter itself.

Eventually, during the 1800s, experiments from scientists including James Joule helped to establish that heat is a form of energy [3].

Over the following years, the laws of thermodynamics were established. In 1850, Rudolf Clausius and William Thomson defined the First Law of Thermodynamics, which provides the basis of energy conservation. The First Law of Thermodynamics relates the energy of the system to recognizable terms, stating that the internal energy of a system is equal to the difference between the heat transferred into a system and the work done by the system [4]. By convention, work put into a system is assigned a positive value and work done by a system is assigned a negative value, showing how energy is being added or removed. This law can be represented by a simple equation,

$$\Delta U = Q - W \tag{1}$$

where ΔU is the internal energy of the system, Q is the heat energy into the system, and W is the work done by the system, each with the unit of Joules (J). For a small spacecraft modeled by the thermal solver, little work will be done by the system. Energy exchange will occur almost entirely through heating and cooling.

Clausius and Thomson also stated the Second Law of Thermodynamics, though in a manner somewhat backwards to how it is known today. They stated that heat does not flow from a colder body to a hotter body [3]. Today, it would be better said that heat flows from a hotter body to a colder body. This is because the Second Law establishes the concepts of entropy and heat flow throughout a system. Entropy is the measure of energy that is unavailable within a system and can be represented by a state variable that is equal to the heat transfer divided by the temperature. Entropy increases throughout a system, meaning that useful energy becomes unable to produce work, over time. This results in irreversible processes in that performing an action in reverse will not release the unusable energy back into a usable form.

Intuitively, one understands Clausius' and Thomson's statement, that heat is always transferred from a hotter entity to a colder entity [5], but the reason becomes clear when the irreversibility of entropy is considered. Equation 2 provides the definition of entropy in mathematical form. As the change in entropy, ΔS, must be greater than zero within the system, the heat, Q, must go from the hotter temperature, T_h, to the colder temperature, T_c.

$$S_f = S_i + Q/T_h - Q/T_c$$
$$\Delta S = S_f - S_i > 0 \tag{2}$$

This reintroduces the concept of heat flow in which heat can sometimes be treated as a fluid, much like how the scientists of the 1700s imagined it to be. Through this concept, one can visualize heat energy flowing viscously through a medium, where a medium is a substance that can convey heat. Heat will spread as the medium dictates and given that certain properties of the medium are known, models can be created to accurately find the change in temperature of the medium over time.

The Third Law of Thermodynamics was formulated by Walther Nernst in the early 1900s [3] with the help of his student, Francis Simon. The Third Law states that the entropy of a system tends to zero as the temperature of the system approaches zero [6]. The practical understanding of this law is that bringing an entity down to absolute zero is essentially physically impossible. As temperatures in a real system never approach zero, this law finds little application to spacecraft thermal analysis.

Implied by the three basic laws of thermodynamics is the Zeroth Law of Thermodynamics. This law defines thermodynamic equilibrium, in that if two systems are separately in thermal equilibrium with a third system, they are also in equilibrium with each other [7]. Two systems that are in thermal equilibrium can therefore be stated to have the same temperature. This is a matter of associativity which allows for inference of systemwide temperatures.

2.1. Heating and Temperature

Heat and temperature are separate quantities, though they are closely related. Heat is a quantity of energy, measured in Joules, whereas temperature is a measure of the average kinetic energy of the particles moving in a substance, which is the intensity of the heat [8], measured in Kelvin. The more heat energy a substance has, the more kinetic energy the particles of the substance will have, and the greater the temperature that will be expressed. As different entities may not be composed of the same type of material, they may not share the same properties and therefore if they are given the same amount of heat energy, they will not necessarily gain the same amount of temperature. This is due to a material property called the specific heat capacity, c_p, which represents the amount of heat that must be added to a unit mass of the material to raise the temperature of the material by one unit. The specific heat capacity thus becomes a coefficient that relates the change in heat energy to the change in temperature of a material.

$$dQ = mc_p dT \tag{3}$$

The change in heat, dQ, which is the difference in heat energy between points in time, is equal to the mass of the material, m, in kilograms, multiplied by the specific heat capacity, c_p, in Joules per kilogram per Kelvin (J/kg/K), multiplied by change in temperature of the material, $dT = T_2 - T_1$. Given that the material and its specific heat capacity is known, a change in temperature, dT, will cause some calculable change in heat, dQ, and vice-versa. Note that dT is used here instead of ΔT. This is because ΔT will be used to indicate a difference in temperature between two different entities while dT is used to indicate a difference in temperature of one entity after a period of time.

2.2. Transference of Energy

Within a dynamic environment, in which conditions are ever changing, temperature cannot be a static quantity. Any environment that sees changes in conditions over time, whether that be within a climate-controlled building, outside in a winter snowstorm, deep under the sea, or up and out of Earth's atmosphere, will result in a material in that environment changing in temperature. Energy is transferred between the system and the environment in such cases, both into and out of the system. This results in changes to the temperatures of components within the system as their internal energy changes. Additionally, there can be additional transfers of energy within the system itself, which may result in temperatures shifting throughout the system. Components inside the system may produce or move heat, therefore changing the present balance of energy within the system.

Given enough time within an unchanging environment, a system will eventually settle into what is known as thermal equilibrium. This is when transfer of energy within the system and between the system and its environment becomes so small that the temperature changes become negligible. However, introducing a significant change into this environment will throw the thermal equilibrium out of balance and it is important to understand how this movement of energy into and out of the system can occur.

Heat transfer rate refers to the change in heat energy as a function of time, thus there must be a representation for the rate of change of heat, Q. The rate of change of heat energy is defined as,

$$\dot{Q} \equiv \frac{d}{dt} Q \tag{4}$$

with units of Joules-per-second (J/s), which is equivalent to Watts (W). This can also be referred to as power. As this is a rate of change in heat energy, it can be used to represent the

physical concept of energy taking time to move some unit distance. The medium for the transfer of this energy determines the rate and direction of the movement, of which there are three types: conduction, convection, and radiation.

2.2.1. Conduction

Conduction is the mechanism by which adjacent particles (i.e., particles next to each other) within a conducting medium transfer energy. This can be within a material or between touching materials. A classic example is the spoon in a hot cup of coffee. Assume that before the spoon is put into the coffee, it starts at thermal equilibrium with the air and thus is at roughly room temperature throughout its body. If an engineer picks up the spoon to stir the coffee, he/she would impart some thermal energy into the spoon, conducting from their hand into the cooler metal. Dunking the spoon into the hot liquid produces the same effect but on a larger scale. The spoon quickly heats up and if the engineer is not careful, the next time they touch the spoon they might burn themselves. In this example, heat has been conducted from the liquid coffee to the head of the spoon, then up through the spoon and into the handle. It is easy to intuit that heat energy has moved throughout the entire system, and with further reasoning one may realize that the coffee has been lowered in temperature by nature of the second law. Energy has been moved away from the coffee, lowering its temperature, and into the spoon, raising its temperature.

The spacecraft is just another system within which such behaviors can occur. If the spacecraft as a whole is taken as the system of interest, it may not have any conductive path to the environment. However, when the spacecraft is attached to another larger system such as the International Space Station (ISS), it is in adjacent contact with another entity and can transfer heat through conduction. Otherwise, it is experiencing the dynamic conditions of the space environment where it may be subjected to radiation, a method of heat transference covered in the following

sections. In zero gravity, convection, another heat transfer mechanism, does not play a role in spacecraft thermal control.

Heat transfer through conduction is determined by the conductivity of materials and the conductance between entities composed of distinct materials. Each material has a property called thermal conductivity, k, of units Watts per meter per Kelvin (W/m/K), which is the measure of a material's ability to conduct heat. The thermal conductivities of various materials can be found in common literature, such as Gilmore's *Spacecraft Thermal Control Handbook*. For Aluminum 6061-T6, a general aluminum alloy often used in aerospace applications, the thermal conductivity is roughly 167 W/m/K [2]. Another way to think of conductivity is as the value representing how easily heat flows through a material. A copper rod, with a thermal conductivity of 398 W/m/K, will allow heat to flow from one end to the other much more quickly than an aluminum rod.

Other properties inherent to a material include the specific heat capacity (covered in Section 2.1), its density, and its mass, the latter two being closely linked. The density of a material is measured as the mass of the material per unit volume, or kilograms per cubic meter in SI units. Knowing the density of some material along with its volume, the mass can be easily calculated. From there, a simplification of the specific heat formula (Equation 3) can be made, letting the product of mass and specific heat capacity be represented by the quantity, C_{th}. This is known as the thermal mass of the material and has units of Joules per Kelvin, (J/K). Using this information the specific heat capacity formula is simplified as follows.

$$dQ = C_{th}dT \tag{5}$$

Overall, this may seem to be a trivial simplification to perform, but it can be important in practical applications. Following the principle of conservation of mass, the thermal mass of a system should remain constant unless mass is truly being lost. This implies that an entity can be

represented solely by its thermal mass and its temperature can be easily related to the heat input or output. In other words, a volume of material can be replaced by a singular point mass with equivalent thermal mass to the material. Though modeling fidelity is lost in that temperatures at various locations across the material can no longer be found, computation resources can be saved.

Complex systems can thus be simplified into networks of point masses which makes for fast simulations that are reasonably accurate. Such networks are called thermal networks, which are defined as one or more entities and their interfaces, linked together by thermal resistances. Figure 1 depicts an example of such a network, in which five nodes are linked together with resistive oaths. These thermal resistances relate the transfer of heat from one node to another, where nodes may sometimes represent distinct entities rather than parts of a single entity. These resistances have the inverse value called conductance, which is derived from the conductivity of the constituent materials. Given only a few values, the heat transfer between each node can be calculated with relative ease.

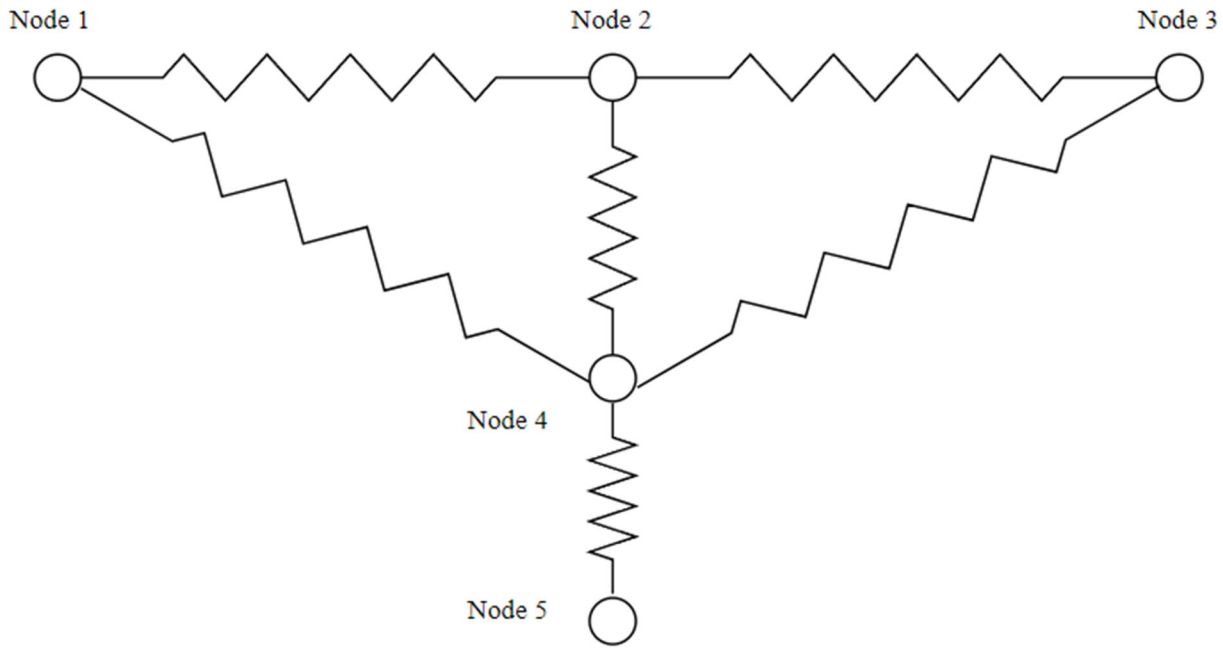

Figure 1. Thermal Network Example. A thermal network composed of five linked nodes.

Thermal networks are expressed almost identically as their electrical counterparts. Just as electrical resistance is the resistance a circuit element has to the internal flow of electricity, thermal resistance is the resistance an element, or a material, has to the flow of heat. Figure 2 provides an example of a simple thermal network. This example depicts a rod of length L, specific heat capacity c_p, thermal conductivity k, and mass m. The rod can be reduced to two point-masses/nodes, each representing half the thermal mass of the bar. The resistance between the two nodes, each approximating the ends of the rod, is the thermal resistance of the network, R, in units of Kelvins per Watt (K/W).

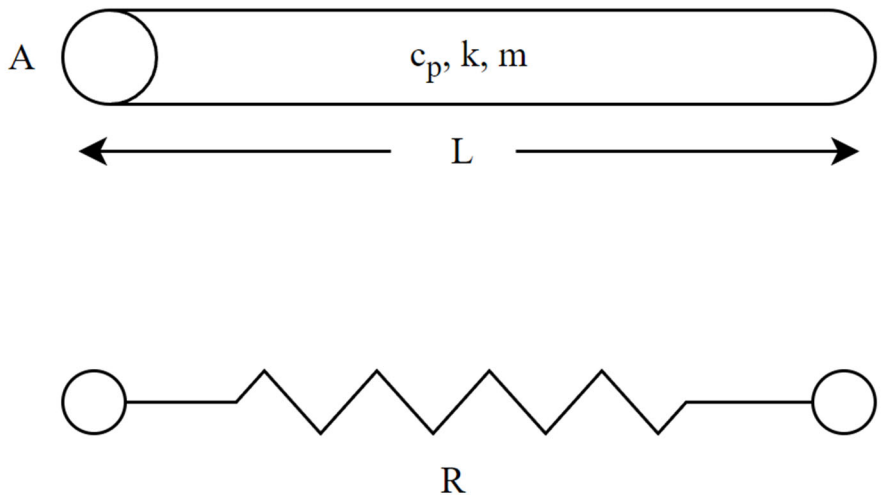

Figure 2. Rod Example. A cylindrical rod is simplified to a two-node thermal network.

A thermal resistance, *R*, has an inverse value called conductance, *G*, in units of Watts per Kelvin (W/K). This is the measure of how well heat flows between individual nodes. The thermal conductance can be used to express heat flow through a single entity as seen above, or the connections between two or more entities, representing how well heat flows through their touching surfaces (also known as interfacing or faying surfaces). In fact, all thermal interfaces can be simplified into connections with *G* and *R* values between representative nodes. By building a matrix of *G* values between each node (i.e., building a thermal network), calculations of heat transfer and temperature change can be computed.

The thermal conductance, *G*, is calculated for connections within an entity with the following equation, given the thermal conductivity, *k*, of the material the entity is composed of.

$$G = \frac{kA}{L} \tag{6}$$

Here, *A* is the cross-sectional area of heat transfer in square meters and *L* is the distance of travel in meters. Its inverse, the thermal resistance, *R*, is calculated as,

$$R = \frac{1}{G} = \frac{L}{kA} \tag{7}$$

When encountering a problem that involves heat transfer between adjacent, individual entities, the thermal area conductance, h, is introduced. Thermal area conductance has the units of Watts per Kelvin per square-meter ($W/K/m^2$) and is found empirically given the adjacent materials and the quality of contact between them. To find the thermal conductance is a simple multiplication of h with the area of the interface.

$$G = hA_{interface} \tag{8}$$

In this fashion, the resistances and conductances of a thermal network can be found for heat transfer computations. For entities that are bolted together, the value of h can be taken as 2000 $W/K/m^2$ for a good approximation. This is only viable for a small area around each bolted joint as the pressure exerted by the bolt upon the surfaces is only fully applied in the circular area defined by the bolt head. The pressure, and therefore the quality of the connection decreases the further one travels from the bolt. Unbolted connections require information about the pressure acting between the two faying surfaces to determine the quality of the connection, and therefore the thermal area conductance value.

A final note should be made about thermal resistance, R. As mentioned above, R can be likened to electrical resistance. This is not just in how it is depicted in thermal resistance diagrams but also in how networks can be further simplified. Take, for example, multi-layer insulation (MLI). The insulation is composed of two or more layers of material, each with its own conductivity and thermal area conductance. The internal temperature of the insulation may not be of interest, but temperatures on either side of the MLI are useful. Therefore, simplifying the MLI's conductances into one value representing how well heat flows through the insulation from one side to the other is required. This can be represented as a series network, where heat flows from one node to the next (Figure 3). The conductance between each layer of insulation is found as the

product of the thermal area conductances and the interface areas and can then be inverted to find the individual resistances between each layer. Just like an electrical circuit, the total thermal resistance can then be calculated as the sum of the resistances in the series.

$$R_{tot,series} = \sum R_i \tag{9}$$

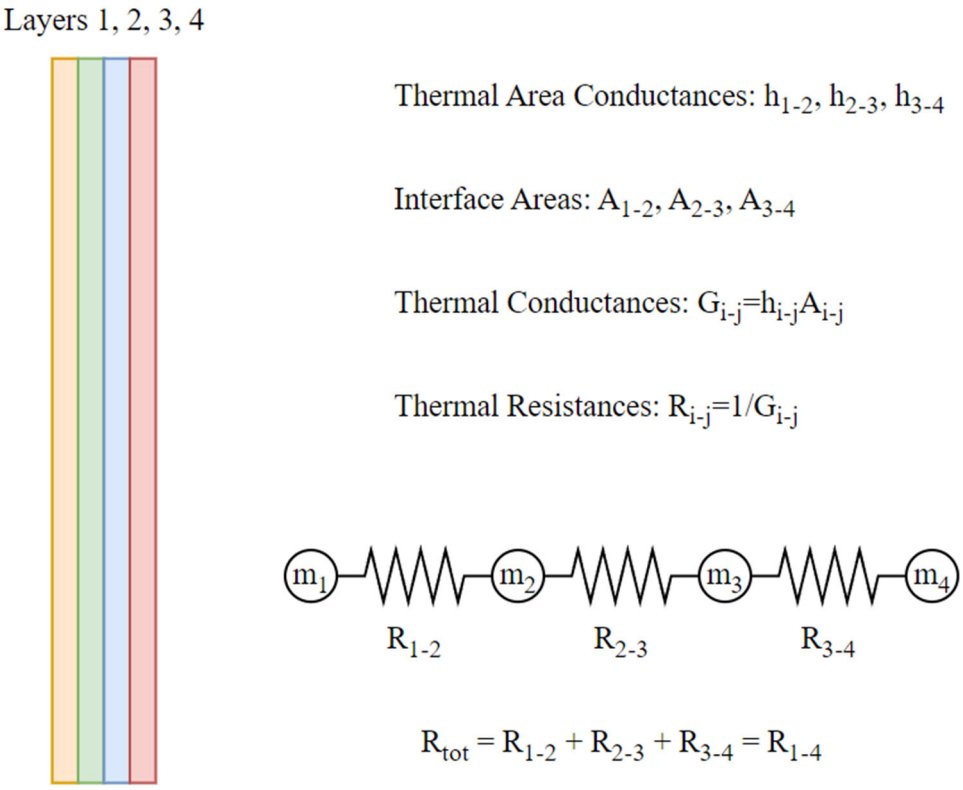

Figure 3. Multilayer Insulation Series Network. The total thermal resistance through MLI can be computed with a series resistor calculation.

Parallel thermal network heat transfer can also be calculated using this electrical network analog. For example, take the case of a wall with a window. The wall is insulated, significantly slowing heat flow from the interior of the building to the exterior and vice versa. However, the glass window is a separate entity through which heat can also flow. A parallel path for heat flow exists: one branch leads from interior through the wall to the exterior while the other branch leads

from interior through the window to the exterior (Figure 4). The total resistance is then calculated as the inverse of the sum of the inverses of each resistance.

$$R_{tot} = \left(\sum \frac{1}{R_i}\right)^{-1} \tag{10}$$

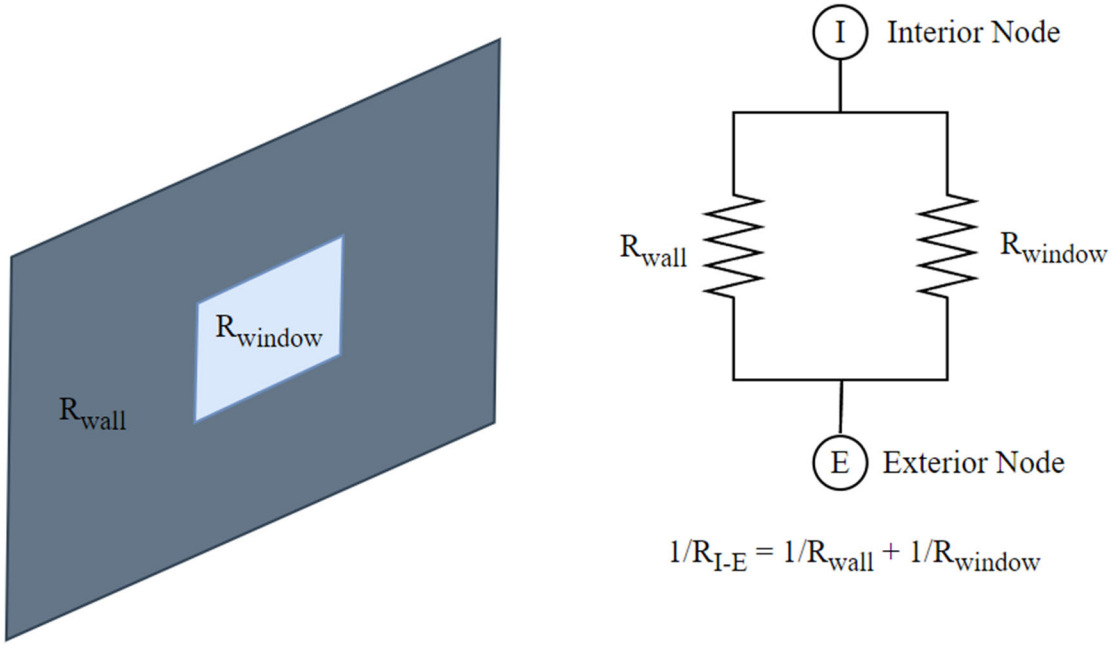

Figure 4. Window and Wall Parallel Network. The total thermal resistance through a wall and a window can be computed with a parallel resistor calculation.

Computation of the conduction through a network is a matter of setting up proper resistances and computing overall conductance through the various materials and paths. This may take empirical data to complete, which is often tabulated or graphically represented in literature, including Gilmore's text [2].

2.2.2. Convection

Convection is the transfer of energy through a fluid or gas. Returning to the earlier example, a cup of hot coffee will eventually cool down to room temperature as molecules of air are imparted

with energy from the cup and the coffee. The molecules of coffee and of the solid cup have much higher kinetic energy than the air around them. When an air particle gets close enough, energy is imparted from the cup or the coffee to the air molecule. This is, in essence, conduction, except as a transfer of energy to a gas. However, in the absence of gravity, beyond the atmosphere, there are very few molecules to convect heat in or outside of the spacecraft. This results in no heat transfer via convection. For this reason, thermal analysis of spacecraft will often neglect convection.

2.2.3. Radiation

Thermal radiation, the remaining mechanism by which heat can be transferred, is the transfer of energy through electromagnetic radiation [9] between entities. Thermal radiation is emitted by all objects above absolute zero [10], which in essence means every material and every surface emits thermal radiation in some form. This means that all objects are emitting energy at a rate proportional to their temperature to the fourth power, and that all objects are also receiving energy through radiative effects. In space, where the effects of convection are negligible, radiation becomes the primary source of heat from the environment as well as a major contributor to heat transfer between objects within the spacecraft. It is extremely important to consider radiation within thermal analysis models.

The sun and nearby celestial bodies (i.e., the Earth, for Earth-orbiting spacecraft) are the primary sources of external radiation impacting a spacecraft. The sun emits a massive amount of energy which is often known as the solar flux, *S*, in units of Watts per square-meter (W/m^2). This value is dependent on the distance the spacecraft is from the sun. It can be calculated as,

$$S = \frac{L}{4\pi d^2} \tag{11}$$

where L is the solar luminosity given as $L = 3.83 \times 10^{26}$ Watts, and d is the distance from the sun in meters [11]. The solar flux is essentially a power calculation of energy given off from a

sphere. In the case of Earth-orbiting spacecraft, over the course of a year, the Earth moves closer or further away from the sun, which means that the solar flux around the Earth will fluctuate. This solar flux changes within a range of 1322 W/m^2 at the summer solstice and 1414 W/m^2 at the winter solstice [2]. At the average distance of the Earth from the Sun, the value can be approximated as 1367 W/m^2, assuming the spacecraft is one astronomical unit (AU) from the sun.

Knowing the solar flux impacting the spacecraft, the heating rate on an incident surface of the spacecraft can be calculated as,

$$\dot{Q}_s = \alpha_s SA \cos\theta_s \tag{12}$$

where $\dot{Q}_s$ is the heating rate in Joules per second or Watts, α_s is the solar absorptivity of the surface, A is the area of the surface in square-meters, and θ_s is the angle of incidence of radiation upon the surface, taken as the angle between the normal of the surface and the vector representing the incoming solar radiation. Note that α_s here represents solar absorptivity.

Objects will emit radiation in specific wavelengths dependent on their composition. The radiation emitted by the sun, for instance, is primarily emitted in wavelengths greater than 700 nanometers [12]. A surface will absorb incoming radiation at different rates depending on the wavelength of the radiation, therefore the value, α_s, must be specifically defined for a surface. Additionally, it is important to recognize that $A\cos\theta$ is simply the projected area of the surface with respect to the incoming radiation, as oblique angles will result in a smaller overall cross-section for radiation to impinge upon.

The solar flux makes a direct contribution to the environmental heating of the spacecraft. Indirect solar radiation, known as albedo, is the reflection of light off the surface of the body around which the spacecraft orbits. Albedo changes depending on the reflectivity of the surface of the orbital body. For instance, the Earth's cloud coverage, the snow and ice below Earth-orbiting

spacecraft, the solar incidence angle, and the presence of water within view, will all significantly affect the reflected light a near-Earth spacecraft might see. Only some portion of the light from the sun will be reflected as albedo, which means that a value, the albedo factor, can be used to express how much solar flux becomes albedo heat flux. The standard practice is to use an approximate value of albedo factor corresponding to the spacecraft's current orbit. Appropriate values can be found in literature such as Gilmore's *Spacecraft Thermal Control Handbook [2]*. For very rough estimates, a value of 0.36 can be assumed for Earth-orbiting spacecraft, though it is important to recognize that the albedo contribution of heating radiation will go to zero when the spacecraft is eclipsed by the orbital body. At that point, there can be no light reflected off the body that directly impacts the spacecraft. Outside of such cases, the contribution of heat energy from albedo on a surface is found as,

$$\dot{Q}_a = \alpha_s ASF \cos\gamma \tag{13}$$

where $\dot{Q}_a$ is the heating rate in Joules per second or Watts, α_s is the solar absorptivity of the surface, A is the area of the surface in square meters, S is the solar flux in Watts per square-meter (W/m^2), F is the albedo factor, and γ is the reflection angle of solar flux off the orbital body. When in eclipse, albedo radiation goes to zero.

As stated earlier, nearby celestial bodies are also sources of radiation, not just a mirror for solar light to reflect into the spacecraft. These give off thermal radiation in the infrared spectrum as they, like the spacecraft itself, are radiating bodies. As with albedo, this value varies depending on the factors of the body's surface. For Earth, this can include the local temperature and the amount of cloud cover. The higher the temperature of the surface, the more infrared radiation is emitted. The greater the cloud cover, the more the emitted radiation is blocked from escaping into

space. To determine the flux of infrared energy emitted from a body, the following equation can be used:

$$q_{IR} = \sigma T_{body}^4 \tag{14}$$

where q_{IR} is the flux of IR energy emitted by the body in units of Watts per square-meter (W/m^2), σ is the Stefan-Boltzmann Constant (SBC) which is equal to 5.67037×10^{-8} $W/m^2/K^4$, and T_{body} is the temperature of the radiating body. This is known as the Stefan-Boltzmann Law [13] for a black body ideal emitter, where a black body is a theoretical entity that perfectly absorbs and emits energy. The local temperature of the body about which a spacecraft orbits varies significantly, so it is computationally efficient to simply take the local average temperature of the body's surface to find the output thermal radiation. For Earth, the temperature varies significantly, so the value T_{body} is often taken as the average temperature of the Earth, 255 Kelvin. From this equation, the amount of heat energy absorbed by the spacecraft per unit time is found as,

$$\dot{Q}_{IR} = q_{IR}\alpha_{IR}V_{body}A \tag{15}$$

where $\dot{Q}_{IR}$ is the heating rate in Joules per second or Watts, α_{IR} is the infrared absorptivity of the surface, V_{body} is the view factor the surface has of the orbital body, and A is the area of the surface in square meters. The infrared absorptivity of a surface, α, can be equated to the infrared emissivity of the surface, ε. The view factor describes how much of surface's view is taken up by the orbital body as a fractional number and is further discussed in Section 2.2.4.

The exterior of the spacecraft is the medium by which it can emit energy out into the surrounding environment. This value, $\dot{Q}_{out}$ is the amount of heat energy per unit time rejected by a surface of the spacecraft and is found with the following equation,

$$\dot{Q}_{out} = \varepsilon A\sigma T^4 \tag{16}$$

where $\dot{Q}_{out}$ is in units of Joules per second or Watts, ε is the IR emissivity of the surface, A is the area of the surface, σ is the Stefan-Boltzmann Constant, and T is the temperature of the radiating surface. This value is a heat loss, and therefore will often be given a negative value within heat balance equations. It should also be noted that this value must be calculated for each individual surface of a spacecraft, including the interior ones. As radiation is both emitted and absorbed on the exterior and interior of a spacecraft, the radiation emitted by any surface must be taken into account to understand the thermal network and heat transfer within a spacecraft.

2.2.4. View Factors Introduction

View factors refer to the fraction one surface takes up of another surface's total view. Refer to the example of a pair of finite identical parallel disks some distance L from one another with radius R, as depicted in Figure 5.

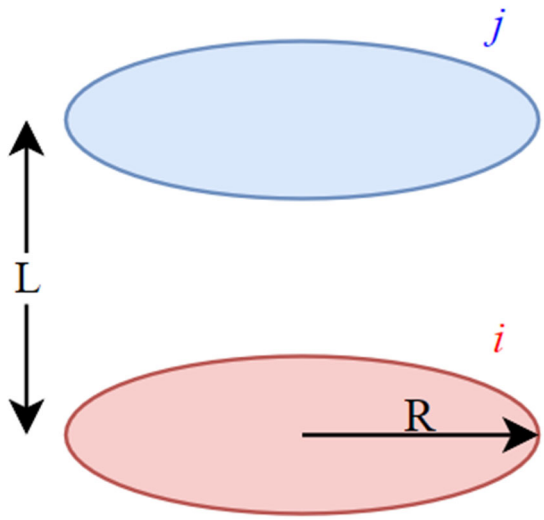

Figure 5. Finite Parallel Disks. A pair of parallel disks with identical finite radii are placed at a distance L from each other.

If one were to draw a hemisphere emitting from the lower disk, *i*, the fraction of the area of that hemisphere of which a projected mask of the upper disk, *j*, takes up, is the view factor.

Another way to think about this is to imagine that *j* casts a shadow upon the hemisphere, visualized as a dome which is centered around *i* (Figure 6).

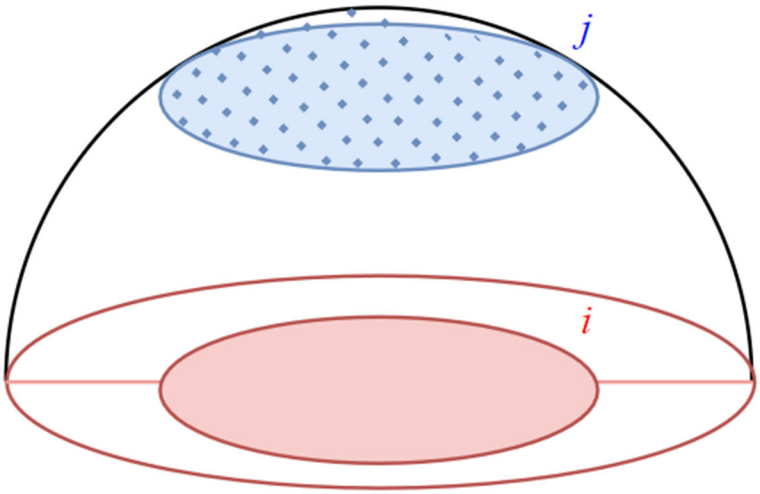

Figure 6. Hemisphere and Shadow. From the perspective of the red disk, the blue disk shadows a portion of the hemisphere drawn from the red disk.

The percentage of the total surface area of the dome that is shadowed is the "view factor." This means that if the disk *i* is radiating energy out from its upper surface, only that percentage of the energy output is received by the other disk *j*. A useful rule, known as the reciprocity rule, can be used to find the reverse view factor,

$$F_{ij}A_i = F_{ji}A_j \tag{17}$$

where F_{ij} is the fraction of the energy leaving i that arrives at j, F_{ji} is the fraction of the energy leaving j that arrives at i, A_i is the area of i, and A_j is the area of j. This equation notes that the view factor in the opposite direction is simply the reciprocal of the view factor initially calculated for i [14]. Thus, knowing the view factor that disk i has of area j allows for the calculation of the view factor that *j* has of *i*, as well as the calculation for the proportion of energy *j* radiates to *i*.

For spacecraft near a celestial body, it may also be necessary to calculate the view factor each surface has to the nearby celestial body to find the infrared radiation heating caused by that body. This is because the surface may not necessarily see the entire celestial body at once, though another case arises when both sides of a surface can see the body at the same time. This is best illustrated with a flat plate modeled above a body.

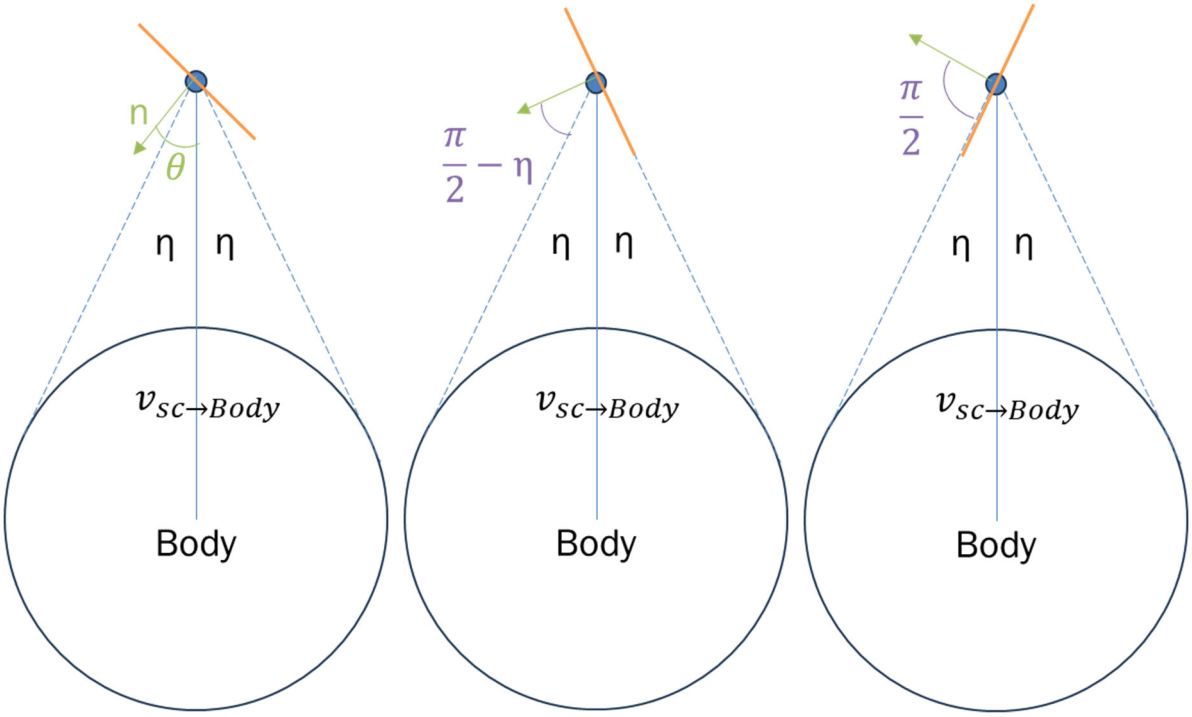

Figure 7. View Factors of a Spherical Body. A flat plate above a spherical body may take three orientations of interest (1) only the top of the flat plate sees the body; (2) both sides of the plate see the body; (3) only the bottom side of the plate sees the body.

A flat plate is illustrated in Figure 7 as an orange line with a normal vector, n, in green. The nadir vector $v_{sc\to Body}$ is drawn between the plate and the orbital body, and the angle between that vector and the horizon of the orbital body is η. The plate receives energy from both albedo and IR coming from the body, though as mentioned earlier, the view each side of the plate has is only partially taken up by the orbital body. The view factor each surface has of the orbital body must therefore be calculated each time the view changes. There are three viewing cases for the plate which can occur as the view changes: (1) exclusively the top side of the plate has view of the

body; (2) exclusively the bottom side of the plate has view of the body; (3) both sides of the plate can view the body. The first case occurs when the angle the normal vector makes with the nadir vector, θ, is less than the angle $\frac{\pi}{2} - \eta$. In such a case, the view factor formed is calculated as,

$$F = \frac{\cos\theta}{H^2} \tag{18}$$

where F is the view factor and H is a normalized value of the distance the spacecraft is from the center of the massive body [14]. Defining,

$$r \equiv \left\| v_{sc \to Body} \right\| \tag{19}$$

as the distance the spacecraft is from the center of the massive body and letting R be the radius of the body, H is found as,

$$H = \frac{r}{R} \tag{20}$$

The second case, where only the bottom side of the plate can see the body, occurs when θ is greater than the angle $\frac{\pi}{2} + \eta$. The view factor formed is calculated in the same manner.

The third case, where both sides of the plate can see the body, occurs when θ is between $\frac{\pi}{2} - \eta$ and $\eta + \frac{\pi}{2}$. The view factor for each side in this case is calculated as.

$$F = \frac{2}{\pi}\left[\frac{\pi}{4} - \frac{1}{2}\sin^{-1}\left[\frac{\sqrt{H^2-1}}{H\sin\theta}\right]\right.$$

$$\left. + \frac{1}{2H^2}\left\{\cos\theta\cos^{-1}\left[-\sqrt{H^2-1}\cot\theta\right] - \sqrt{H^2-1}\sqrt{1-H^2\cos^2\theta}\right\}\right] \tag{21}$$

It is necessary to break up these views in this manner as the angle θ cannot approach zero or the view factor calculated with Equation 21 would diverge. Therefore, two equations are necessary to provide the view factors a surface would have of a nearby celestial body.

2.2.5. Radiative Exchange Between Surfaces

Energy is radiated by both interior and exterior spacecraft surfaces. While the externally rejected radiation may be lost to deep space, the internal radiation will remain part of the system. The surfaces that radiate and are impinged upon by radiation within the spacecraft are undergoing what is called "radiative exchange between surfaces."

Each surface is radiating energy at a rate, $\dot{Q}_{out}$, described in Equation 16. This occurs on both sides of the surface, defined as the top and the bottom sides. This energy will radiate to other surfaces, and the energy absorbed by the other surface is dependent on the view factor the originating surface has with the other radiating surfaces within the system, as well as their areas and radiated energy. This is summed up per surface as,

$$\dot{Q}_{i,in} = \varepsilon_i \sum_{j=1}^{N} F_{ij} \dot{Q}_{j,out} \tag{22}$$

where $\dot{Q}_{i,in}$ is the heating rate of surface i, ε_i is the IR emissivity of surface i (which is equivalent to the surface's IR absorptivity, $\alpha_{i,IR}$), F_{ij} is the fraction of energy that leaves surface j that is incident on surface i stated as the view factor i has of j, and $\dot{Q}_{j,out}$ is the rate of heat energy leaving surface j as radiation. This gives the direct radiative exchange rate between each surface i and j, where radiated energy from j is absorbed by i.

There are several different methods to calculate the view factor F_{ij} between surfaces i and j. Closed form solutions exist for specific cases but do not cover all possible configurations that may arise within a spacecraft. To find the view factor for an ambiguous case, Monte Carlo Ray Tracing (MCRT) is a common method. MCRT involves the use of rays, which are vectors of light, which are "shot" out of a source surface. The rays are shot from random positions and at random angles, and given enough rays, the dome described in Section 2.2.4 can be visualized. If the rays

hit a surface that is not their source surface, they can be stopped and counted. If the number of hits on a surface is known, this value can be compared to the number of rays shot from the source surface, which produces a proportional value approximately equal to the view factor the source surface has of the hit surface. The more rays that are shot out from the source surface, the more accurate this solution becomes. Figure 8 depicts an example of MCRT with the two parallel disks used in the previous examples.

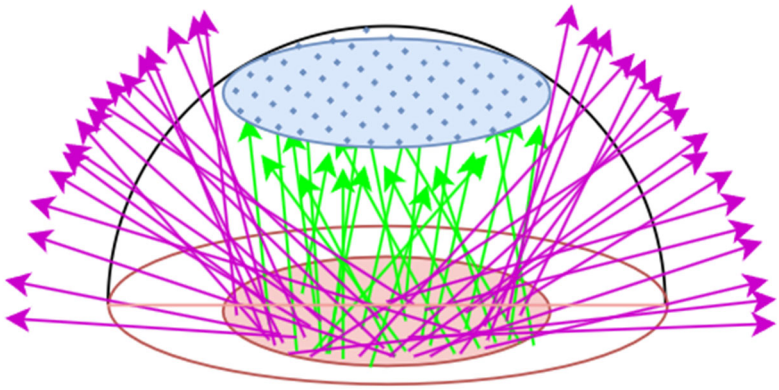

Figure 8. Monte-Carlo Ray Tracing with Finite Parallel Disks. MCRT can be performed between two parallel finite disks to approximate the view factor one disk has of the other.

MCRT also allows for shadowing of one surface upon another. That is, if the view of one surface overlaps another, the latter surface will have a lower view factor of the source surface. The rays shot from the source surface impact the first surface and do not travel to hit the second surface. Figure 9 displays an example created in MATLAB of a surface shooting blue rays out to hit two other surfaces above it, with rays that do not hit any surface removed for visual clarity. The middle surface blocks the rays shot from the lower surface, preventing the lower surface from seeing the full upper surface.

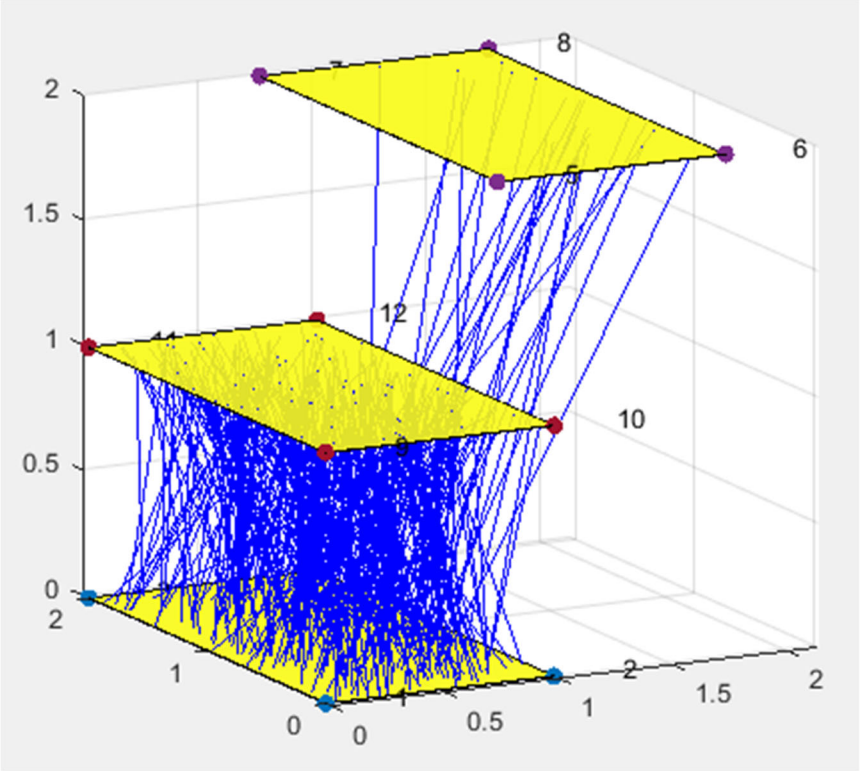

Figure 9. MATLAB Example of MCRT Shadowing. A set of three rectangular plates are used to demonstrate how MCRT can include shadowing in view factor calculations.

There also exists indirect radiative exchange, where the energy radiated from a surface k reflects off a surface j and then impinges upon the surface of interest, i (Figure 10). This happens because not all energy impinging upon a surface is absorbed, which is why ε is a fractional number unless considering an ideal case, in which it is unity. The reflectivity of a surface, ρ, is calculated as $\rho = 1 - \varepsilon$, and describes the proportion of energy that impinges upon a surface that is reflected away from the surface.

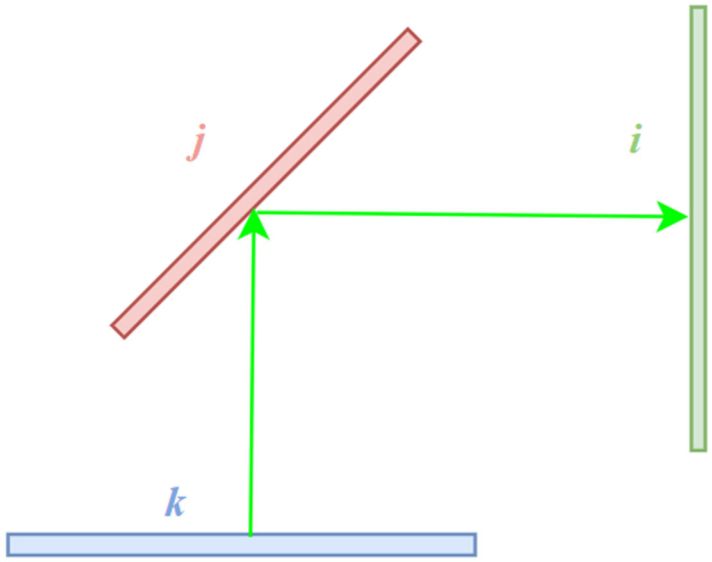

Figure 10. Reflected Ray. A ray is emitted from a surface k and reflects off surface j to impinge upon another surface i.

Within a closed volume, such as the interior of a spacecraft, radiation can be reflected numerous times given that the reflectance of each surface it impinges upon is high enough. When radiation is treated as a set of rays, the rays of radiation can be tracked as they reflect within the spacecraft. Each ray has some value of energy, and each time a ray hits a surface a proportion of this energy is absorbed by the impacted surface, as defined by the IR absorptivity of the surface. The ray then reflects away from the surface with its remaining energy and may impinge upon another surface to then lose more energy and reflect again. Eventually, the ray will lose enough energy such that it provides negligible impact upon the next surface it hits and essentially "dies." The direction of reflection is determined by how specular or diffuse a surface is. Specular surfaces are mirror-like, reflecting incoming rays of radiation at the same angle it came in but on the opposite side of the normal vector of the surface. Diffuse surfaces on the other hand reflect rays at random angles (Figure 11).

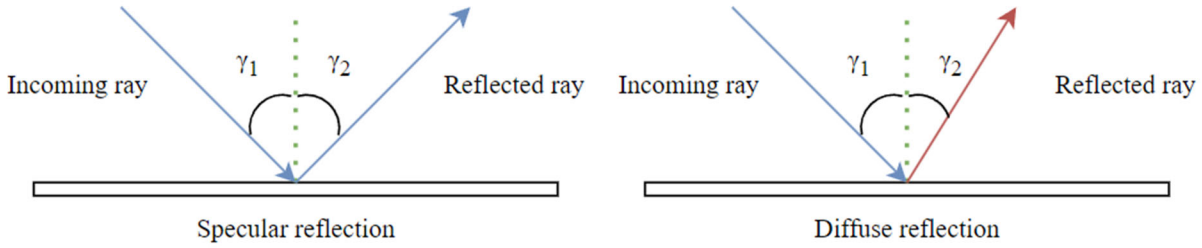

Figure 11. Specular and Diffuse Reflections. Specular surfaces reflect rays of light with an identical incidence angle while diffuse surfaces reflect rays at a nonuniform angle.

There are two ways of handling reflection calculations. The rays used in the MCRT view factor calculations can be treated as the rays of radiation. Instead of stopping as they hit a surface, they can reflect at some angle γ described by the diffusivity of the surface. They can then hit other surfaces, including the source surface. This is an accurate method for finding reflections, but it comes at the cost of being rather computationally expensive. Because the method requires the tracking of each ray's angle of reflection as well as each ray's local energy, it must be performed for several thousand rays.

The second method relies on the assumption that all surfaces within the system can be treated as perfectly diffuse surfaces, which means that any ray that hits the surface will reflect in a fully random direction. First, the view factors each surface has of every other surface are calculated with MCRT. Next, the assumption is made that the view factors approximate the random reflections of the rays. That is, if radiation hits a surface k, the reflection of the radiation will proportionally radiate to the other surfaces j according to the view factors k has of those surfaces. The energy emitted from a surface i can then be tracked as it reflects throughout the spacecraft by calculating the proportion of energy that leaves i and hits each other surface j, including through numerous reflections, in a matrix known as the Radiation Proportionality Matrix (RPM). The method to find the RPM is as follows:

1. Find the reflectivity vector and diagonalize. The reflectivity vector, ρ, is a list of the reflectivities of each surface. Diagonalizing ρ it creates a matrix, ρ', in which the diagonal is populated by the elements of the reflectivity vector and all other elements of the matrix are zero.
2. Find the proportions of direct energy transfer from surface i to surface j as $E_{ij} = \varepsilon_i F_{ij}$.
3. Each matrix describing proportion of energy transfer from a surface i to a surface $k, l, m, ...$ per reflection is described as:

$$E_{ik} = (\rho' F) E_{ij}$$

$$E_{il} = (\rho' F) E_{ik}$$

$$E_{im} = (\rho' F) E_{il}$$

$$\dots$$

   This can be performed as a while-loop to check if the maximum value of each matrix E is below some cutoff value prescribed by the engineer. This cutoff value determines when to "kill" off a reflection. Also note that each surface $k, l, m, etc.$ can be surface i. This is the proportion of energy emitted by *i* that reflects to itself.
4. The final matrix, RPM, is the sum of each matrix E.

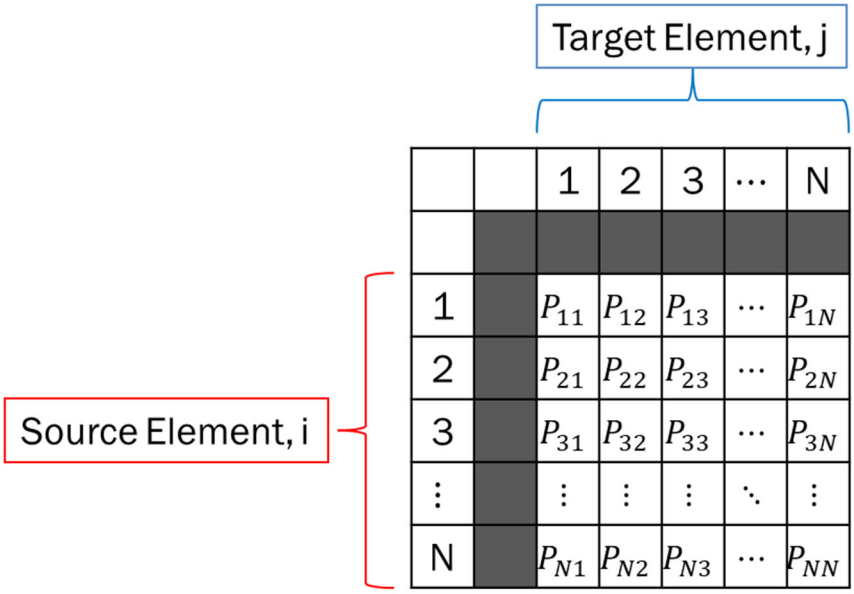

Figure 12. Radiation Proportionality Matrix. The RPM lists the proportion of radiation emitted by a surface i that hits a surface j.

The RPM describes the radiative exchange fraction between any diffuse surface i and any other diffuse surface j given that MCRT has been used to calculate the view factors between the surfaces (Figure 12). The radiative heat transfer rate from a surface i to any surface j is calculated as,

$$\dot{Q}_{ij,in} = \dot{Q}_{i,out} RPM_{ij} \tag{23}$$

2.3. Orbital Propagation

A spacecraft in orbit is constantly changing its orientation to other heat sources in the external environment. To simulate these dynamic conditions, an orbital propagator is necessary. The orbital propagator iteratively computes many parameters for a spacecraft at each time step while solving the N-body problem. This includes the position and orientation of the spacecraft in a specific frame of reference.

There are a variety of commercially available orbital propagators, each one with advantages and disadvantages. The selected a.i. Solutions' FreeFlyer mission analysis tool

propagates the spacecraft orbit and exchanges information with the thermal solver. The thermal solver requests four key pieces of information from FreeFlyer: the epoch of each time step (i.e., time at which the step occurs relative to January 1st, 2000); the position vector of the orbital body relative to the spacecraft (translated from the MJ2000 coordinate system to the spacecraft's body coordinate system); the position vector of the system star (in most cases, the sun) relative to the spacecraft (also translated from the MJ2000 coordinate system to the spacecraft's body coordinate system); and a Boolean value indicating whether or not the spacecraft is currently in the shadow of the orbital body. This information is sufficient to calculate incoming solar radiation, planetary albedo, and planetary infrared radiation that, taken together, represent the environmental boundary conditions that apply to the modeled spacecraft.

2.4. A Primer on Spacecraft Thermal Control

The subject to be analyzed is a spacecraft orbiting a planet in the space environment. External conditions affecting the spacecraft are highly dynamic. The spacecraft is heated by external sources and emits heat back into the environment via radiative means, as discussed in the prior sections. This section provides a primer into the considerations that may be may made to control the heat energy going into, leaving, and moving throughout the internal components of the spacecraft. Further background is provided into the setup of the thermal solver created for this thesis.

Critical components on spacecraft have specified temperatures in which they are qualified to operate and/or survive. Passive systems, such as insulation and finishes, as well as active systems, such as heaters and cryocoolers, may be used to control the thermal state of the spacecraft. Passive systems do not use electrical power to perform their functions. However, they may be

limited in how much they can move or prevent the movement of heat. On the other hand, active systems typically are powered by electrical energy, either to generate heat or move heat around.

Three examples of passive systems are insulation, surface finishes, and heat straps. Insulation is a material, or a set of materials, which have high thermal resistance, drastically slowing the movement of heat. This is often used on the exterior of spacecraft to slow the heating rate of the interior from external radiation. This is also used in cases to keep a component from losing heat too quickly. Surface finishes include paints and tapes that change a surface's emissivity, absorptivity, and reflectivity. The design might incorporate paint on the exterior of a spacecraft with high emissivity to help radiate internal energy out to deep space.

Alternatively, a paint with high reflectivity may be used to reduce the amount of incoming heat energy by reflecting most of it back out to deep space. Inside the spacecraft, paints are used to control the amount of energy exchanged between components. The interior of a radiator panel may be given a paint with low IR emissivity, for example, such that it does not reradiate heat within the spacecraft.

Heat straps are connective straps that allow for heat to flow from one end to another, for example from a hot component to a radiator panel, where it can be radiated out to space. They are typically made of highly conductive materials that allow heat to flow through them easily.

Heaters are an example of an active system that produces heat using electrical power. Resistive heaters are a simple type of heater that generates heat though the application of electrical current through a resistive material. Electrical coolers, cryocoolers for example, remove heat from a surface (Figure 13). For cryocoolers, this heat energy is absorbed by expanding an internal working fluid which is then compressed back into a liquid in a cycle releasing the heat into a

conduction path to a radiator on the spacecraft exterior [15]. Thermoelectric devices may either heat or cool, depending on the direction of current through the device.

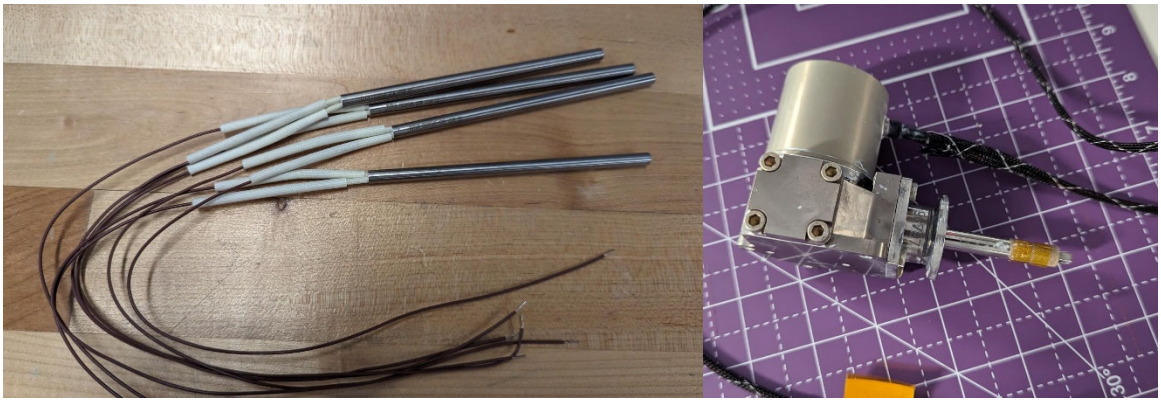

Figure 13. Resistive Heaters and a Cryocooler. Resistive heaters are one method for adding heat energy to a system while a cryocooler is a method to remove heat from a part of the system.

Both passive and active systems may be used to balance the spacecraft thermal environment. Thermal analysis tools can provide insight into the appropriate use of these systems. In the next section, the use of thermal solvers to perform this analysis is presented.

CHAPTER 3: THERMAL SOLVERS

A thermal solver is the mathematical formulation that iteratively calculates the temperature of a thermal network given boundary conditions at each step. As the conditions change between each step, closed form analytical methods become difficult to use. Instead, a thermal solver performs stepwise numerical calculations, propagating forward in time until either a constant equilibrium has been found or the simulation timer has run out. This section provides an introduction to thermal solvers to build a foundation of understanding for the later implementation.

3.1. History and Tools

Before the 20th century, the tools necessary to perform accurate analysis of complex thermal systems were not available. Simple techniques, including empirical approaches, were used to gather the needed data. Johann Amos Comenius, in the 17th century, described the functions of a rudimentary thermoscope which could measure three degrees above and below the ambient temperature [16]. Additional tools were developed over the following centuries, including thermocouples, thermoelectric pyrometers, and of course, the thermometer. With the creation of these devices, it became possible to gather heating data over time, allowing for differential thermal analysis (DTA). By comparing the difference between preceding and following time steps predictions of system performance could be made [17].

In the modern era, thermal analysis can be done on simulated systems. Numerical solvers, which can accept thermal networks as inputs, are used to output calculated temperatures and heat flows. Computational tools began development midway through the 20th century, starting with basic computing software and languages used to solve models of thermal systems. Throughout the rest of the century, these tools were built and expanded upon by companies such as ANSYS, COMSOL, and Siemens, incorporating greater capabilities and speed. Today, these tools have

been integrated with computer-aided design (CAD) software, allowing users to generate models within separate applications, and then prepare and solve the models with the linked thermal analysis software. Each tool is not entirely dissimilar in their underlying mathematics, having all been built upon the same principles. These such principles are described in the following sections.

3.2. Numerical Propagation

Heat transfer happens over a period of time, and as the energy moves, the rate of energy flow will also change. Equations of energy transfer are iteratively propagated with a fixed time step such that a new state is calculated for each proceeding point in time.

There are several different stepping schemes available for numerical propagation. The simplest method is known as the Forward Euler or the Runge-Kutta One-Step method. This is an explicit method, meaning that it takes currently known information about the system, then steps forward in time to find the next new state [18]. First, a scalar, first-order ordinary differential equation (ODE) must be defined,

$$\frac{dy}{dt} = f(t, y) \tag{24}$$

An initial state at time, $t = 0$, is defined as y_0. The value of y changes through each timestep n for $t > 0$ according to the ODE. The size of the timestep is noted as h, where $h = t_{n+1} - t_n$, which allows for the value of y at each time step, $n + 1$, to be found as,

$$y_{n+1} = y_n + hf(t_n, y_n) \tag{25}$$

This necessitates a known state y_n, which is why an initial value y_0 is required to begin the stepping process. A graphical representation is given below to demonstrate the change in temperature a node may experience over time (Figure 14).

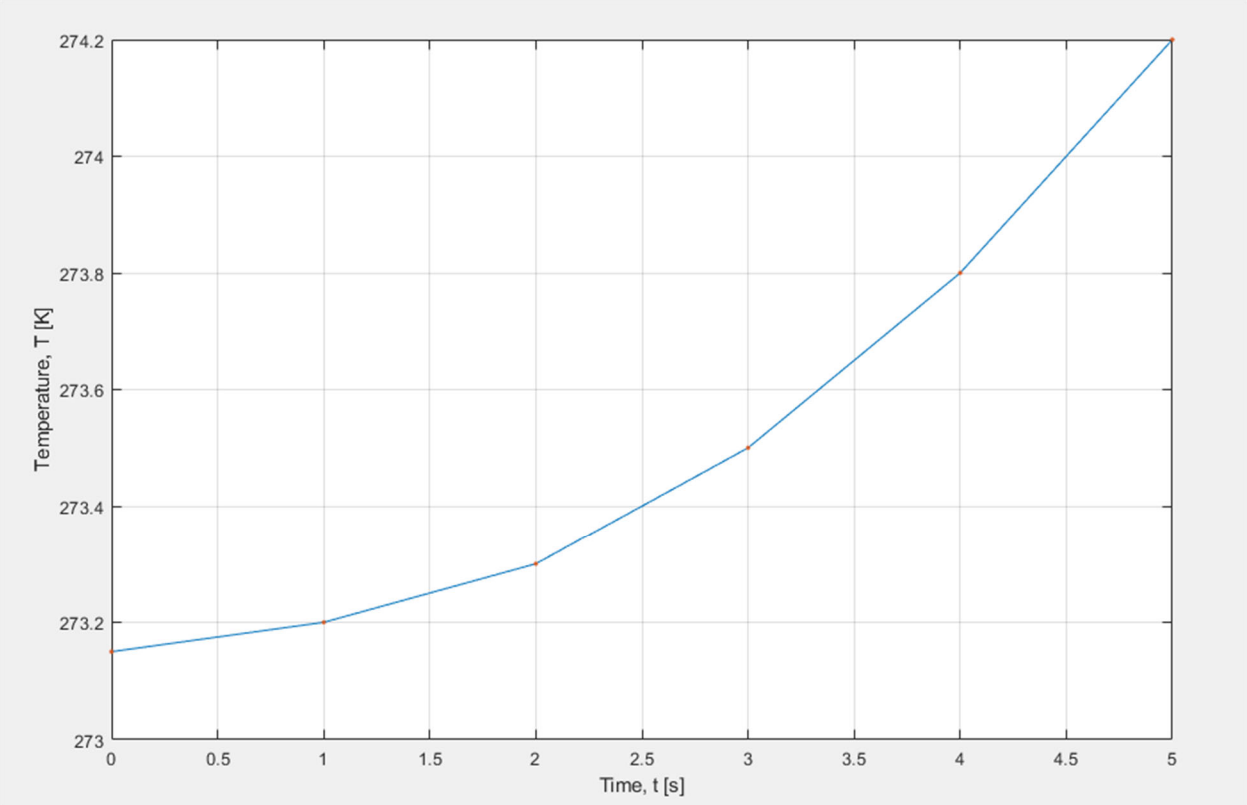

Figure 14. Graphical Representation of a Forward Euler Stepping Method. The Forward Euler numerical method can be used to compute the temperature iteratively given an ordinary differential equation.

The ODE, simplified as $f(t_n, y_n)$, provides the slope for each section of the plot at each time step. The total temperature at each new time step, y_{n+1}, is the sum of the current time step with the ODE applied across that time step. This results in a propagation of temperature over time.

Another explicit method for numerical propagation is the Runge-Kutta Four-Step (RK4) method. As noted in the name, this method takes four steps while the Forward Euler, also known as the Runge-Kutta One-Step method, only takes one step. The RK4 takes a weighted average of four increments which are themselves informed by the preceding increment, to provide a more accurate slope between each time step. Again, an ODE and an initial condition are defined in the same manner as for the Forward Euler, and then the following equation is followed [19],

$$y_{n+1} = y_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4) \tag{26}$$

$$k_1 = f(t_n, y_n)$$

$$k_2 = f\left(t_n + \frac{h}{2}, y_n + \frac{hk_1}{2}\right)$$

$$k_3 = f\left(t_n + \frac{h}{2}, y_n + \frac{hk_2}{2}\right)$$

$$k_4 = f(t_n + h, y_n + hk_3)$$

Note that if there are no values within the ODE, *f*, which are in themselves functions of time, the term $t_n + h/2$ can be ignored.

Both the Forward Euler and the RK4 methods are viable options for thermal propagation, though the Forward Euler has an accumulated error directly proportional to *h* while RK4 has a total truncation error on the order of $O(h^4)$. That is, the larger the time step, *h*, the greater the error. This is the same for the Forward Euler method, though since RK4 is of the fourth order, it will tend to have greater stability. However, RK4 will also take more computation time than the Forward Euler method as it requires four distinct steps before the iterative solve. This becomes a tradeoff in terms of solution time versus accuracy. Both have been implemented in the thermal analysis code written for this thesis, allowing the user to choose which method to use during the solve. The faster method may be used when exploring different thermal designs, while the more accurate method may provide final results for design verification.

MATLAB has several functions for numerical propagation with ordinary differential equations. These include *ode45*, *ode78*, and *ode89* to name a few. These functions perform the Runge-Kutta (4, 5), (7, 8), and (8, 9) methods respectively, which are variations on the higher order Runge-Kutta methods. These various ODE solvers can be highly efficient, solving quickly with good stability. However, these solvers cannot be used for the primary stepping of the thermal solver as there is no means of inputting orbit-dependent variables during the stepping. This means

that if *ode45* were to be used, the integrator will only intake Sun and orbital-body positions at the initial point in time from the orbital propagator and updating this throughout the integration is not practical. However, this does not mean that these ODE functions are unusable. These can still be implemented in intermediate propagation, where they are used to propagate between orbital steps when the orbital steps are large. For example, *ode45* can be implemented to step from the initial conditions to the end of the first timestep, then output the temperatures at that timestep and end the intermediate integration. The next step in orbital propagation can then be performed and the requisite information for thermal solving can be obtained. Following that, *ode45,* or whichever MATLAB ODE function is in use, can be used again to step forward in time. This essentially breaks every orbital time step into smaller steps, thus increasing the stability of the integrator over large time steps.

Going forward, the main goal is to derive the ODE that serves as the input for the numerical propagator. The time step, h, as well as the initial conditions, y_0, are values dictated by the user of the thermal analysis software.

3.3. Finite Element and Finite Difference Methods

Real life systems are complex. A satellite, a table, a building, a person, are all composed of molecules and bonds existing in a continuum. Unfortunately, it is not possible to simulate every molecule, keeping track of every unit of energy along the way. When dealing with complex systems, problems may be simplified through discretization, in which parts of a system are broken up into individual pieces that pass information back and forth. Thermal networks are an example of one type of system discretization, specifically used for solving thermal analysis problems.

The two most popular methods for discretizing thermal problems are the Finite Difference Method (FDM) and the Finite Element Method (FEM). FDM has a long history, having been used

for solving discrete problems since 1715, when it was introduced by Brook Taylor [20]. Finite difference involves the division of a surface into a number of points with uniform spacing. These points, also called nodes, have material properties associated with them that inform the solver as to how heat will flow from node to node. A difference approximation is made between each node and their connected nodes at each instance of time, allowing for step-by-step propagation of information throughout the network. The heat equation, a partial differential equation, is often used with FDM. This equation, discussed in Section 3.5, takes in information at two or more nodes to calculate the new information at each node, given that the distance between nodes and a value known as the thermal diffusivity of the material is known.

FDM is relatively simple to implement, but it can have issues with irregular geometry unless specific case handling has been added. Curved geometries are the greatest challenge, requiring special approximations or mapping to a flatter domain.

FEM can handle such cases more easily. FEM was first used in the 1960s, having been developed for the structural analysis of aircraft. It was then expanded into use for thermal problems and can now be found in numerous solvers including NASTRAN and ANSYS. With FEM, a domain is discretized into subdomains, known as elements, which are also partially defined by nodes. Information is transferred through the elements to the nodes with formulas defined for the varying shapes the elements can take. This allows for greater flexibility in geometry; however, it comes at a greater computation cost.

The thermal solver written for this thesis utilizes FDM. The systems that this solver is being applied to are primarily composed of regular geometries such as small spacecraft known as CubeSats. CubeSats are made of uniform units (1U) each being a cube with ten centimeters side lengths. The structures of these satellites are constructed mostly out of flat, rectangular plates,

which lend themselves well to analysis with a finite difference solver. Additionally, the implementation of conductance calculations, rather than the use of the heat equation, allows for more complex geometries to be handled, bypassing many of the issues FDM has with nonuniform geometry, as will be further discussed in Section 3.6.

3.4. Discretization and Building the Thermal Network

As noted above, the thermal network is the basic discretization of a system for use in finite difference or finite element analysis. The thermal network is composed of nodes and linkages by which the nodes pass heat flow information to one another. Each node represents the state of a physical system at a particular location. For instance, nodes may represent small, interfacing disks within an actuator. If the required fidelity is low, then these disks could each be represented by a node that contains information about the disk's mass, specific heat capacity, and temperature. The disks interface with one another, and therefore pass heat back and forth. In the thermal network, the nodes are connected with links of thermal resistance that represent the interface capability for heat exchange (Figure 15).

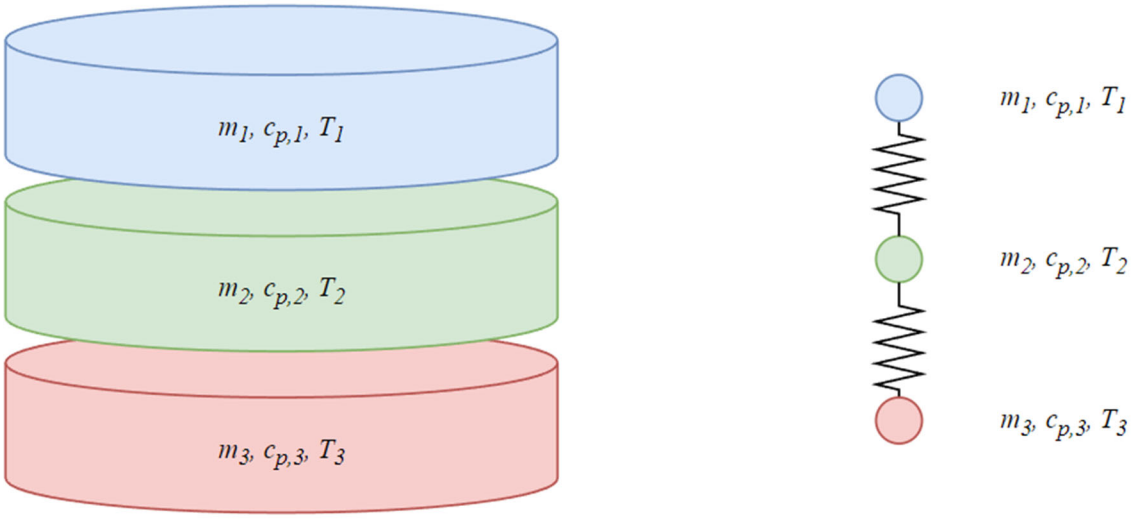

Figure 15. Disk Simplification. A series of disks can be simplified to a series of nodes, linked with resistive paths.

The thermal network is built upon this kind of simplification where the system is discretized as a whole. For higher fidelity, discretization is done within individual components, such as a thin rectangular plate. The rectangular plate can be parameterized with its own coordinate system, with the x_1, x_2, and x_3-axes defined by the straight lengths of the plate. It can then be subdivided, either with elements or by regularly spaced nodes as shown in Figure 16. These nodes and elements connect together, and just as before, pass information as part of a thermal network.

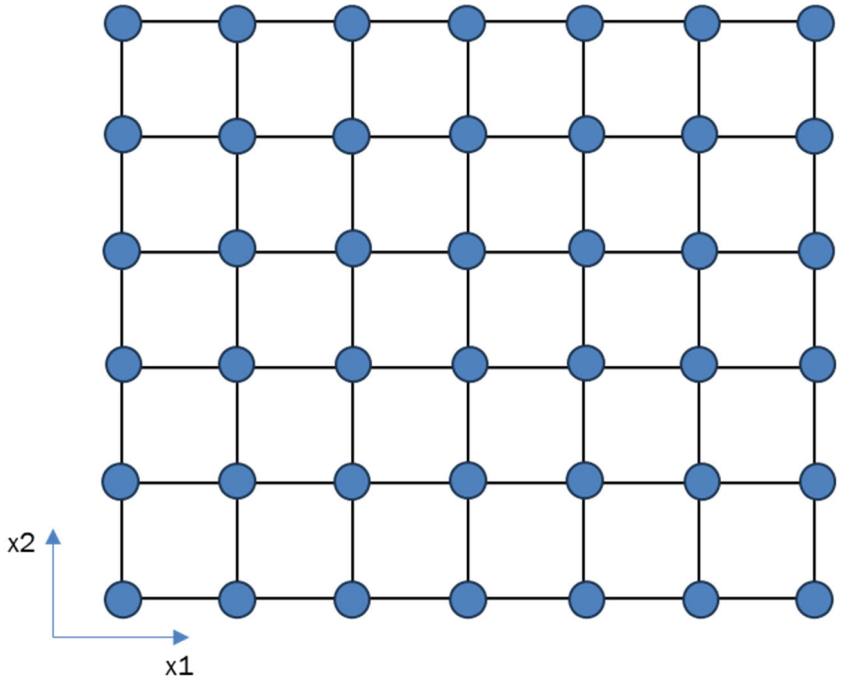

Figure 16. Discretization of a Rectangular Plate. A rectangular plate can be discretized into a meshed network of linked nodes.

Given that the plate is thin enough, further simplification can be made by defining only a thin sheet of nodes, also called a mesh. The thickness of the plate is no longer geometrically represented and is instead only taken into account within the mass at each node, depending on the density of the plate. Components with greater volume may need discretization in the third dimension.

All components of a system must be discretized using one, or both, of these methods. The components are linked together to form the full network in which each node informs or is informed by at least one other connected node. This requires significant effort to create networks that represent the physical system, though this effort can be assisted by the tools in use. For example, Siemens NX can create a model in CAD, then use the pre/post-processing application, Simcenter 3D, to discretize the geometry. The solver written for this thesis includes pre-processing tools that allow for grids of nodes to be created given a few inputs. This speeds up the building process of the thermal network as it automatically generates the nodes given the lengths of the surface the user wishes to create, as well as creates the resistive links between the nodes.

Thermal networks, though idealizations of a real-life system, can also grow to be complex depending on the fidelity of the simulation. The greater the fidelity, the more nodes and links must be created. This leads to more computation time as the solver must handle many more nodal calculations per time step. It is therefore necessary to understand the fidelity necessary to accomplish the analysis goal. However, it is cautioned that the user of any thermal analysis software be aware of the solution time increases exponentially with the density of the thermal network.

3.5. Heat Equation

The heat equation was developed by Joseph Fourier in 1822 to model the diffusion of heat throughout a system [21]. It is the most well-known method for solving heat transfer problems and is used often in FDM thermal analysis software. It is a partial differential equation that can be performed in *n*-dimensional space. For thermal analysis, this is limited to three dimensions at most since the systems dealt with either exist or are to exist in the real world. The general equation is stated as follows.

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x_1^2} + \frac{\partial^2 u}{\partial x_2^2} + \frac{\partial^2 u}{\partial x_3^2} + \cdots + \frac{\partial^2 u}{\partial x_n^2} \tag{27}$$

This gives the change in a system given the Laplacian of the function *u*, which forms the ODE input for numerical propagation. For a thermal case, a coefficient known as the thermal diffusivity of the medium by which heat is transferred is pre-pended to the formula. This value is typically represented by the variable, *α*, but to avoid confusion with absorptivity, thermal diffusivity shall be referred to by the variable *δ*.

$$\frac{\partial T}{\partial t} = \delta\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right) \tag{28}$$

Thermal diffusivity, *δ*, is defined as $\delta = k/\rho c_p$ with *k* being thermal conductivity, *ρ* being density, and c_p being specific heat capacity. The variable *u* has been replaced with *T* to show that this equation is specifically for the temperature case. Note that this equation utilizes Cartesian coordinates, taking inputs of discrete differences in lengths along *x, y,* and *z*. Formulations of this equation exist for other coordinate systems, but those are largely irrelevant here. Each term within this equation is found using the following formulas.

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_{past}(x_{i+1}) - 2T_{past}(x_i) + T_{past}(x_{i-1})}{dx^2} \tag{29}$$

$$\frac{\partial^2 T}{\partial y^2} = \frac{T_{past}(y_{j+1}) - 2T_{past}(y_j) + T_{past}(y_{j-1})}{dy^2} \tag{30}$$

$$\frac{\partial^2 T}{\partial z^2} = \frac{T_{past}(z_{k+1}) - 2T_{past}(z_k) + T_{past}(z_{k-1})}{dz^2} \tag{31}$$

To better understand these equations, it is perhaps easier to begin with a one-dimensional case, then step up from there. Figure 17 depicts two nodes along the *x*-axis for which temperature information of each is known for one step in the past, T_{past}.

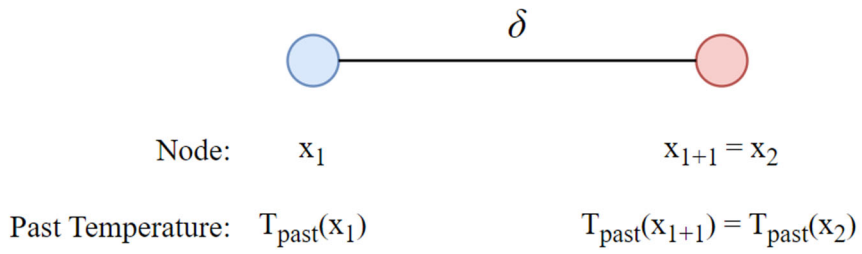

Figure 17. One-Dimensional Two-Node Network. A two-node network flows heat along the resistive path depending on the difference in nodal temperature at a time t.

Heat is passed between nodes x_1 and x_2 by the equation,

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_{past}(x_2) - T_{past}(x_1)}{dx^2} \tag{32}$$

The node x_1 is informed only by node x_2 and vice versa. Adding another node onto the end of this series provides the node x_2 with another linkage by which heat will flow through (Figure 18).

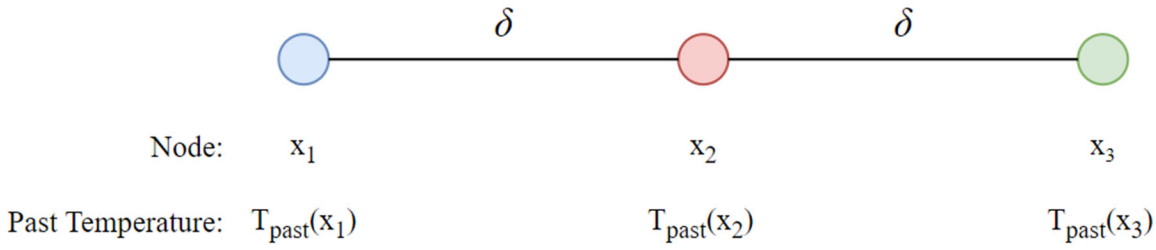

Figure 18. One-Dimensional Three-Node Network. A three-node network passes heat along two resistive paths depending on the differences in nodal temperatures at a time t.

For this series of nodes, the heat flow is characterized by two equations, in which heat travels between nodes x_1 and x_2, and nodes x_2 and x_3.

$$\left(\frac{\partial^2 T}{\partial x^2}\right)_{23} = \frac{T_{past}(x_3) - T_{past}(x_2)}{dx^2} \tag{33}$$

$$\left(\frac{\partial^2 T}{\partial x^2}\right)_{12} = \frac{-T_{past}(x_2) + T_{past}(x_1)}{dx^2} \tag{34}$$

Summing these two terms together gives the specific case of Equation 29. Given a larger network of nodes, such as in Figure 15, nodes may be connected in multiple dimensions, and therefore each term must be involved within the heat equation.

The mathematical form of the heat equation demonstrates why regularly spaced grids of nodes are preferred for FDM. The lengths and angles between nodes cannot be varied easily within this formulation as these fall along specific axes and take in constant values for *dx*, *dy*, and *dz*.

3.6. Conductance Calculations

Conductance has been touched on briefly in Section 2.2.1, but not fully explained in its use for heat transfer calculations. This section explains the use of conductance calculations within thermal solvers and why it is often more advantageous to use conductance rather than diffusivity to perform these operations.

Recall that conductance is the measure of how well heat flows from one node to another, whether the nodes represent separate parts of a single entity or separate entities altogether. This value is used as a coefficient, modifying another formula to determine heat or temperature change, much like the diffusivity seen in Equation 28. The major difference between calculations with diffusivity and calculations with conductance, however, is that calculations with diffusivity react to a change in temperature whereas calculations with conductance calculate rate of heat transfer [22].

$$\dot{Q} = \frac{\partial Q}{\partial t} = G\left(T_i - T_j\right) = G\Delta T_{ij} \tag{35}$$

Given that the conductance, *G*, between nodes *i* and *j* are known, either the heating rate $\partial Q/\partial t = \dot{Q}$ or the temperature difference $\left(T_i - T_j\right) = \Delta T_{ij}$ can be calculated knowing the opposing value. The form of this equation lends itself to matrix multiplication, allowing a vector

of heating rates to be calculated if *G*, a matrix of conductances between every node *i* and *j,* and $(T_i - T_j)$, a matrix of differences in temperature between these nodes, is already known.

This simulates the transfer of heat through conduction throughout a system. The total heat transfer between every node in a system includes the heat lost through IR heat rejection and the heat gained through environmental radiation. These are summed together with any additional heat loads, $\dot{Q}_{HL}$, and radiation exchange between surfaces, $\dot{Q}_{e2e}$, to produce a heat transfer rate,

$$\frac{dQ_{tot}}{dt} = \dot{Q}_S + \dot{Q}_a + \dot{Q}_{IR} - \dot{Q}_{out} + G\Delta T + \dot{Q}_{HL} + \dot{Q}_{e2e} = \dot{Q}_{tot} \tag{36}$$

This can then be substituted into Equation 5, which results in the following equation,

$$\frac{dT}{dt} = \frac{\dot{Q}_{tot}}{C_{th}} \tag{37}$$

This is the ODE that can be propagated forward in time using numerical methods, including those discussed in Section 3.2. The advantage this has over the standard heat equation is that it is far easier to visualize with a thermal network. The conductance between nodes is much easier to understand, as is the heat flow between the nodes that is based on energy transfer. Mathematically, it is also easier to form matrix multiplication formulas with this ODE, as *G*, ΔT, and C_{th} can all be written in matrix or vector form, where each row represents a node *i*. For the conductance matrix, *G*, the columns simply represent the nodes *j* that each node *i* connects to. This means that *G* is a symmetric matrix with zeros down the diagonal, as each node *i* will have zero conductance to itself. These conductance calculations are also easier to use with FDM as the values of *G* inherently hold the positional difference between the nodes. Additional vectors containing internodal information of *dx*, *dy*, and *dz*, as used in the heat equation, do not have to be formed and manipulated into a more complex ODE. The full implementation of the conductance calculations is further discussed in Chapter 4.

CHAPTER 4: IMPLEMENTATION

A thermal solver with basic pre/post-processing was written for this thesis with the intention of use on small satellites such as CubeSats. This thermal solver utilizes the theory and methods described in the prior chapters to solve temperatures of small spacecraft in orbit, given that orbital data is available from the linked orbital propagator. This solver utilizes the Finite Difference Method due to the simplicity and solution speed afforded by this method and requires the manual creation of a thermal network prior to solving. This section describes the implementation of the theory and methods in MATLAB, detailing the issues and solutions encountered throughout the process.

4.1. Building the Thermal Network

Pre-processing is a major step in thermal analysis in which the thermal network of a model is built. To do so requires the definition of nodes, surfaces, and conductances that adequately represent the structure that is desired to model. To facilitate this, several tools were developed, all as MATLAB functions to modularize the code as much as possible. This will allow for the future creation of a graphical user interface (GUI) to provide users with a visual understanding of evolving systems.

Recall that nodes are points that represent discretized pieces of a structure. A node is simply a point mass, holding information required by the solver to solve for the temperature at that point. Six key pieces of information per node are sufficient to carry out the solution:

1. Node ID
2. Thermal mass (C_{th})
3. Initial temperature (T_0)
4. X-coordinate (relative to spacecraft origin)

5. Y-Coordinate (relative to spacecraft origin)
6. Z-coordinate (relative to spacecraft origin)

All the information about nodes is held inside of a global matrix called MESHGRIDS_1, named in reference to thermal networks often being called “meshes” and nodes being called “grids” in NASTRAN. This matrix lists the above information for every single node, using the ID of the node additionally as the row index. This allows for quick access later when nodes need to be edited.

Two functions are used to create nodes. One directly creates nodes discretely, taking the direct input of each of the listed quantities and adding them to MESHGRIDS_1. The other creates surfaces, which are networks of connected nodes. The surface creation function takes in a different set of information as listed below:

1. Surface ID
2. Origin coordinates
3. Relative flag (switches between defining the surface relative to the origin of the surface or relative to the origin of the spacecraft)
4. x_1 vector of the surface
5. x_2 vector of the surface
6. Number of nodes in the x_1 direction
7. Number of nodes in the x_2 direction
8. ID of origin node

The surface creation function is meant to assist the user in building large rectangular surfaces of a spacecraft that should be discretized as such. Each surface is tracked with an identifier (ID) number for future manipulation and is defined by its origin and the vectors along two sides.

The origin determines where the rectangular surface begins and acts as the first node of the surface. The x_1 and x_2 vectors are local vectors that define the lengths and directions of the surface, determining the angle and size of the surface. As this function creates only rectangular surfaces, the x_1 and x_2 vectors must be orthogonal to each other. Each vector is then broken up into several nodes, ultimately creating a grid of nodes. Figure 19 depicts two such surfaces, one flat and one at an angle.

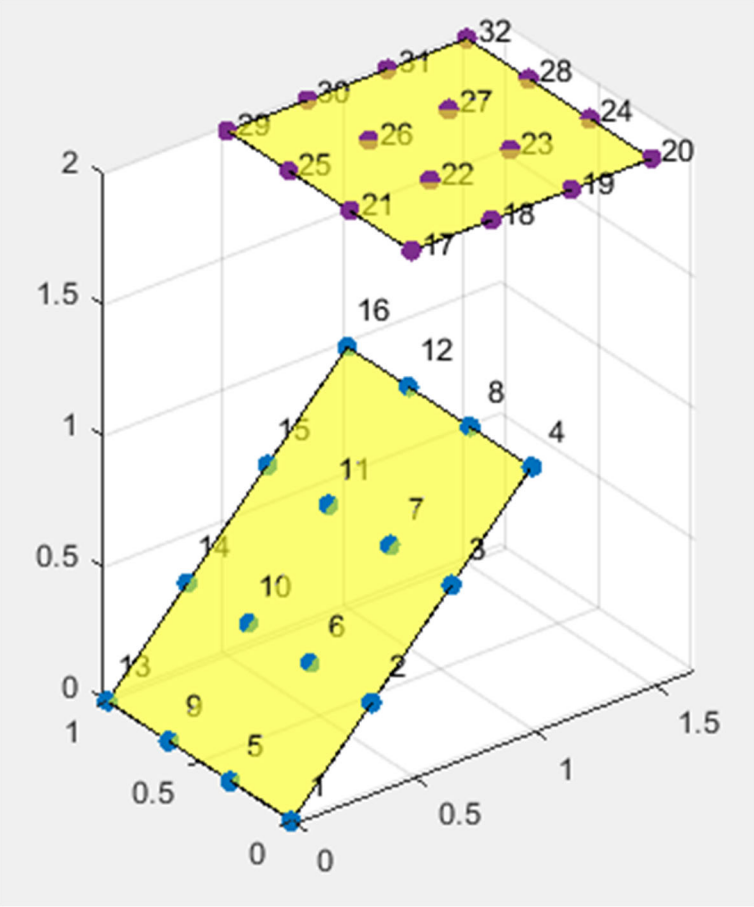

Figure 19. Surfaces Created. The surface creation function can be used to create surfaces of different densities, locations, sizes, and angles.

The surface creation function takes the ID of the origin node and iterates upward until each node has an ID number, starting along the x_1 direction of the surface and going up along the x_2 direction. The function also checks if each node already exists within MESHGRIDS_1 and ensures

that there is no overlap or overwriting of existent information unless the user specifies such. This is also performed for the surface itself, with its ID checked against a list of surfaces, held in a global cell array called SURFACES_2. These surfaces can be edited with another function, which finds all the nodes in the given surface and transforms them per user input. A similar function exists for editing individual nodes.

Each surface also has thermo-physical and thermo-optical properties assigned to them. These are first defined by the user in another function. Thermo-physical properties refer to a material's density, thermal conductivity, and specific heat. The user can list this data and add a name to identify it, which is then held in an array called THERMOPHYSICAL_1, and then assign the named thermo-physical property set to any surface. This is similarly done for thermo-optical properties, which refer to a material's solar absorptivity (α) and IR emissivity (ε) values. The array holding data for each named thermo-optical property is called THERMOOPTICAL_1. Different sides of a surface can also have differing thermo-optical properties, and so when the user assigns a property to a surface, these are held separately within another cell array called SURFACES_1.

Information is held in five distinct arrays. First, there is MESHGRIDS_1, which holds identifying information for every node in the model. It includes the ID, the thermal mass, the initial temperature, and the coordinates of each node. At this point, the thermal mass and the initial temperature are currently unassigned with placeholder NaN values, requiring later functions to fill them out later. There may also be gaps between nodes, where node IDs have been skipped. MATLAB automatically fills those rows with zeros which allows for easy pruning later. There also exists THERMOPHYSICAL_1 which holds the density, thermal conductivity, and specific heat of each named material in the model, which may not necessarily be assigned to a surface. THERMOOPTICAL_1 is treated the same, holding the solar absorptivity and IR emissivity of

each named property. These properties are assigned to the surfaces in the cell array, SURFACES_1, where they can be further manipulated later. SURFACES_1 also lists the thickness of the surface that allows for volumetric definition given the area of the surface. SURFACES_2, the other array for surfaces, holds more detail defining the surface itself, listing the nodes in the surface, the vectors defining the surface, the number of nodes along each vector, and the area of the surface.

The two sets of undefined values in MESHGRIDS_1 are assigned following the creation of the above cell arrays. Given the information listed in SURFACES_1 and SURFACES_2, the volume of each surface can be calculated as area multiplied by thickness. This is then multiplied by the density of the surface and the specific heat to get the total thermal mass (units of J/K) for each surface. These surfaces are defined with nodes on edges and at corners, and as such cannot have the mass uniformly distributed throughout every node making each surface. Consider the surface/grid of nodes depicted in Figure 20.

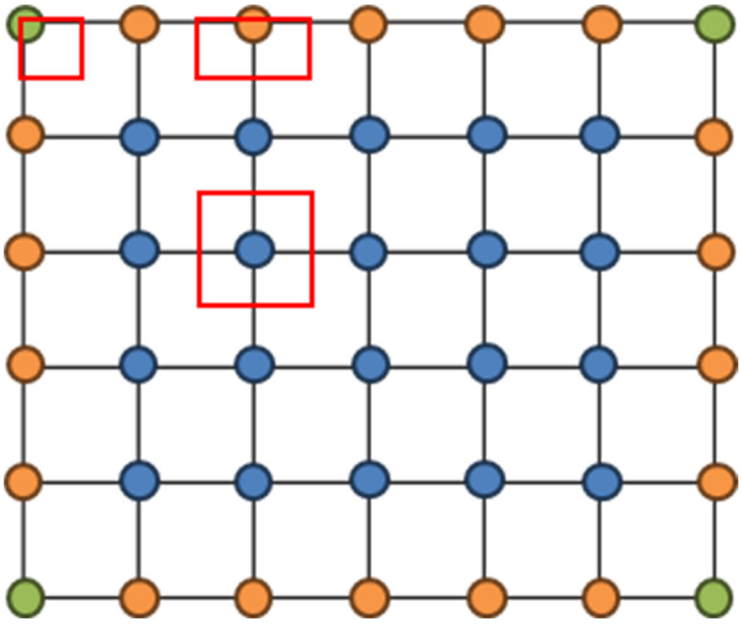

Figure 20. Corner, Edge, and Internal Nodes. Nodes in different locations on a surface may have differing nodal areas. The red rectangles represent the nodal areas associated with the corner, edge, and internal nodes.

The green nodes are corner nodes, the orange nodes are edge nodes, and the blue nodes are internal nodes. Each blue node represents the volume of material in a square around it (depicted as the larger red square in the image), composed of four quarter-units of volume. Each orange node represents half of that, with only two quarter-units of volume. The green node represents half again, with one quarter-unit of volume. This shows that the internal nodes are weighted four times as much as a corner node, while edge nodes are weighted twice as much. Given the total mass of a surface, the mass should be divided by the number of nodes in the surface and then by four to find the mass of a single quarter-unit. This is the thermal mass of each corner node and is then assigned to these nodes within array MESHGRIDS_1. Edge nodes are identified and assigned twice the value, and similarly internal nodes are given four times that value. This is performed for each surface until all nodes have their appropriate thermal mass. Nodes that were created with direct input do not need this surface-to-node thermal mass assignment as they were initially created with defined thermal masses.

The other set of undefined values, the initial temperature of each node, is assigned through another function that takes in the list of nodes or list of surfaces to which the user wishes to assign a given temperature, and then finds the nodes in MESHGRIDS_1 and inserts the temperature value. It is important to remember that this temperature is written in units of Kelvin.

Though this solver utilizes FDM, elements must still be defined to allow for granularity in temperature visualization and flux-to-heat calculations. This is performed after all surfaces are defined by the user to enable completion in one full step. Elements are defined by four nodes each, starting with the bottom-left node and then rotating counter-clockwise. These are referred to as nodes *a*, *b*, *c*, and *d*, throughout the MATLAB code for convenience. For these surfaces, each element is rectangular in shape, with uniform area throughout each surface. This means that each

element is regular and will conduct similarly to one another within each surface. A function is used to generate the elements array called ELEMENTS_1 which lists the surface each element belongs to, as well as the IDs of the *a*, *b*, *c*, and *d* nodes making up each element.

Defining the elements presents a small challenge in that their node IDs are not guaranteed to be consecutive. To circumvent this problem, only one surface is considered at a time for elemental division. The bottom-left node of the surface is the first node in the list of nodes held for the surface within the array, SURFACES_1. The nodes are listed in the order of left to right, then bottom to top, without regard for the actual ID of the node. Each element can be defined iteratively starting with the element's node *a*, then adding one to its index to get node *b*. Node *d* is at the index value of the index of node *a* plus the number of nodes along the x_1 direction and node *c* is at the index of node *d* plus one. Figure 21 depicts the indexing method for defining the nodes in the element.

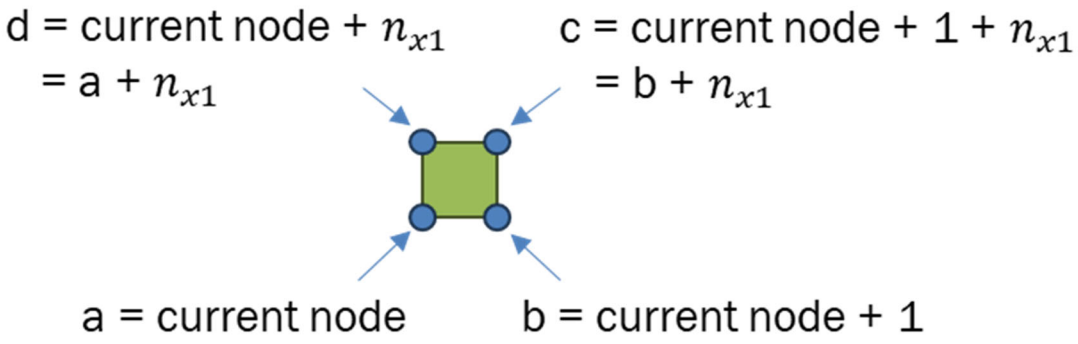

Figure 21. Element Node Indexing. The indexing method for nodes within an element starts from the bottom left and increments counterclockwise.

This is performed for each element, stepping to the next element in the surface once the previous element is complete. This allows for elements to be connected along their edges as they share nodes with their neighbors. Care must be taken at the right and top edges of the surface as using the far right or top nodes as the first node of an element will produce nonexistent elements.

The iterative element definition loop must therefore check for these nodes and prevent the creation of elements from them.

Created elements can be visualized with gradient temperatures using MATLAB's *fill3* function. This function takes in the x, y, and z coordinates of each node in counterclockwise order and produces a patch to fill the inside 3D space. A fourth input for each node allows for the definition of colors throughout the patch. Combining this with MATLAB's *quiver* function allows for quick visualization of the temperature gradient through an element along with the element's normal vector. Figure 22 shows an element with a sample gradient for values of 0 to 1. A vector, created with *quiver*, shows the normal vector of the element.

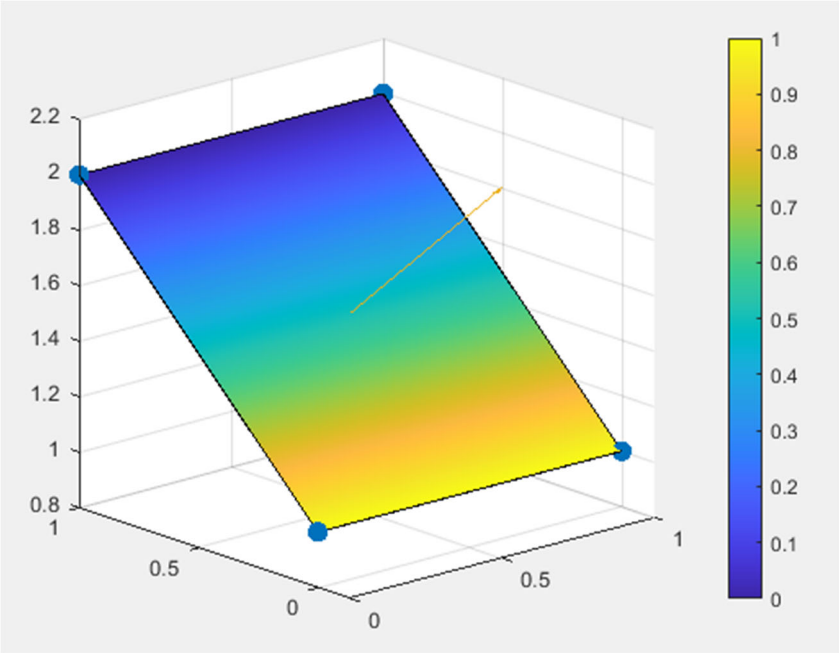

Figure 22. Rectangle with Simple Gradient and Normal Vector. An example of a rectangle with a simple color gradient and a normal vector is created using MATLAB's fill3 and quiver functions.

Another useful tool can be created with these colored patches. Sometimes it is difficult to determine which side of an element or surface has been defined as the top and which is the bottom. By creating a second patch offset slightly beneath each element of a different color and with lower opacity, the user can easily make this identification, as shown in Figure 23.

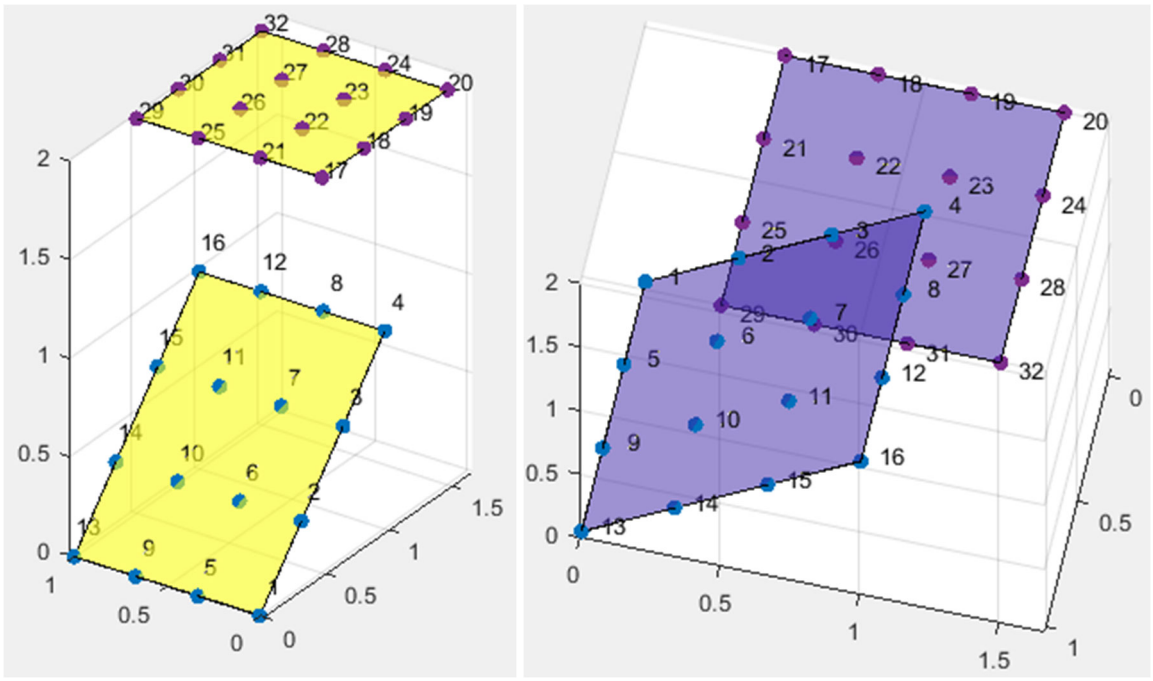

Figure 23. Top and Bottom Sides of Surfaces. The top and bottom sides of surfaces can be visually differentiated with the fill3 function.

By default, MATLAB's color set (*colormap*) is set to "parula." This is useful for initial visualization of elements but can be changed by the user. Later, during temperature visualization, the color set is set to "jet" which produces a blue-to-red color gradient. "Turbo" is another viable color set for temperature visualization, and is similar to "jet." Table 1 lists these useful colormaps and depicts examples of their hues [23].

Table 1. MATLAB Colormaps. MATLAB has several default colormaps that may be useful for visualizing temperature gradients.

Colormap Name	**Color Scale**
Parula	
Jet	
Turbo	

Internodal lengths (i.e., distances between nodes) are needed for calculations whether the heat equation or the conductance equation is used. This is a relatively simple endeavor when considering rectangular surfaces. Knowing the dimensions of the surface and the number of nodes in each direction, the lengths between pairs of nodes can be found with a division. Alternatively, the norm of the difference in coordinates between the two nodes produces the same result. This only needs to be done for nodes that will pass heat to each other. That is, nodes that will be connected with a measurable thermal resistance (Figure 24). As the surface has been divided into elements, each side of an element represents a connection between nodes. Note that the diagonals through the element are not considered for heat transfer. The result is a matrix of internodal lengths held as the matrix, SURFACELENGTHS_1, in which the internodal lengths between nodes that pass heat are defined and all others are set as Infs. Infs are used instead of NaNs as conductance is calculated with internodal length in the denominator. Conductance between nodes that do not connect will be found as approaching zero.

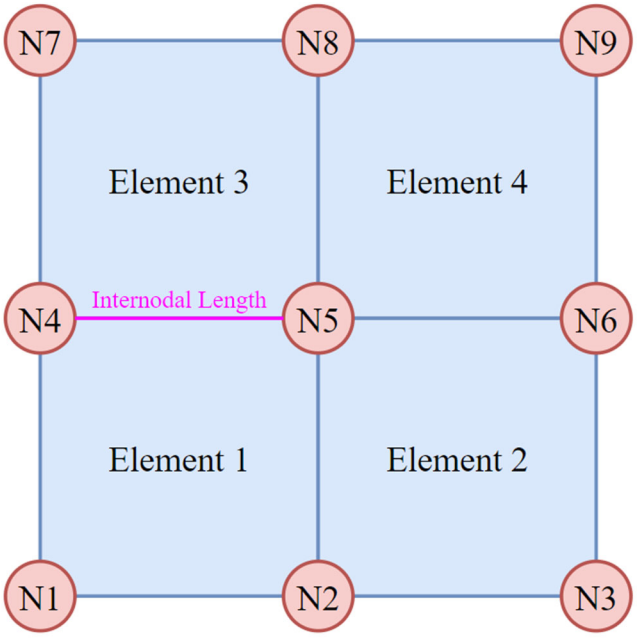

Figure 24. Internodal Lengths. An internodal length between two nodes (N4 and N5) is shown in pink.

An internodal cross-sectional area is also necessary for conductance calculations within surfaces. This is the two-dimensional area between nodes, representing the area through which heat will conduct.

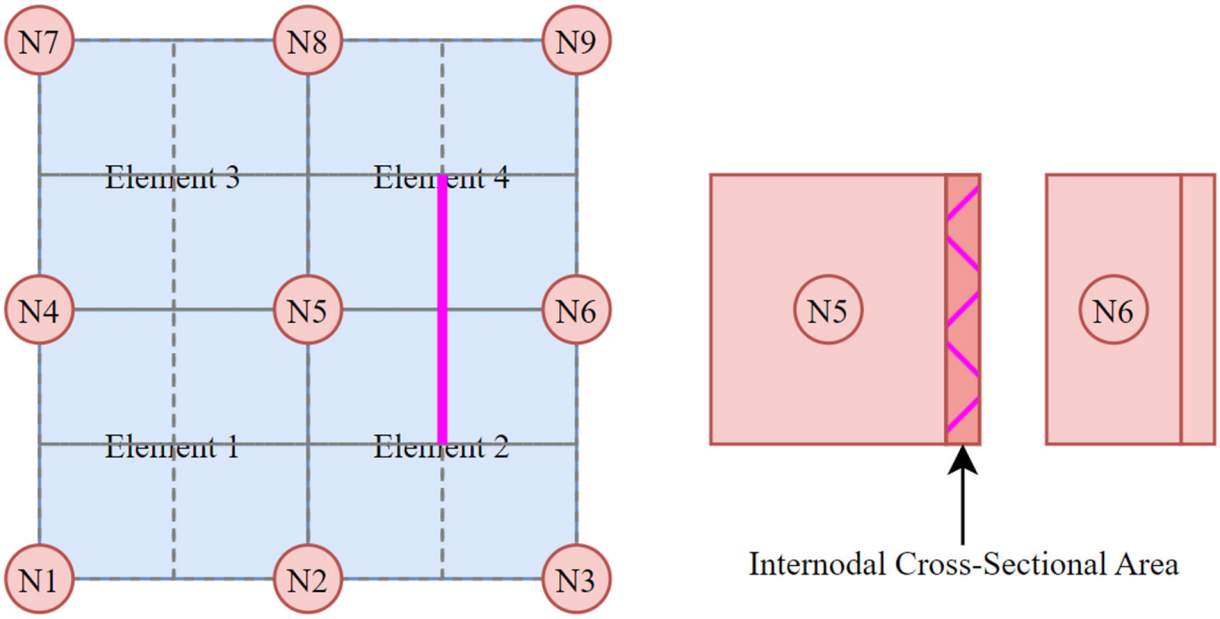

Figure 25. Internodal Area. The internodal area between two nodes (N5 and N6) is shown with a cross-sectional view.

Figure 25 depicts a surface composed of nine nodes and four elements. Each element can be subdivided into quarter-units. Node 5 (N5), in the center of the surface, represents the volume equivalent to four quarter-units of volume while Node 6 (N6), on the right edge, represents the volume equivalent to two quarter-units of volume. There is a cross-sectional area between these two volumes, designated in pink, with a width of two quarter-units and a thickness equal to that of the surface. The cross-sectional area is calculated similarly between each pair of conducting nodes, with the width being modified if the pair of conducting nodes is on the edge of the surface.

The internodal lengths and cross-sectional areas are now known between each node with the user-defined surfaces. The conductivity of each surface is also a user-defined value, captured within THERMOPHYSICAL_1 and SURFACES_1. This allows for the conductance to be

calculated between every node in each surface, following Equation 35, which is then stored in a holding matrix called CONDUCTANCES_1. Note that this discussed method is only for conductance within surfaces. Different operations must be performed to create conductances between lone nodes and between surfaces.

A "conductor" is a conductance link created between a singular node and a set of other nodes, separate from the conductance assignment performed within a surface. Conductors are necessary to ensure that lone, individual nodes are connected to the thermal network, otherwise they will not have any heat propagation to or from them. Conductors are implemented as another function, in which the user can select the source node (the singular node) and a set of other nodes. This set of other nodes can additionally be represented as elements, surfaces, or edges of surfaces. To allow for this, the function requires the user to input the IDs of the entities in the set, whether those be node IDs, element IDs, or surface IDs. The user then inputs the type of the target with the code described in Table 2.

Table 2. Conductor Inputs. The target type of a conductor can be defined with the listed codes.

Code	**Entity Type**
1	Node
2	Element
30	Surface
31	South Edge of Surface
32	East Edge of Surface
33	North Edge of Surface
34	West Edge of Surface

A conduction value is needed to define the conductance between the source node and the set of targets. This conduction value can be given as conductivity, thermal area conductance, or total conductance. If conductivity or thermal area conductance are given, then an area must also be defined by the user. This is the internodal cross-sectional area needed for the calculation of total conductance. For further flexibility, the user can also flag the input to be split amongst the targets (i.e., dividing the input conduction value by the number of targets) or to have it identically represented amongst all the targets.

Ultimately, a conductance value is calculated between the source node and each of the target nodes using either Equation 6 or Equation 8. These values are stored in a cell array called CONDUCTANCES_2, which holds identifying information about the source node, the target node, the conductance between the nodes, and the length between the nodes. The conductance values in CONDUCTANCES_2 are added to CONDUCTANCES_1 later but are initially stored separately so a user can edit the conductor without having to undo the changes to CONDUCTANCES_1.

A similar function creates “contactors” which links two surfaces or edges. A source element or surface is identified, as is a target element or surface, and then a conduction value is specified in the same manner as before. The input of area is slightly different in that the area can be taken as the area of the target surface, the area of the source surface, the area of the smaller surface, or an area specified by the user. The area of the smaller surface is often used, as when two surfaces are in contact, the smaller of the two defines the actual faying area.

The nodes of the surfaces may not be directly overlaying each other which calls for a method to handle surface to surface node connections. A search range is defined by the user within which each node from the source surface will search for nodes of the target surface. This works by

simply looking through the internodal lengths already calculated in SURFACELENGTHS_1 and checking if any of the nodes within the target surfaces or edges have an internodal length to each source node that is less than or equal to the defined search range. Any target nodes within that range can then be linked to the source node in question (Figure 26). This is repeated for each node of the source surface. In some cases, the user may wish to only have the source node link to one target node. To do so, the user sets a Boolean flag which tells the code to select only the node with the lowest internodal length. If there are multiple target nodes within the search range that have the same internodal length, then the target node with the lowest ID number will be linked to the source node (Figure 27).

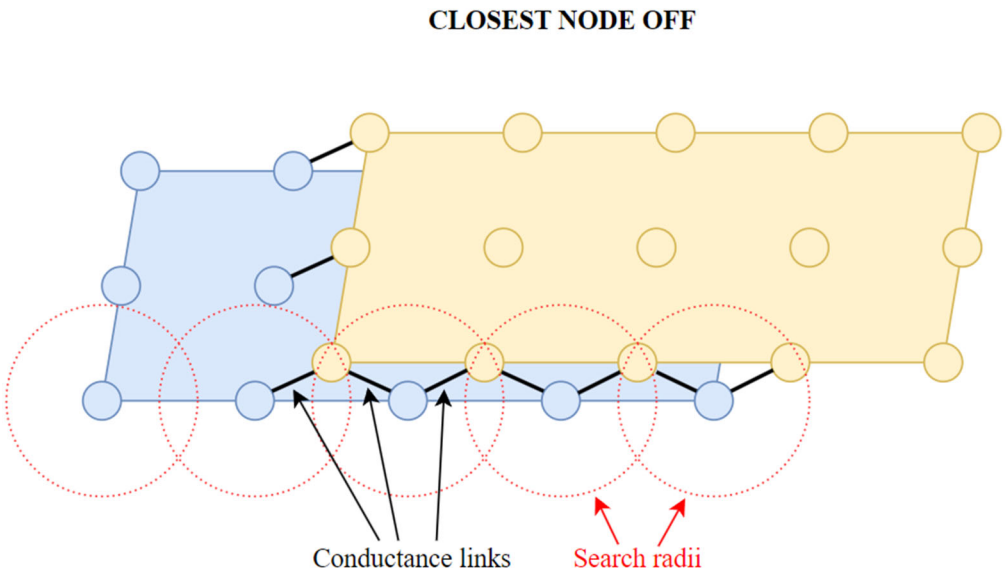

Figure 26. Contactors without Closest Node Flag. Contactors create conductance links with any nodes of a differing surface if the nodes are within the search radii.

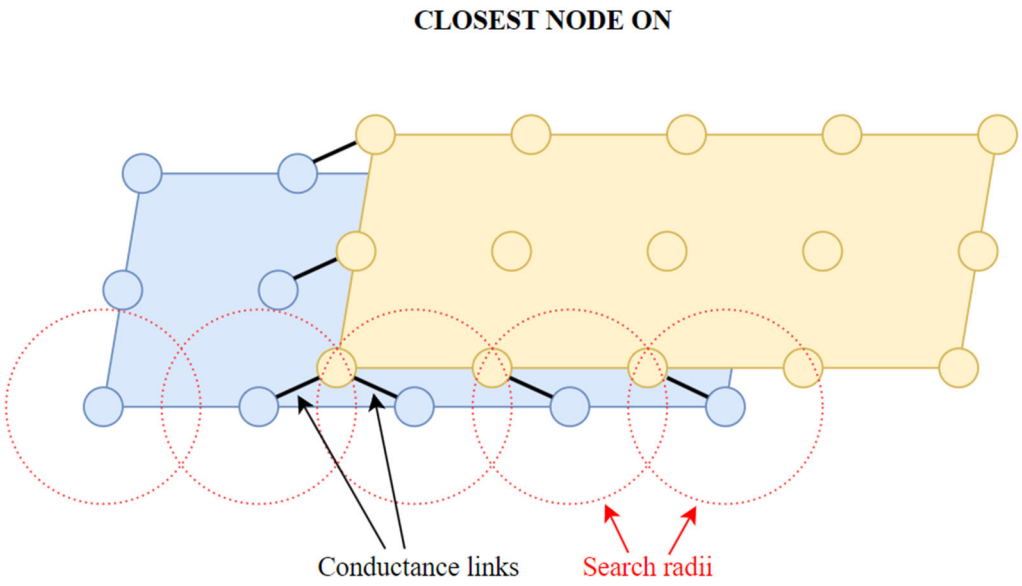

Figure 27. Contactors with Closest Node Flag. Contactors create conductance links with the closest nodes of a differing within their search radii. If multiple nodes are at the same distance, the node with the smallest ID is kept.

Surface-to-surface contact is created with contactors at times when the two surfaces are separate components. For example, two walls of a satellite may meet and will conduct heat to one another. Monolithic entities can also be represented, in which the two surfaces represent a singular component such as an L-bracket. Conductors and contactors can represent this but do introduce a small amount of thermal resistance at the joint which does not accurately represent the real component. In these instances, the surfaces need to be "merged" into one part. This is done at the nodal level, in which nodes from one surface are set to be shared with another surface.

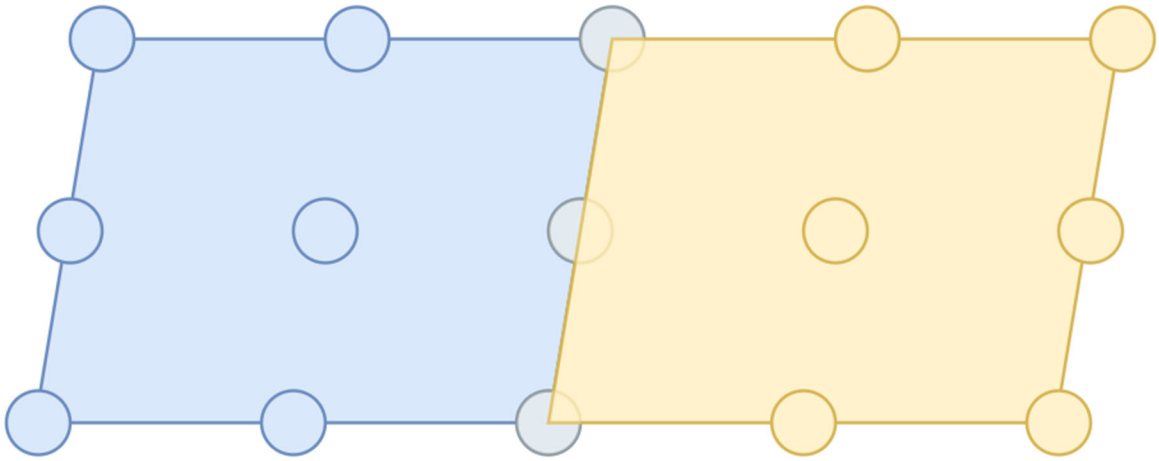

Figure 28. Merging Nodes. Two surfaces with overlapping nodes can have their nodes merged to share nodes with each other.

The two surfaces must have overlapping nodes. Figure 28 depicts an example of two surfaces intersecting along one edge. At this edge are six nodes: three from the blue surface and three from the yellow surface. When the merge function is used, three of the nodes will be subsumed by the other three, transferring over all relevant nodal information including thermal mass and conductance links. This also redefines both surfaces to ensure that both use the remaining three nodes, replacing the eliminated three within their node lists if needed. The nodes that are eliminated are either the nodes with the higher ID or the lower ID depending on the user input. Merging nodes retains all previous conductance values and preserves the surface information. Mass at these nodes is additive, ensuring that the total thermal mass of the model does not decrease with each merge. The primary result is that both surfaces are now fully linked by the shared nature of the nodes. Figure 29 displays an example of two perpendicular plates with merged nodes in which a heat load was applied to the one plate. The heat spreads continuously between the two surfaces through the shared nodes.

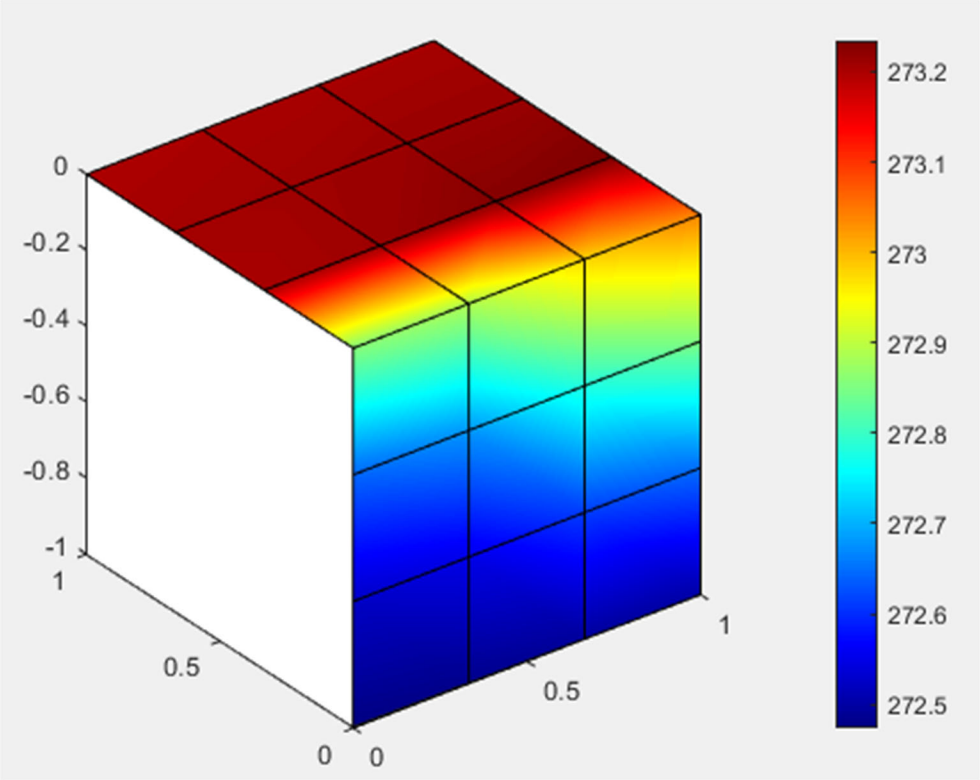

Figure 29. Result of Merging Nodes. Two plates with merged nodes will produce a continuous gradient as heat spreads through the system.

The penultimate pre-processing step is defining radiation groups. Radiation groups are sets of entities that radiate heat to one another and/or interact with the environment. These are used to partition radiation calculations, allowing radiation to be calculated within sections of a spacecraft. This is useful for increasing processing speed as not every surface will be able to radiate to every other surface. For example, the solar cells on the outside of the spacecraft are unlikely to perform surface-to-surface radiative heat exchange with a component on the inside. Also, most internal components will not experience direct solar radiation, so defining internal radiation groups that do not perform direct solar radiation calculations can again save processing time. Essentially, splitting surfaces, and even sides of surfaces, out to separate radiation groups, allows radiation calculations to be simplified, being performed only when there is actual radiation heat exchange.

Radiation groups are defined by adding one surface to the group at a time, or even one side of a surface at a time if greater detail is needed. The function for creating radiation groups takes five inputs from the user:

1. ID of the surface.
2. Flag indicating if the top side, the bottom side, or both sides of the surface are being added.
3. The ID of the radiation group that the surface/side is being added to.
4. The role of the surface, referring to its capacity to perform radiative heat exchange to other surfaces/sides in the radiation group.
5. A flag indicating if the surface/side should calculate incoming environmental radiation. This is referred to as the "space node" flag.

By allowing each side of a surface to be defined separately, the sides of a single surface can be placed within different radiation groups. This becomes especially useful for the walls of a spacecraft where the exterior side of the surface will experience environmental radiation and the internal side will not. Sides and surfaces can also be referenced in multiple radiation groups if needed.

The role of a side or a surface refers to its capacity to perform radiative heat exchange with other entities within the radiation group. There are three states that can be set: "ignored," "active and blocking," or "blocking." Ignored means that while the surface/side is within the radiation group, it does not radiate to other surfaces in the group. This can be useful for exterior surfaces/sides where the user may care only about environmental radiation and not surface-to-surface heat exchange. The active and blocking state means that the surface/side performs radiative heat exchange and blocks radiation from passing through the surface/side. The blocking state only performs the latter function. Figure 30 depicts a simple satellite composed of five surfaces, four

forming the walls of the spacecraft and one forming a blocking plate within the spacecraft. There is a rectangle labeled “A” in the spacecraft, which is part of a radiation group called “Radiation Group 1.” This rectangle is a blocker within its own radiation group but cannot block the radiation passing between the walls of the spacecraft as those are part of a separate radiation group. The blue surfaces/sides, however, are part of a single radiation group. The interior blocking wall prevents some radiation from exchanging between the left and the right walls of the spacecraft.

The walls of the spacecraft are split into their interior and exterior sides, shown in blue and orange, respectively. The blue sides, as already mentioned, perform radiative heat exchange to one another, but are partially blocked by the interior fifth surface. The orange sides are within a separate radiation group that can see the “space node” which represents the environment. These sides have their “space node” flag turned on, meaning that they can experience environmental radiation, and they are also set to active and blocking, meaning that they can reject heat out to space.

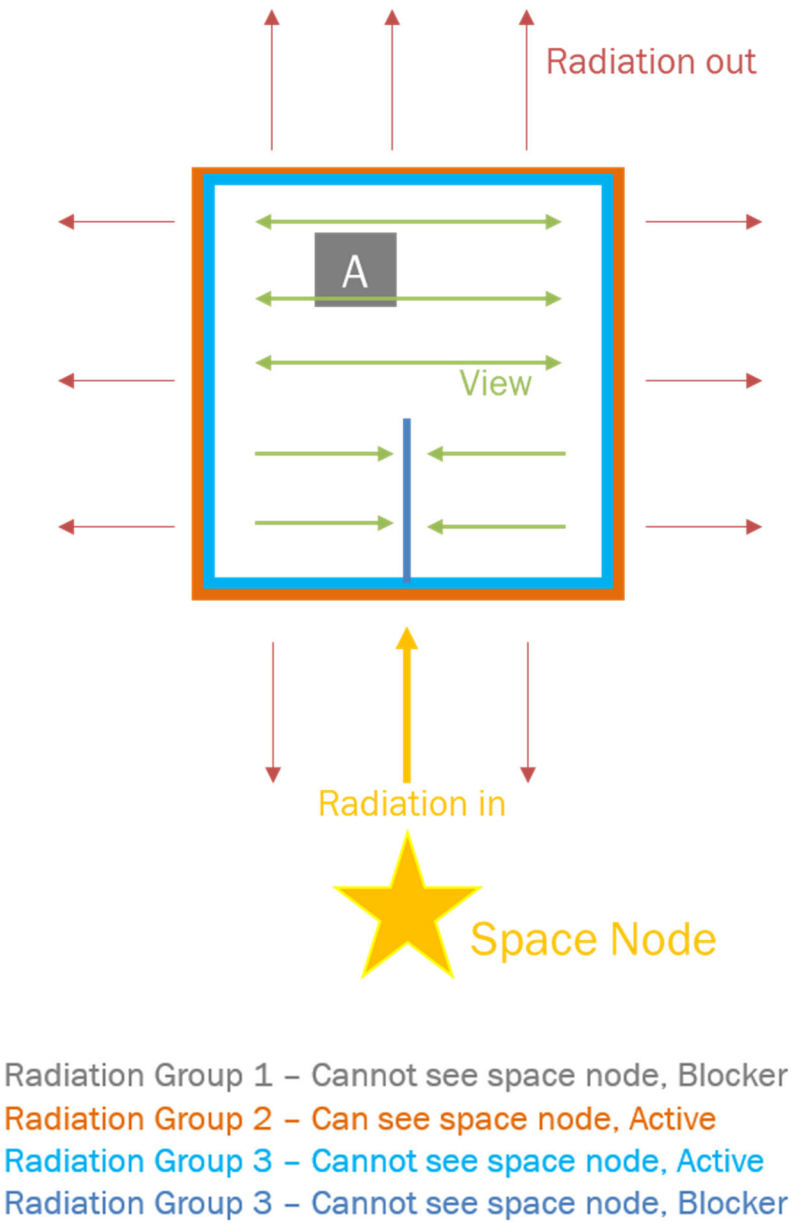

Figure 30. Radiation Groups in a Spacecraft. Radiation groups can be used to define which surfaces/sides of surfaces can see each other, radiate to space, or receive environmental radiation.

Radiation group information is stored within the SURFACES_1 cell array for later use. Boolean operations are used later to identify surfaces for which preliminary radiation view factor calculations must be performed, relying on the defined radiation groups.

The last step of pre-processing is to define heat loads. This is easily done by identifying the nodes to which the user wishes to add a constant heat input in units of Watts. A vector with the

same number of rows as MESHGRIDS_1 is created, listing the heat at the index of the node to which the heat is applied. This is then passed to the solver for constant heating throughout an orbit.

Once the thermal network is complete, the data is written to a text file called a "bulk data file." This file consists of the information that does not change throughout the iterative solution (e.g., node IDs, coordinates, conductances, etc.), which is useful when the user wishes to use the same thermal network for differing simulations. The bulk data file is loaded into the solver in such cases, allowing the user to skip the lengthy pre-processing step. If needed, the user can even edit the bulk data directly using a text editor.

The bulk data is written in sections, listing an overview of the data, and then providing detailed information created by the pre-processing functions. The overview simply lists the number of nodes, elements, surfaces, thermophysical properties, and thermo-optical properties which allows for post-processing functions to parse the following data more quickly. Each section is preceded with the name of the section and ends with "END OF SECTION," allowing the parser to read each section separately. Table 3 lists the sections and the data within each section.

Table 3. Bulk Data File Sections. This table lists the various names and details of the sections in the bulk data file.

Section Name	**Description**
OVERVIEW	Provides an overview of entity counts in the model. This includes the node count, the element count, the surface count, the thermophysical properties count, and the thermo-optical properties count.
NODES	Lists a truncated version of MESHGRIDS_1 in which the rows consisting fully of zeros are removed. This includes the node ID, the thermal mass of each node, the initial temperature of each node, the x, y, and z coordinates of each node.

ELEMENTS	Lists an expanded version of ELEMENTS_1 that now includes the normal vector of each element. This includes the parent surface of the element (surface to which the element belongs), the ID of node *a*, the ID of node *b*, the ID of node *c*, the ID of node *d*, the element area, and the x, y, and z components of the element normal vector.
THERMOPHYSICAL PROPERTIES	Lists THERMOPHYSICAL_1, including the ID, the density, the conductivity, and the specific heat of each property.
THERMO-OPTICAL PROPERTIES	Lists THERMOOPTICAL_1, including the ID, the solar absorptivity, and the IR emissivity of each property.
SURFACE PROPERTIES	Lists data from both SURFACES_1 and SURFACES_2, linking the surfaces and sides to geometry, properties, and radiation groups. This includes the ID, surface area, thermo-physical material, thermo-optical property of the top side, thermo-optical property of the bottom side, thickness, x, y, and z components of the normal vector, top side and bottom side radiation groups, roles of the top and bottom sides, and the space node flags of the top and bottom sides of each surface.
SURFACE DEFINITIONS	Lists the nodes in each surface.
CONDUCTANCES	Lists the data in CONDUCTANCES_1 with nonexistent nodes pruned out, similar to the NODES section.
USER DEFINED HEAT LOADS	Lists the heat load for each node as a single vector, as defined by the user.

4.2. Solving the Model

The pre-processing steps create a thermal network that represents a real-life system but does not yet have the desired temperature data. This data is generated in the propagation of the thermal network through time, as performed by the solver.

The first step the solver must do is read the bulk data file. This step assumes that all data generated in the pre-processing step is lost, therefore requiring the reading of the bulk data into memory. This also reshapes the data into a more easily workable set, reducing the calculation steps required during the solve. To read the data, the solver function is given the name and/or path to the bulk data file, then accesses the data line by line. This data is read in and converted back to the requisite data type per section being read, hence the need for section headers. Once all the data is fully read back into memory, preparation can begin for solving.

Preparation is needed for radiation solving. Conduction is handled fully by the conductance matrix, but radiation introduces additional heat into the system through a different variable set. Radiation is handled by a flux, meaning that an area is required to find the heat transfer rate which can then be split amongst the nodes. The areas are provided by surfaces or elements, of which there may be many depending on the size of the model. The goal of the preparation stage is to pre-calculate values that will not change during the simulation, thus saving processing time during the iterative solve. This also includes the identification of nodes and elements for which radiation is a factor in calculations. However, thanks to the prior definition of radiation groups, this becomes a simpler task.

The radiation discussed in Chapter 2 can be split into three sections: radiation out of surfaces, environmental radiation into surfaces, and radiation exchange between surfaces. Radiation out of surfaces refers to the heat rejection rate as defined by Equation 16. This is a loss

of energy from a surface, and therefore will be found as a negative value, hence the negative sign in the heat balance. This value is calculated on a per node basis, but as can be seen in the equation, requires an area to be defined. This area is the average nodal area, found using the same method taken to find the thermal mass of each node. The surface is once again divided into elements, with each element divided into quarter units. The quarter-units around each node sum up to become the area of the node. This area is then multiplied by the Stefan-Boltzmann constant and the IR emissivity value of the surface to which the node belongs to, returning a constant for later use. This must be performed twice for each node as each surface has two sides, both of which radiate heat. These sides may have different emissivity values too. Radiation groups are useful for determining which sides of surfaces will emit radiation. These can then be used to identify the thermo-optical properties of the surface. Ultimately, the constant value is found for each node for which radiation rejection is expected.

Environmental radiation includes direct solar radiation, albedo, and planetary IR. In a typical simulation, a spacecraft will be placed in orbit around a planetary body from which it will receive albedo and planetary IR radiation. It will additionally receive direct solar radiation from the sun. The orbital propagator provides the vector between the spacecraft and the sun and the vector between the spacecraft and the planetary body. These vectors, however, do not consider shadowing, in which parts of the spacecraft may block radiation from reaching other parts. Preparation is needed before the propagation stage to evaluate the radiation groups and create rotation matrices for each element in the radiation groups. Rotation matrices are used to transform a vector in one reference from to another reference frame. There are a multitude of ways to form a rotation matrix, but the selected method in this case is Rodrigues' rotation formula. Two vectors are inputted, the first being the vector that must be transformed and the second being a unit vector

describing the desired transformation. A rotation vector, *e*, is created by taking the cross product of the two aforementioned vectors, which is an axis about which the transformation can be related into a singular rotation. The angle of rotation, θ, is simply the arccosine of the dot product of the input vectors. Rodrigues' rotation formula produces a rotation matrix *R* with the following steps [24],

$$K = \begin{bmatrix} 0 & -e_3 & e_2 \\ e_3 & 0 & -e_1 \\ -e_2 & e_1 & 0 \end{bmatrix}$$

$$R = I_3 + \sin(\theta)\, K + (1 - \cos(\theta))K^2 \tag{38}$$

If the first input vector represents one of the principal axes of the initial reference frame and the second input vector represents the transformation of the same principal axis, this output rotation matrix will rotate vectors from the initial reference frame to the subsequent reference frame. The reason for the creation of these rotation matrices is to prepare for rotations of any vector from the body frame of the spacecraft to the local reference frame of each element. Shadowing calculations involve the use of more ray tracing techniques, which involve the drawing of rays between several elements. By rotating these rays into the reference frames of certain elements, calculations for ray-element intersection can be simplified into two-dimensional problems. This is further discussed in the description of the iterative solve.

Preparation for radiation exchange between surfaces section involves the calculation of the RPM for each radiation group. Section 2.2.4 introduced view factors but did not go into the full detail of implementation. View factors are calculated using the Monte Carlo Ray Tracing method which involves the use of randomized rays to determine the fraction of projected area one element takes up on another element [25]. In terms of radiation exchange between surfaces, this can be better explained as the proportion of energy emitted by a source element that impacts another

element. To produce these random rays, a random origin of the ray must first be selected for the source element in question. This is done by creating localized parameters for the element, using u and v to describe the local x_1 and x_2 coordinates. MATLAB's *rand* function produces a uniformly distributed random number in the interval of 0 and 1. Multiplying this by the length or width of the element produces the u and v coordinates for the random ray. The direction in which the ray leaves the element is determined using two additional parameters, θ and φ. These describe angles in the spherical coordinate system and can be seen in Figure 31.

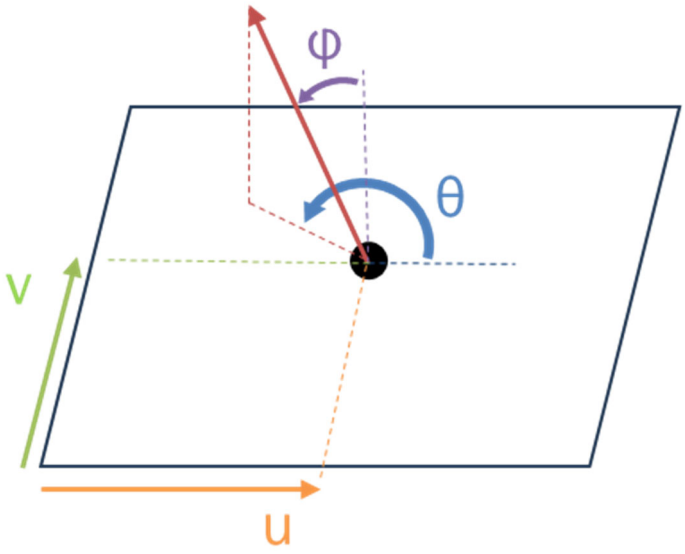

Figure 31. Monte-Carlo Ray Formulation. The rays launched in MCRT are randomly created using four parameters describing the origin of the ray and the spherical angles of its direction.

To fully capture the hemisphere of possible ray directions, the parameterized angles θ and φ are calculated with the following equations,

$$\theta = 2\pi c \tag{39}$$

$$\varphi = \sin^{-1}\left(\sqrt{c}\right) \tag{40}$$

where c is the random value in the interval of 0 and 1 generated by the MATLAB *rand* function. For completeness, the equations for u and v are found as,

$$u = L_1 c \tag{41}$$

$$v = L_2 c \tag{42}$$

where L_1 and L_2 are the side lengths of the element in the local x_1 and x_2 directions, respectively.

A number of rays are launched from each side of each element that may emit or receive radiation. These elements are defined by the radiation groups created earlier. For a reduction in computation time, these elements only search for view factors within their radiation group rather than finding view factors for the entire spacecraft at one time. The elements are stepped through one at a time, with each one launching a specified number of rays. Ray-element intersection checks are performed to determine which rays hit which elements, and then the quantities on each element are summed to find the view factors. The major step of ray tracing is checking for ray-element intersection, which is when a ray "hits" an element. This is performed with the following steps:

1. The ray is "shot" from the source element at a random origin and in a random direction.
2. The ray is rotated from the local reference frame of the source element into the spacecraft body reference frame. This is done by left-multiplying the inverse of the rotation matrix of the source element with the ray vector.
3. Every other element in the radiation group under consideration may potentially be blocking the shot ray. These elements are hypothetically extended into infinite planes with the simple identification of their normal vectors (Figure 32).
4. The point at which the ray intersects each plane is found with the equations:

$$t = \frac{(p - o) \cdot \vec{n}}{\vec{d} \cdot \vec{n}} \tag{43}$$

$$r(t) = o + td \tag{44}$$

where p is a point on the target/blocking element, o is the origin of the ray, $\vec{n}$ is the normal vector of the target/blocking element, and $\vec{d}$ is the unit vector describing the direction of the ray. The function r produces a position vector in three-dimensional space where the ray intersects the infinite blocking plane, though not necessarily the element.

5. The point *r(t)* is rotated using the target/blocking element's rotation matrix calculated earlier from the spacecraft body reference frame to the target/blocking element's local reference frame. The third component of the point *r(t)* and the points defining the element are all identical as they now lie within the same plane, thus converting a three-dimensional problem into two-space.
6. The MATLAB function *inpolygon* takes the input of the point *r(t)* and the definition points of the element and returns a set of Boolean values indicating if the point is in the bounds of the element, on the bounds of the element, or outside the element altogether (Figure 33).
7. Repeat for each ray shot from the source element. Repeat again for each radiating element in the radiation group.

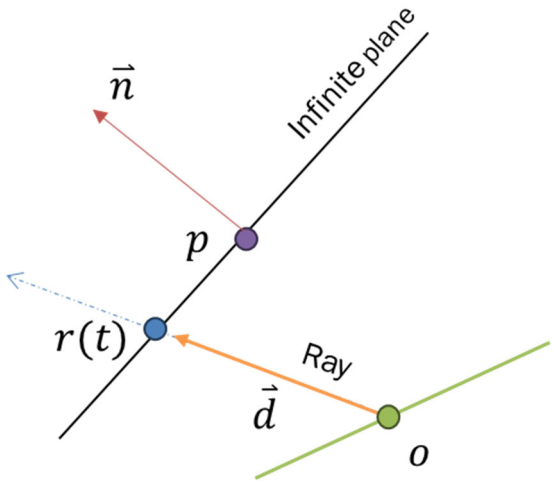

Figure 32. Ray Intersection with Infinite Plane. The ray is launched from the source element and intersects with an infinite plane described by the normal vector of a potential blocking element.

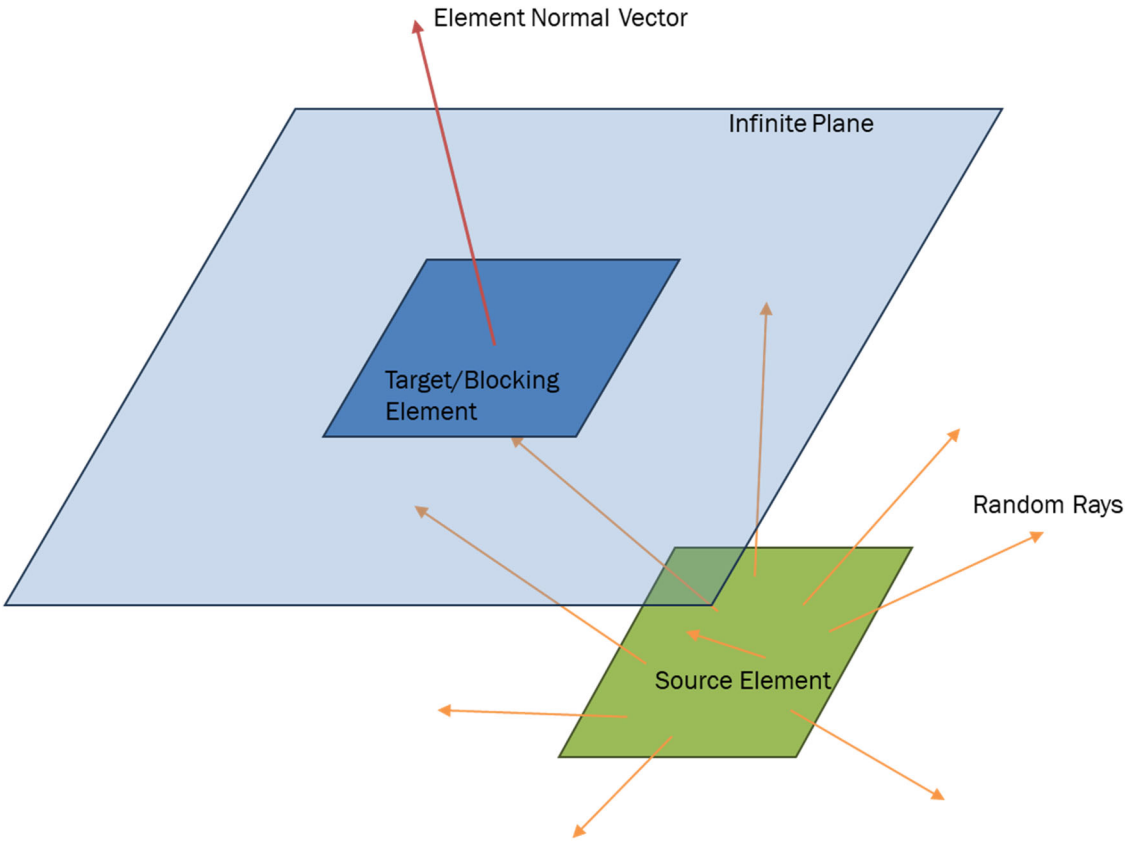

Figure 33. Ray Intersection with Finite Plate. The rays from the source element are checked against the element definition to determine intersection.

The proportion of radiation from each radiating element in the radiation group to each absorbing element in the radiation group is found in this manner. Reflections and the RPM are calculated as discussed in Section 2.2.5.

Preparation is now complete and solving can be performed given information from the orbital propagator. Within the thermal solver are several more sections to be discussed in greater detail. This includes the request for information from the orbital propagator, the various radiation calculations, the change of heat energy at each node, and the solution of final temperatures for each time step.

The FreeFlyer orbital propagator is used in tandem with the thermal solver. There are multiple ways to send and receive data between FreeFlyer and MATLAB. One of the easiest ways

is to run the programs independently. FreeFlyer can output the requested data (epoch, vector from spacecraft to sun, vector from spacecraft to orbital body, Boolean indicating if the spacecraft is in shadow) to a text file. This is done by requesting this data within FreeFlyer, converting it to a concatenated string, then printing the string to the text file. MATLAB can then read the text file into its memory and perform the thermal solve using the difference between epochs as the time step size. This method was implemented but does not match the desired implementation of a linked orbital propagator.

Another method involves the running of the MATLAB script from within FreeFlyer. This is the direct link imagined in the conception of this thesis, but it has its disadvantages. FreeFlyer has a built-in interface to MATLAB that allows for any script to be run given a call within the FreeFlyer code. This makes FreeFlyer the host and MATLAB the client. An issue arises in that the MATLAB call instances a MATLAB session and the data is not persistent between MATLAB sessions. As large portions of data, such as the conductance matrix and the nodes matrix, are stored within MATLAB, this is a problem. This data must either be sent to FreeFlyer for every call or be stored in another text file and read in each instance. This drastically slows down the solver, making this solution untenable.

A third method is the use of the FreeFlyer application programming interface (API). This allows MATLAB to run FreeFlyer instances given a short call for fast execution. However, this method was not implemented, as it requires a higher tier license of FreeFlyer. As this thesis was written with the goal of facilitating thermal simulation for small spacecraft laboratories, it is determined that this code should be viable even with the standard FreeFlyer license. Hence, a fourth method, which utilizes sockets between FreeFlyer and MATLAB was implemented.

The socket method opens two sockets on both FreeFlyer and MATLAB, enabling communication between the two programs. One socket is for sending and the other is for receiving data. To perform a mission thermal analysis, a mission must be defined first in FreeFlyer and the path to the mission file is then handed to MATLAB. MATLAB launches an instance of FreeFlyer using the mission file and notifies FreeFlyer to begin the orbital simulation. At each step, FreeFlyer pushes the desired spacecraft data to MATLAB and waits for confirmation from MATLAB to move on to the next step. MATLAB performs the solve for the step, then notifies FreeFlyer to step forward in time. This is repeated until the mission is complete. This method only requires the singular instance of FreeFlyer as a client while MATLAB remains the host. This ensures that memory is persistent between steps, reducing the overall solution time. Appendix B lists the code that must be implemented in the FreeFlyer mission enabling this information transfer method.

Within each step, the heat balance equation (Equation 36) is constructed. This includes environmental heating ($\dot{Q}_S, \dot{Q}_a, \dot{Q}_{IR}$) rates, radiation rejection from spacecraft surfaces ($\dot{Q}_{out}$), conductive heat transfer rates between nodes ($G\Delta T$), internal heat loads defined by the user ($\dot{Q}_{HL}$), and radiative heat exchange between spacecraft surfaces ($\dot{Q}_{e2e}$). Note that these are all in units of Watts. This creates the differential equation propagated for each step through a numerical method such as the Forward Euler or the Runge-Kutta 4-Step method. Figure 34 shows how the conduction of heat is performed between each node, summing to produce the heat to or from every node in the network. A source node is linked via the conductance value to all other nodes in the thermal network, passing heat through the conductance link. In the cases where the link is nonexistent, the conductance value is zero which reflects into a value of zero for the heat transfer to or from the node. Figure 35 depicts a visual representation of the heat balance equation, showing how each node has a heat input or output, which is then summed up to produce the total heating rate, $\dot{Q}_{tot}$.

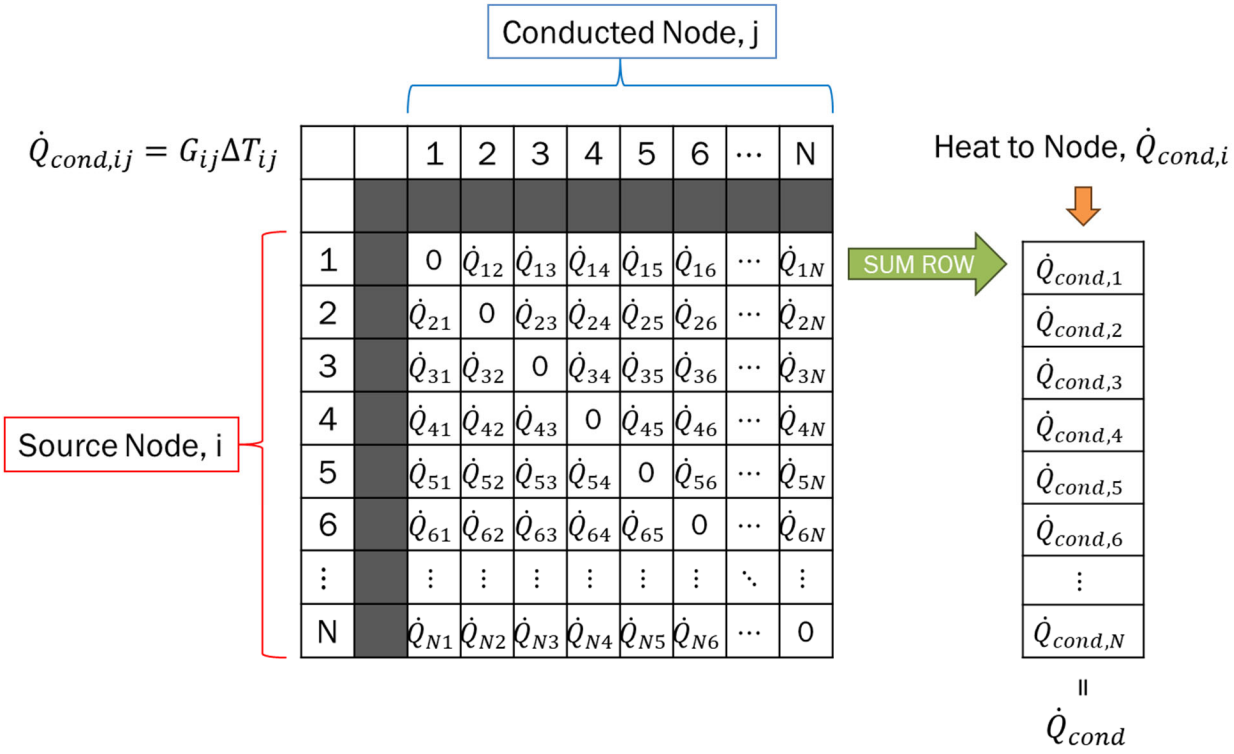

Figure 34. Heat Between Nodes. Heat is conducted between nodes and summed to find the heat going to each node.

$$\dot{Q}_{tot} = \dot{Q}_S + \dot{Q}_a + \dot{Q}_{IR} - \dot{Q}_{out} + \dot{Q}_{cond} + \dot{Q}_{HL} + \dot{Q}_{e2e}$$

$$\begin{bmatrix} \dot{Q}_{tot,1} \\ \dot{Q}_{tot,2} \\ \dot{Q}_{tot,3} \\ \vdots \\ \dot{Q}_{tot,N} \end{bmatrix} = \begin{bmatrix} \dot{Q}_{S,1} \\ \dot{Q}_{S,2} \\ \dot{Q}_{S,3} \\ \vdots \\ \dot{Q}_{S,N} \end{bmatrix} + \begin{bmatrix} \dot{Q}_{a,1} \\ \dot{Q}_{a,2} \\ \dot{Q}_{a,3} \\ \vdots \\ \dot{Q}_{a,N} \end{bmatrix} + \begin{bmatrix} \dot{Q}_{IR,1} \\ \dot{Q}_{IR,2} \\ \dot{Q}_{IR,3} \\ \vdots \\ \dot{Q}_{IR,N} \end{bmatrix} - \begin{bmatrix} \dot{Q}_{out,1} \\ \dot{Q}_{out,2} \\ \dot{Q}_{out,3} \\ \vdots \\ \dot{Q}_{out,N} \end{bmatrix} + \begin{bmatrix} \dot{Q}_{cond,1} \\ \dot{Q}_{cond,2} \\ \dot{Q}_{cond,3} \\ \vdots \\ \dot{Q}_{cond,N} \end{bmatrix} + \begin{bmatrix} \dot{Q}_{HL,1} \\ \dot{Q}_{HL,2} \\ \dot{Q}_{HL,3} \\ \vdots \\ \dot{Q}_{HL,N} \end{bmatrix} + \begin{bmatrix} \dot{Q}_{e2e,1} \\ \dot{Q}_{e2e,2} \\ \dot{Q}_{e2e,3} \\ \vdots \\ \dot{Q}_{e2e,N} \end{bmatrix}$$

Figure 35. Heat Balance. The heat balance equation can be visualized as the sum of the vectors.

Radiation rejection out from surfaces ($\dot{Q}_{out}$) uses the prepared constant values for each node. Given that the temperature of each node is calculated for every step, radiation rejection rates can be easily calculated for the following step using Equation 16. This incorporates identification of nodes that are part of surfaces that radiate, again to reduce necessary calculations. It also handles

radiation from both sides of a surface, splitting the sides into “top” and “bottom” radiation. Rate of heat loss from radiation rejection is calculated for both nodes and elements of each surface. The former is needed for Equation 36 and the latter is needed for radiation exchange between surfaces.

Environmental radiation requires more ray tracing to handle object shadowing. There are many cases where a spacecraft with a deployed component, such as a solar panel, has surfaces shadowed by the component. Additionally, there are cases where a surface is partially inside the spacecraft and is shadowed by the structure of the spacecraft. Elements outside of the spacecraft should be capable of receiving radiation from external sources while the elements of the surface inside the spacecraft are shadowed. Environmental radiation calculations are performed one radiation group at a time, just like with radiation rejection calculations. To reduce the number of elements for which this shadowing calculation must be performed, the angle between each element’s normal vector and the vector to the sun or the orbital body is first calculated. If the angle is greater than ninety-degrees, the bottom side of the element is potentially illuminated, excluding the possibility of shadowing. If the angle is less than or equal to ninety-degrees, the top side of the element is potentially illuminated. This produces a Boolean vector indicating which side of each element in the radiation group experiences environmental radiation. Shadowing is then checked by drawing rays from each element to the sun and the orbital body and checking for an intersection with other elements in the source element’s parent radiation group, similar to MCRT shadowing for radiative exchange. In this case, the ray vector is not random, instead being a targeted ray that extends toward specific points in space, these being the sun or the orbital body. However, for orbital body shadowing, if the spacecraft is close enough, the vector drawn should not be to the orbital body’s center but to a point tangent to its surface. Drawing a ray directly to the center of

the body may result in cases where an element can see parts of the body (i.e., a partial view factor exists) but are falsely flagged as shadowed (Figure 36).

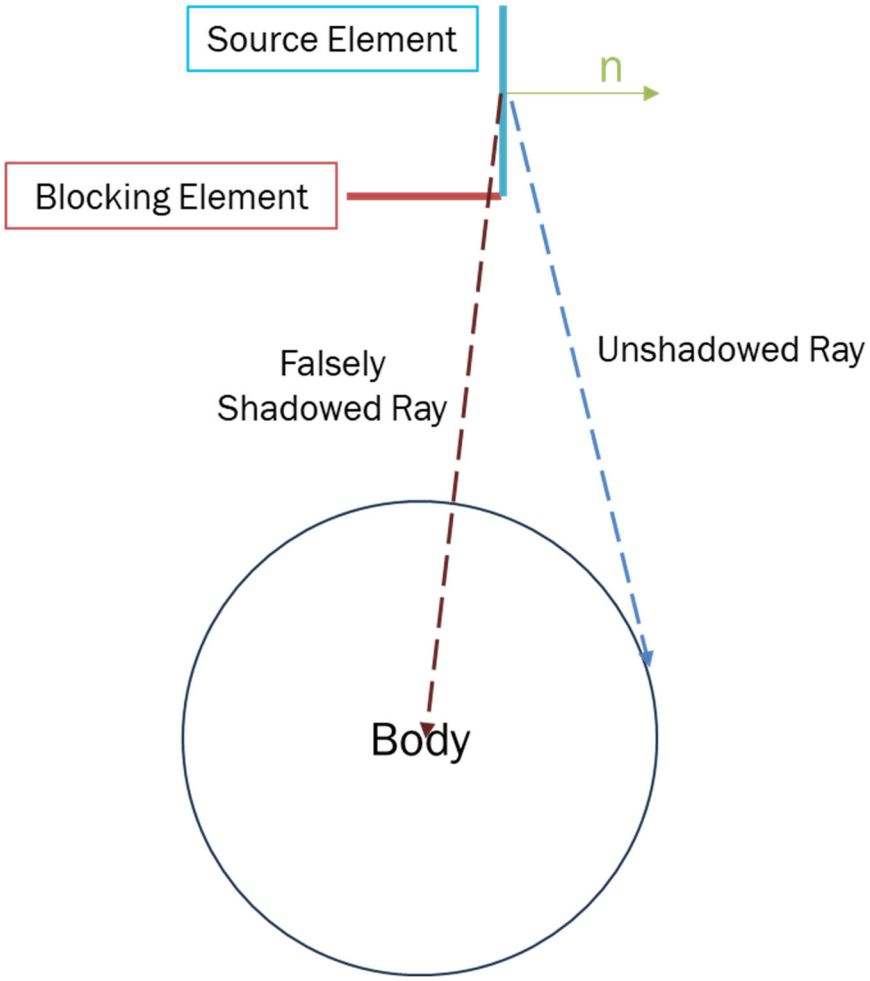

Figure 36. False Shadowing. Some elements may be falsely shadowed if the shadowing check is between the element and the center of the body.

Shadowing calculations are then performed only for four cases for each element in the current radiation group of interest.

1. The top side of the element of interest has the space node flag and is illuminated by the sun.
2. The top side of the element of interest has the space node flag and is illuminated by the orbital body.

3. The bottom side of the element of interest has the space node flag and is illuminated by the sun.
4. The bottom side of the element of interest has the space node flag and is illuminated by the orbital body.

Essentially, several criteria must be met for each side of each element in the radiation group of interest to have radiation calculations performed. This reduces computation time per step, ignoring the elements that do not need to have radiation and shadowing calculations performed. Heating rates on each element are then calculated using Equation 12, Equation 13, and Equation 15. The environmental radiation heating rate on each element is the sum of $\dot{Q}_S$, $\dot{Q}_a$, and $\dot{Q}_{IR}$ which is then split to the nodes of each element.

User heat loads on each node, $\dot{Q}_{HL}$, are already set up to be summed directly into $\dot{Q}_{tot}$. This is a constant value between steps, though it is possible to implement $\dot{Q}_{HL}$ as a time-dependent equation for a time-varying heat load.

The last step is to perform radiative heat exchange between surfaces calculations. As the RPM for each radiation group and the rejected radiation from each element has already been calculated, this calculation is trivial. Each radiation group is handled one at a time as a different RPM is held for each group. The radiation from each relevant element in the radiation group is found and formed into a vector. Multiplying the elements of this vector with the RPM results in a matrix of radiation heat exchanges between a source and a target element. Summing along the columns results in the total heat that transfers from the source elements to each target element. This is performed for the active top/bottom sides of elements within the radiation group of interest, then summed to produce $\dot{Q}_{e2e}$. This value is calculated for elements and must be split to the nodes

of each element through a separate function. This is then transposed to create a column vector and summed into Equation 36, as seen in Figure 37. This is repeated for each radiation group.

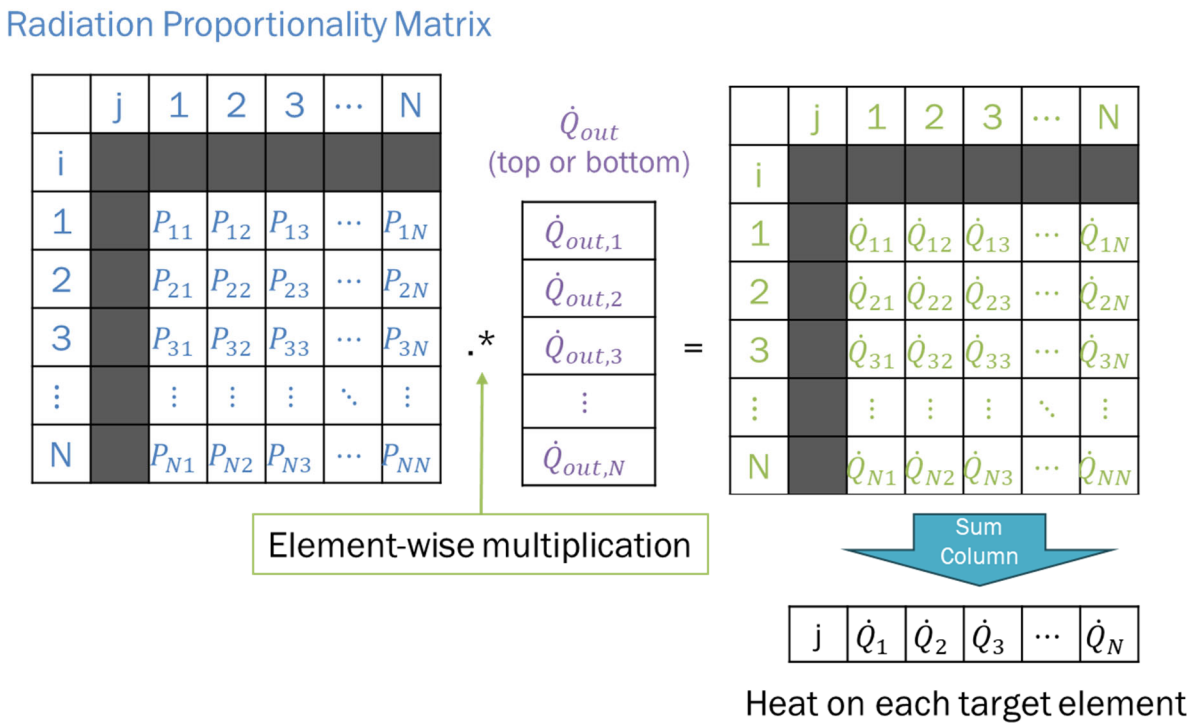

Figure 37. Surface-to-Surface Heat Transfer. Heat exchange between surface calculations involves the RPM to find the heat transferred to each target node.

Additional computation speed was added with the use of parallelization within MATLAB. The *parfor* loop within MATLAB allows for multiple computation cores to work at a time. This was used extensively for radiation calculations, solving for multiple radiation groups at one time. The number of "workers" can be increased depending on the available cores on the machine.

The total heating rate on each node, $\dot{Q}_{tot}$, can now be inserted into the FE or RK4 solver. The solver outputs the temperature of each node at the new time step which is saved in a matrix. This is the final output of the solver. Post-processing visualization code can be stepped alongside the solve, displaying the temperature change in animated sequence. This uses similar code to the pre-processing visualization tools, except it updates with each iteration.

CHAPTER 5: VERIFICATION

A test case was conducted to verify the functionality of the thermal solver. This test case involved the generation of a simple mesh, composed of six thin rectangular plates, to represent a 6U CubeSat. A similar mesh was created in Simcenter 3D, and then both were run through identical orbits with the same initial conditions. Correlating the results of the thermal solver with the results of the Simcenter 3D simulation verifies that the formulas used throughout the solver were correctly implemented and validates that the solver as a whole can be used to perform thermal analyses. Simcenter 3D is an FEA tool while the thermal solver written for this thesis utilizes FDM. Though there is a significant difference in the solvers, there should be little impact on the comparison. Verification is based solely on the results of the simulations, not on how the computations were internally executed.

The mesh, or the thermal network, was composed of 90 total nodes and 88 total elements. The edges of each rectangular plate forming the sides of the spacecraft were merged with their neighbors to reduce the complexity of the model. A similar mesh was created in Simcenter 3D with the same number of nodes and elements. Each rectangular plate was composed of Aluminum 6061-T6 with a thickness of three millimeters. The dimensions of the spacecraft in the x, y, and z directions were 0.3, 0.1, and 0.2 meters, respectively. Figure 38 and Figure 39 show the meshes created for the written thermal solver and for Simcenter 3D, respectively.

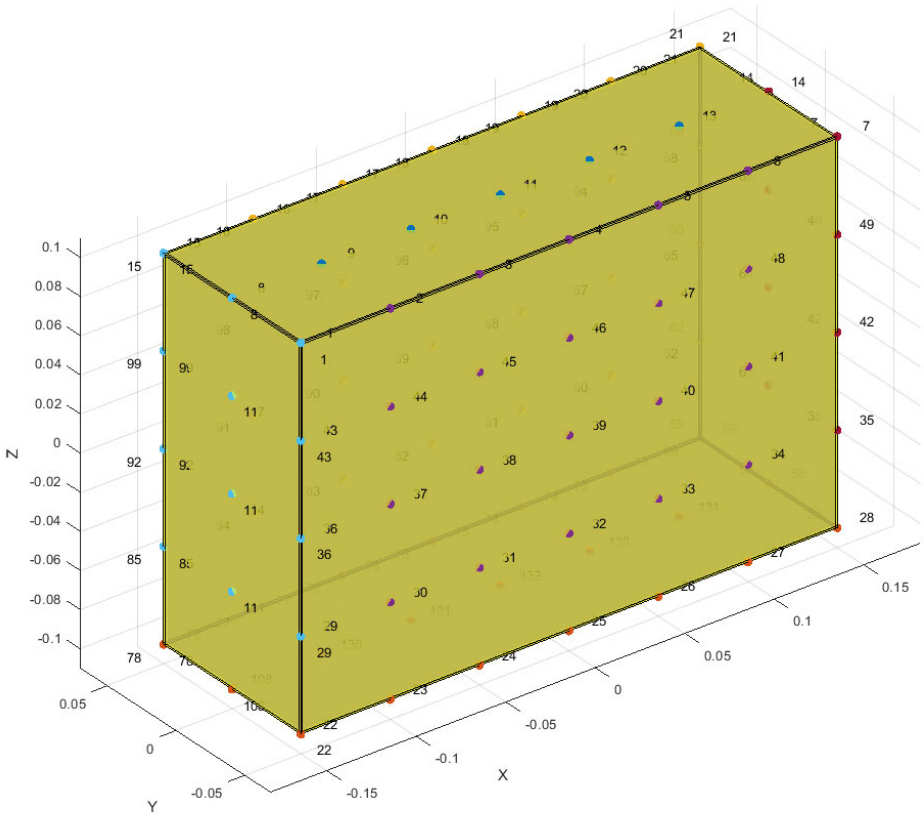

Figure 38. Mesh Created for Thermal Solver. A 6U CubeSat is visualized using the coded tools.

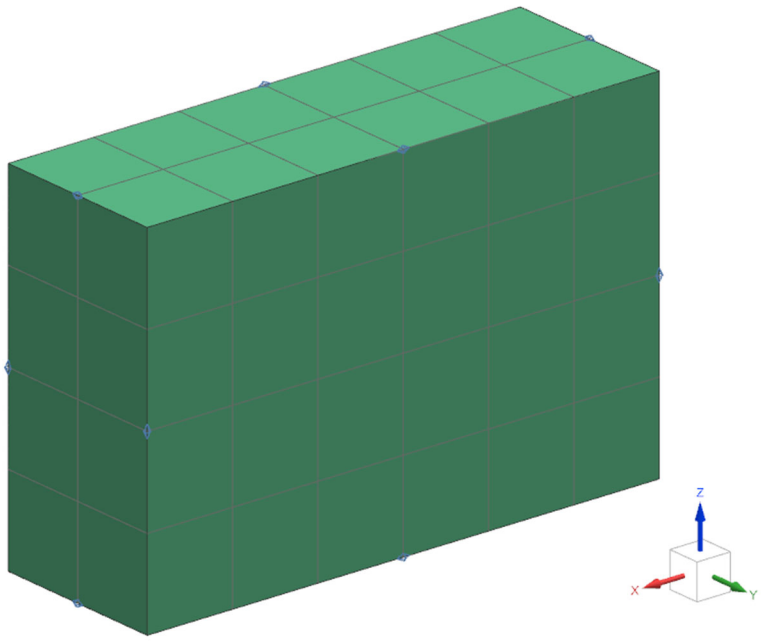

Figure 39. Mesh Created in Simcenter 3D. A 6U CubeSat is meshed for the test case in Siemens Simcenter 3D.

The initial temperature for both simulations was set as 273.15 K and both simulations began at the Julian date of 01/01/2000, 00:00:00 UTC for orbital reference. The orbit was circular (eccentricity of zero) with an orbital radius of 6778.137 km, inclination of 0 degrees, right ascension of ascending node of 0 degrees, argument of periapsis of 0 degrees, and true anomaly of 0 degrees. This orbital setup was performed in both FreeFlyer and Simcenter 3D Space Systems Thermal. Figure 40 depicts the view of the spacecraft in FreeFlyer with the spacecraft beginning its orbit south of the United States of America. The visualization of the spacecraft does not match the actual mesh, instead using one of the default visualization options, the TDRS satellite. Figure 41 depicts the view of the spacecraft in Simcenter NX in the same location, though it uses the created mesh as the visualization of the spacecraft.

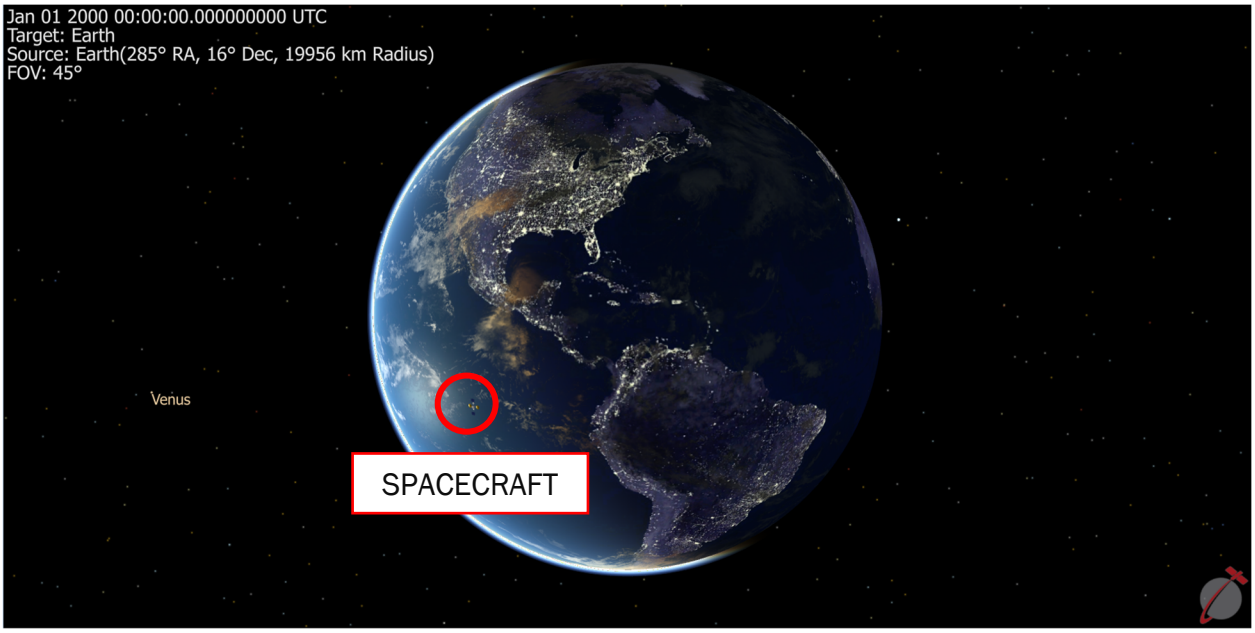

Figure 40. FreeFlyer Orbit Visualization. The spacecraft orbit is visualized in FreeFlyer.

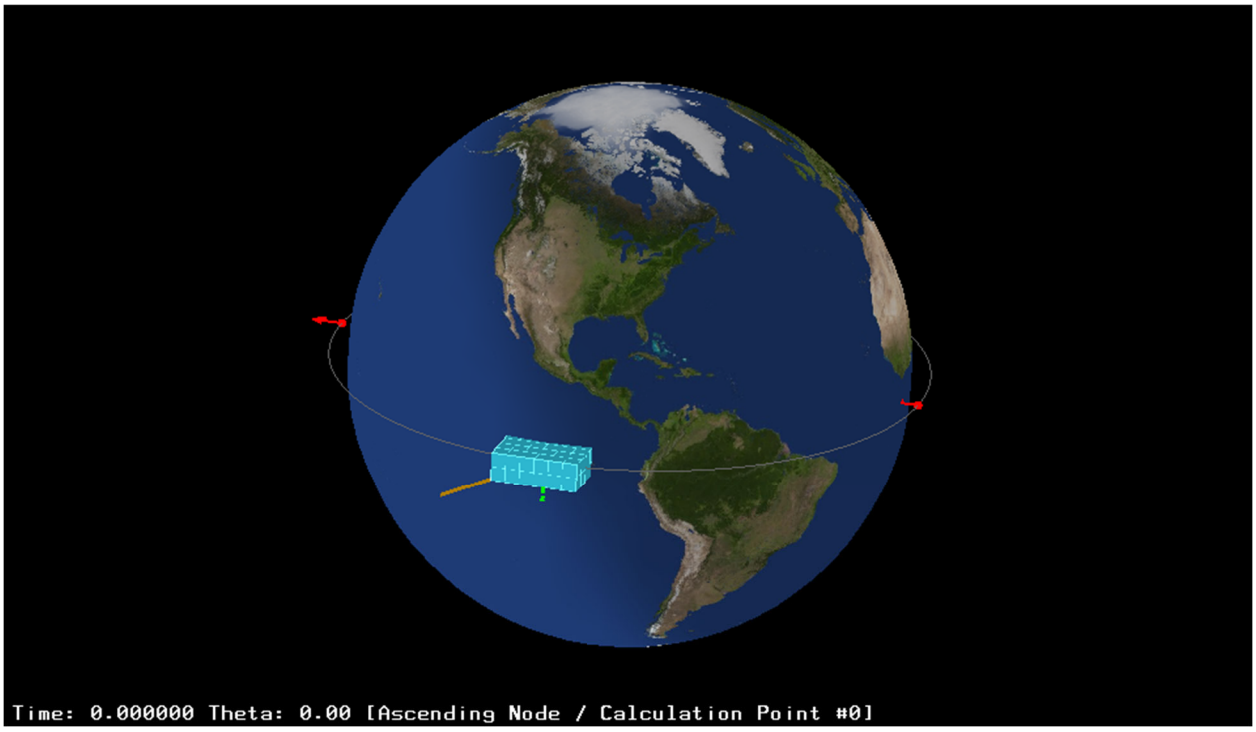

Figure 41. Simcenter 3D Orbit Visualization. The spacecraft orbit is visualized in Simcenter 3D.

In both cases, all element topsides face outwards from the spacecraft with the origin of the spacecraft body coordinate systems located at the geometric center of the spacecraft. Both sides of the surfaces were given IR emissivities/absorptivities (ε) of 0.80 and solar absorptivities (α) of 0.15. The radius of the orbital body, in this case, Earth, was set to 6378 km with an albedo factor of 0.36 and an IR temperature of 255 K. For internal radiation, both the thermal solver and Simcenter 3D were given the parameter of 2000 rays for Monte-Carlo Ray Tracing to find view factors, with a cutoff factor of 0.001 for the thermal solver. Implicit integration was used for Simcenter 3D while the *ode89* function was used for the thermal solver. Both solvers ran with a time step of 30 seconds.

The solvers were run for a total of 32400 seconds, which amounts to roughly six orbits around the Earth given the listed orbital parameters. This results in 1080 timesteps, not counting the intermediate time steps internally run by the *ode89* integrator. Output samples were saved for each time step in the thermal solver while only twenty total samples were saved for the Simcenter 3D results to save memory. Three nodes, one on each of the +Z, -Y, and -Z faces, were used to obtain temperatures across the history of the simulation. Figure 42 displays the temperatures of the three nodes in the thermal solver and Figure 43 shows the temperatures of the equivalent three nodes in Simcenter 3D Space Systems Thermal.

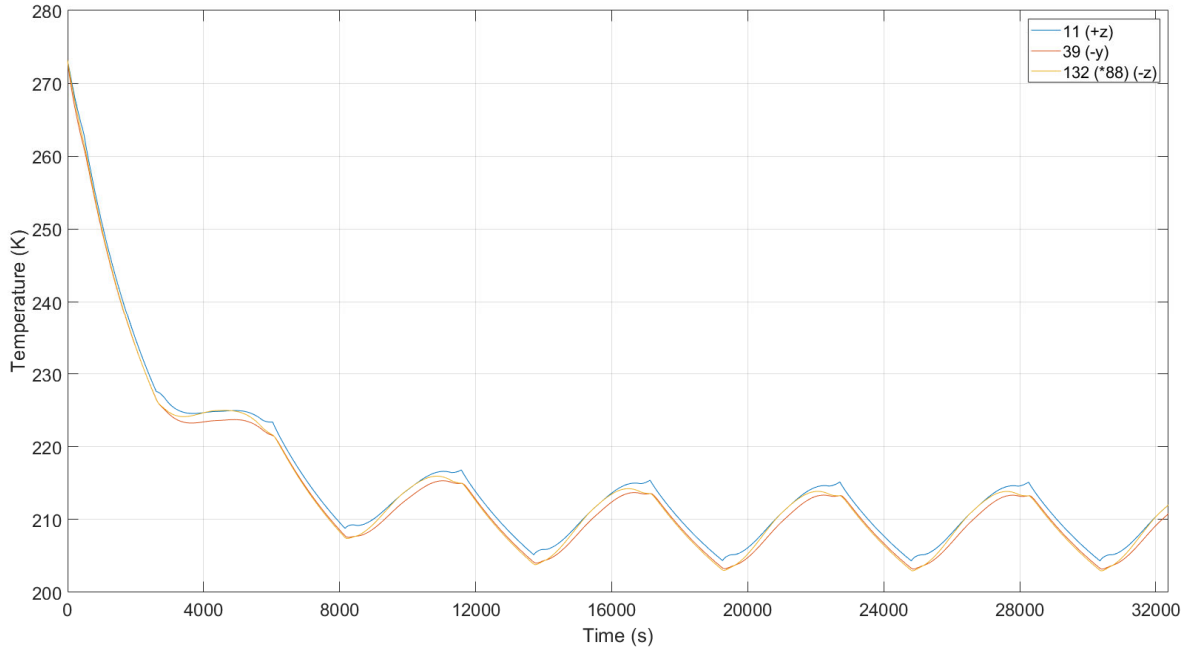

Figure 42. Thermal Solver Temperature across Time. The temperature of three nodes on the 6U CubeSat is plotted across the timespan.

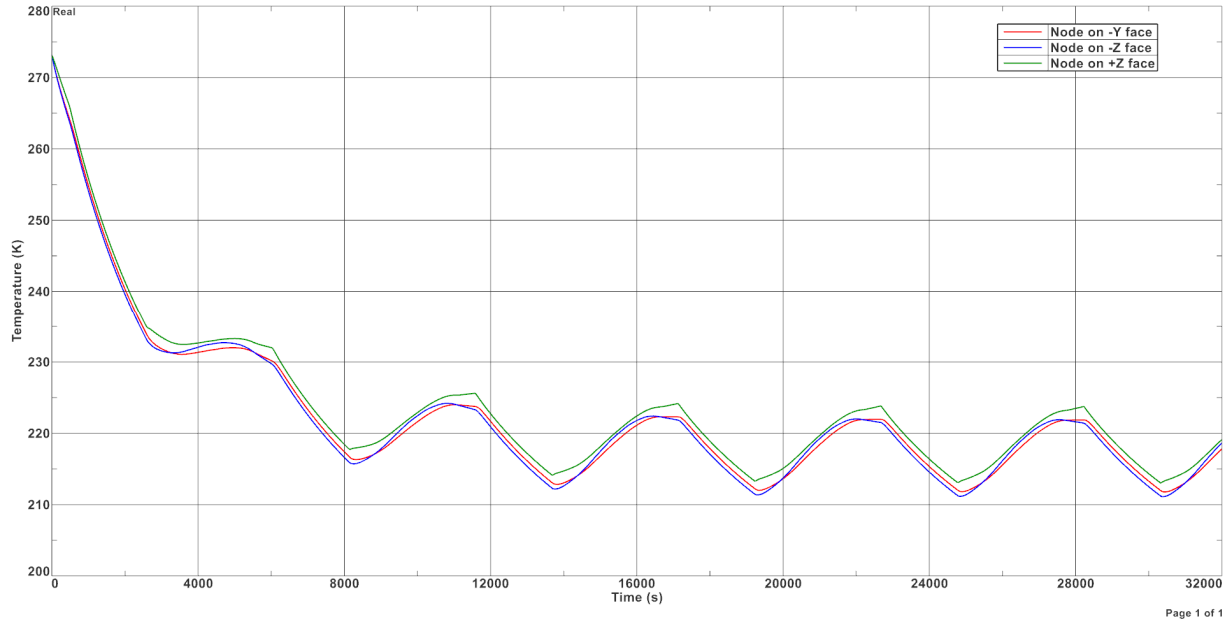

Figure 43. Simcenter 3D Temperature across Time. The temperature of three nodes on the 6U CubeSat is plotted across the timespan.

The node temperatures from the thermal solver are lower than those computed by Simcenter 3D by approximately 4%, indicating a fair correlation. Small differences in solar flux (1412.67 W/m^2 in Simcenter 3D and 1408.3–1408.6 Kelvin in the written thermal solver) and radiation rejection calculations contribute to the lower overall temperature produced by the written solver. Examining the post-processing visualization of the satellite yields Figure 44, which shows that there is a smooth gradient between every element in the mesh. This proves that the merging of the nodes has been correctly implemented as discontinuous jumps in temperature between surfaces would indicate an issue with conduction. Additionally, the high temperature is primarily on the +Y face which matches the direction of the sun at the associated time (Figure 45).

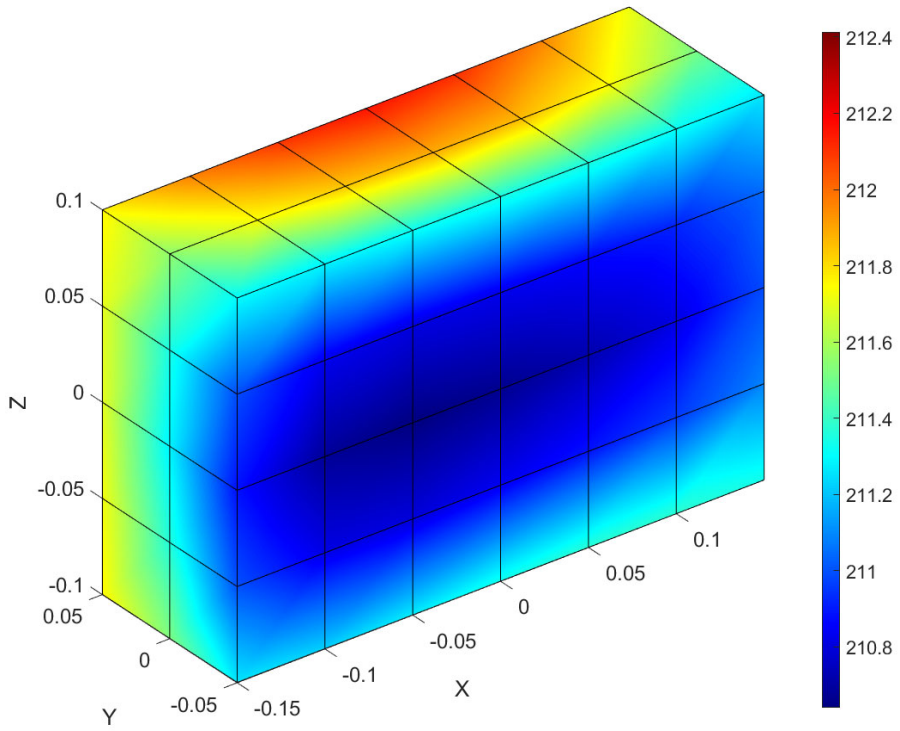

Figure 44. Final Temperature Visualization. The final temperature across the 6U CubeSat is visualized using the coded post-processing tools.

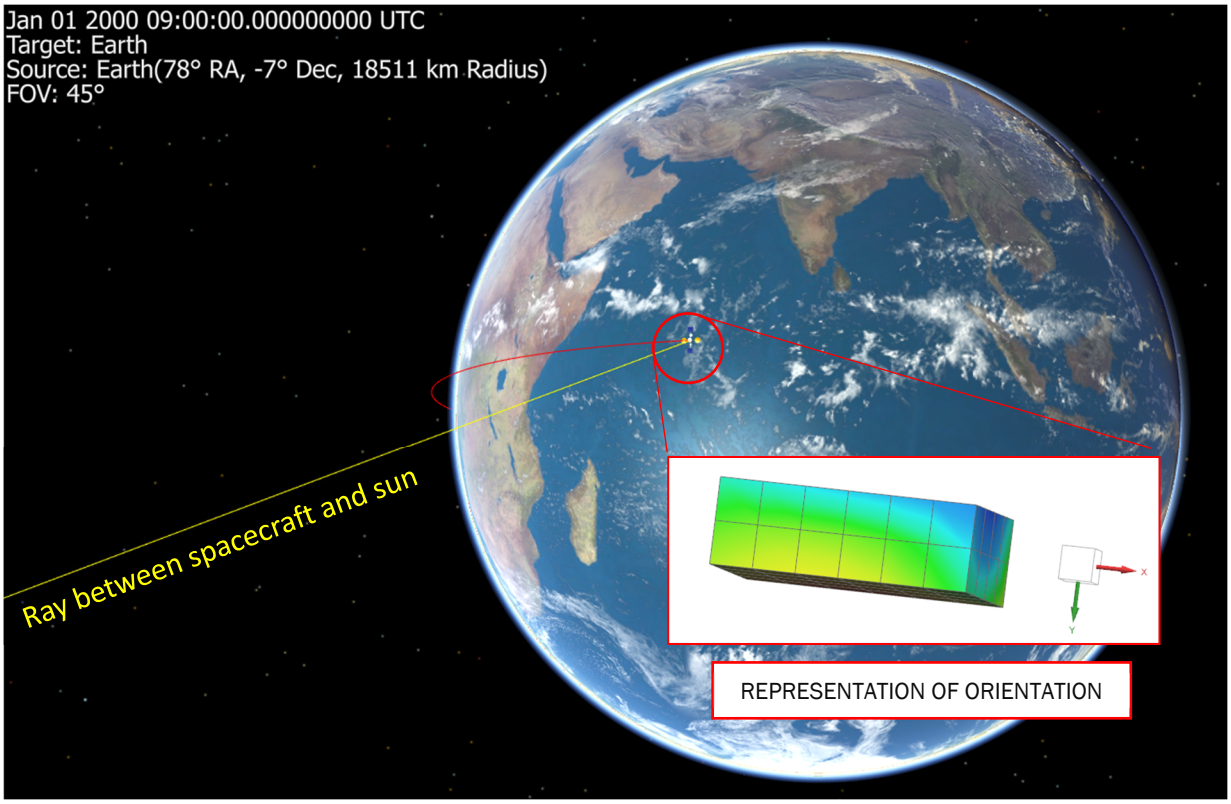

Figure 45. Ray to Sun Visualization. A ray/vector depicts the direction of the sun relative to the 6U CubeSat.

The overall run time of the MATLAB solver and simulation for this test case is 245 seconds. Simulation time can be reduced with larger time steps though this may impact the accuracy and stability of the simulation. The written solver is noticeably slower than the Simcenter 3D Space Systems Thermal solver (76 seconds) due to the data transfer through the FreeFlyer-MATLAB socket and the computation time necessitated by the external FreeFlyer propagator. In comparison, running a simulation using orbital data saved to a text file takes 80 seconds to complete. In many cases, if a mission design is set and the spacecraft thermal design is still variable, a user of this software may find it advantageous to perform the FreeFlyer simulation first, then create several different thermal network input files to run through the solver. This will eliminate time consumed by spacecraft orbit propagation during iteration of the thermal design.

In conclusion, the thermal solver appears to produce results comparable to a commercial grade analysis tool. Even though there are significant differences in the implementation of FEA and FDM models, this result shows how both methods produce near identical results for simple models such as a 6U CubeSat. Additionally, though the FreeFlyer propagation consumes the majority of the simulation time, it allows for greater, more realistic, control over the spacecraft attitude and orbit than what is available in Simcenter 3D.

CHAPTER 6: CONCLUSION AND FUTURE WORK

The thermal solver written for this thesis is capable of simulating temperatures across a spacecraft over time given the input of a linked orbital propagator. This solver works on the principles of thermodynamics, utilizing the finite difference method and associated numerical integrators to determine the movement and changes in heat for the nodes in a prescribed thermal network. The thermal solver includes numerous pre-processing tools to ease the creation of a thermal network given the user's familiarity with the model to be simulated. These allow the definition of surfaces, nodes, thermophysical properties, thermo-optical properties, heat loads, surface-to-surface contact, radiation groups, conductances, and more. Throughout the pre-processing stage, a series of holding matrices and vectors are created and then saved to an editable text file. The file is then submitted to the mathematical solver, which performs radiation and conduction calculations to determine heat transfer through the spacecraft and stepwise temperatures at each node. Afterwards, the user can take the outputs to perform post-processing and analysis, which includes using several visualization tools built specifically for this purpose. A test case was analyzed, verifying the thermal solver's results and demonstrating how it can be as effective as and more flexible than commercial grade software.

The written thermal solver is linked via a socket to the orbital propagator, FreeFlyer. Within FreeFlyer, a spacecraft mission can be designed, involving complex maneuvers that may not be feasibly replicated in a commercial-grade thermal solver. Each step of the orbital propagator outputs information to the thermal solver. Between these steps, the thermal solver may have additional propagation steps to improve stability, taking advantage of MATLAB's built-in ordinary differential equation functions. As the thermal solver only requests positions of celestial

objects relative to the spacecraft body frame, it can be easily adapted for socketing to a different orbital propagator.

The creation of a graphical user interface would make pre- and post-processing much easier for users. Additional functionality could be included, such as model checks that verify the integrity of the model before solving, additional visualization options to view radiation groups, uniform error-handling across the software, and presets to load in standard CubeSat models. Additionally, this thesis provides details into how the solver works but does not lay out the steps of how to use the solver. A user's guide, separate from this document, is being created, and can be provided by the author upon request.

In conclusion, the tool developed for this thesis fulfills the role of a standard thermal solver for mission development. It has been verified by comparison to commercially available packages. However, it exceeds the capabilities of those packages by providing greater fidelity and flexibility for complex mission analysis through the incorporation of real-time orbiting dynamic effects affecting the heat flow through a spacecraft.

REFERENCES

[1] S. Loff, "Apollo 11 Mission Overview - NASA." Accessed: Feb. 17, 2024. [Online]. Available: https://www.nasa.gov/history/apollo-11-mission-overview/

[2] D. G. Gilmore, Ed., *Spacecraft thermal control handbook*, 2nd ed. El Segundo, Calif: Aerospace Press, 2002.

[3] S. Wolfram, *A new kind of science*. Champaign, IL: Wolfram Media, 2002.

[4] H. C. Foust Iii, *Thermodynamics, Gas Dynamics, and Combustion*. Cham: Springer International Publishing, 2022. doi: 10.1007/978-3-030-87387-5.

[5] "Second Law - Entropy," Glenn Research Center | NASA. Accessed: Apr. 16, 2024. [Online]. Available: https://www1.grc.nasa.gov/beginners-guide-to-aeronautics/second-law-entropy/

[6] "Walther Nernst | Biography & Third Law of Thermodynamics | Britannica." Accessed: Apr. 16, 2024. [Online]. Available: https://www.britannica.com/biography/Walther-Nernst

[7] M. J. De Oliveira, *Equilibrium Thermodynamics*. in Graduate Texts in Physics. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017. doi: 10.1007/978-3-662-53207-2.

[8] W. designed and developed by Z.- https://www.zarr.com, "What is the difference between heat and temperature?," Applied Thermal Control. Accessed: Feb. 24, 2024. [Online]. Available: https://www.app-therm.com/faqs/glossary/heat-vs-temperature.aspx

[9] F. Wille, M. Nehrig, and M. Feldkamp, "Thermal performance of transportation packages for radioactive materials," in *Safe and Secure Transport and Storage of Radioactive Materials*, Elsevier, 2015, pp. 107–121. doi: 10.1016/B978-1-78242-309-6.00008-3.

[10] J. Meseguer, I. Pérez-Grande, and A. Sanz-Andrés, *Spacecraft thermal control*, 1. publ. in Woodhead Publishing in mechanical engineering. Oxford [u.a]: Woodhead Publ, 2012.

[11] J.-Y. Yu, "Solar Energy Incident On the Earth," University of California, Irvine.

[12] J. M. Fontenla, J. Harder, W. Livingston, M. Snow, and T. Woods, "High-resolution solar spectral irradiance from extreme ultraviolet to far infrared," *J. Geophys. Res.*, vol. 116, no. D20, p. D20108, Oct. 2011, doi: 10.1029/2011JD016032.

[13] "Stefan-Boltzmann law | Definition & Facts | Britannica." Accessed: Apr. 16, 2024. [Online]. Available: https://www.britannica.com/science/Stefan-Boltzmann-law

[14] E. A. Thornton, *Thermal Structures for Aerospace Applications*. Washington DC: American Institute of Aeronautics and Astronautics, 1996. doi: 10.2514/4.862540.

[15] R. Radebaugh, "Cryogenics," National Institute of Standards and Technology. Accessed: Feb. 25, 2024. [Online]. Available: https://trc.nist.gov/cryogenics/cryocoolers.html

[16] J. Šesták, P. Hubík, and J. J. Mareš, "Historical Roots and Development of Thermal Analysis and Calorimetry," in *Glassy, Amorphous and Nano-Crystalline Materials: Thermal Physics, Analysis, Structure and Properties*, J. Šesták, J. J. Mareš, and P. Hubík, Eds., in Hot Topics in Thermal Analysis and Calorimetry. , Dordrecht: Springer Netherlands, 2011, pp. 347–370. doi: 10.1007/978-90-481-2882-2_21.

[17] I. Müller, *A History of Thermodynamics: The Doctrine of Energy and Entropy*. Guildford Boulder: Springer London NetLibrary, Inc. [distributor].

[18] D. J. Estep, *Practical analysis in one variable*. in Undergraduate texts in mathematics. New York: Springer, 2002.
[19] A. Bennett, "Elementary Differential Equations," Kansas State University. Accessed: Apr. 16, 2024. [Online]. Available: https://onlinehw.math.ksu.edu/math340book/chap1/xc1.php
[20] L. Milne-Thomson, *The Calculus of Finite Differences*. in AMS Chelsea Publishing Series. American Mathematical Society, 2000.
[21] J. R. Cannon, *The one-dimensional heat equation*. in Encyclopedia of mathematics and its applications ; Section, Analysis, no. v. 23. Cambridge [Cambridgeshire] ; New York, NY, USA: Cambridge University Press, 1984.
[22] "Thermal Modeling." Accessed: Apr. 16, 2024. [Online]. Available: https://lennon.astro.northwestern.edu/Cryostat/Thermal/manual.html
[23] "View and set current colormap - MATLAB colormap." Accessed: Apr. 16, 2024. [Online]. Available: https://www.mathworks.com/help/matlab/ref/colormap.html
[24] E. W. Weisstein, "Rodrigues' Rotation Formula." Accessed: Apr. 16, 2024. [Online]. Available: https://mathworld.wolfram.com/
[25] M. J. Welch, "Introduction to Thermal Desktop," [Online]. Available: https://tfaws.nasa.gov/TFAWS03/software_training/thermal_desktop.pdf

APPENDIX A: MATLAB CODE

This appendix presents the MATLAB code written and used in this thesis. Numerous functions were created in preparation for use in a GUI. Scripts were additionally written to verify these functions. Table 4 lists these functions and scripts with short descriptions of their purpose. This code can also be requested from the author or accessed at https://github.com/qilim2/ThermalSolver.

Table 4. List of MATLAB Scripts and Functions. This table lists the various MATLB scripts and functions written for this thesis.

checkout_surface_04.m is the main script used for validation of the thermal solver, as noted in Chapter 5. This script exercises the majority of the following functions and provides an example of how to set up a model and solve it.
bulk_data_parser_R02.m reads the bulk data text file and converts it to usable variables within the solver.
bulk_data_process_R02.m writes the bulk data text file using the information created during pre-processing.
conductor_create_R01.m creates node-to-node conduction paths between selected nodes.
contactor_create_R01.m creates surface-to-surface contacts between selected surfaces.
create_rotmat_R01.m creates rotation matrices to convert from one given vector to another.
elem2nodes_R01.m intakes heating rates on elements and splits them to the nodes comprising the elements.
elements_define_R01.m defines the elements of a surface, creating the ELEMENTS_1 matrix.
heat_flux_create_R01.m creates a heat flux at the given surface.
heat_load_create_R02.m creates a heat load on either a node or a surface.

input_file_parser_R01.m reads the input file from the orbital propagator and converts it to usable data within the solver.
internal_view_factors_R02.m finds view factors within a radiation group.
internodal_area_R03.m finds the internodal area within nodes of a surface.
mcrt_R03.m performs Monte-Carlo Ray Tracing between given elements.
merge_nodes_R02.m merges nodes within a given set and within a maximum search range.
mesh_visualization_R01.m visualizes the elements and normal of the thermal network/mesh during pre-processing.
nodal_area_elem_normals_R01.m finds surface areas local to nodes and defines the element normal vectors.
nodal_lengths_R02.m finds the internodal distances within a mesh.
node_create_R01.m explicitly creates user defined nodes.
node_edit_R01.m edits node definitions.
node_move_R01.m moves nodes in three-dimensional space.
rad_group_assign_R01.m assigns sides of surfaces to radiation groups.
rad_proportions_R01.m creates the Radiation Proportionality Matrix given surface information and view factors.
radiation_environmental_R03.m calculates the environmental radiation (solar, albedo, IR) upon elements in a radiation group.
radiation_internal_R01.m calculates the internal radiation within a radiation group during the solve.
radiation_internal_prep_R01.m prepares element-to-element connections using the RPM.

radiation_out_R03.m calculates the radiated energy emitted by each node within the thermal network.
ray_elem_intersect_R04.m determines if a ray hits a given element.
rect_elem_center_R01.m calculates the center of a given element.
set_nodal_temperature_R02.m sets the initial temperature at each node given in MESHGRIDS_1.
set_nodal_thermal_mass_R01.m calculates the thermal mass for each node in the mesh and sets it in MESHGRIDS_1.
solar_flux_R01.m calculates the solar flux given the distance from the sun.
solver_integrated_R08.m is the primary solver function, outputting the temperature of each node at each timestep.
surface_conductance_R02.m calculates the conductances between nodes in a surface and sets them within CONDUCTANCES_1.
surface_mesh_create_R01.m creates surfaces in three-dimensional space.
surface_mesh_edit_R01.m edits the properties of already created surfaces.
surface_move_R01.m moves created surfaces in three-dimensional space.
surface_post_visualization_R02.m visualizes the temperatures in the mesh within the integrated solver function.
surface_post_visualization_R03.m visualizes the temperatures in the mesh after the solve given a timestep.
surface_pre_visualization_R01.m visualizes the nodes of the thermal network/mesh during pre-processing.

surface_properties_assign_R02.m assigns thermophysical and thermo-optical properties to surfaces.
thermooptical_create_edit_R01.m creates and edits thermo-optical properties.
thermophysical_create_edit_R01.m creates and edits thermophysical properties.
vector_angles_R03.m calculates the internal angle between two vectors.

checkout_surface_R04.m

```
close all; clear; clc;
format short
tic

%% Pre-processing plot switch
pre_plotting = 1;

%% Initialization
MESHGRIDS_1 = []; % Global holding matrix of all nodes in mesh
surface_count_current = 1; % Current index of surfaces (counts up per number of
surfaces, used only as indexing)
SURFACES_1 = {}; % Global holding cell array of surface properties
SURFACES_2 = {}; % Global holding cell array of nodes owned by surfaces and surface
nodal definitions
THERMOPHYSICAL_1 = ["thermophysical name","density (rho)","conductivity
(k)","specific heat (cp)"]; % Global holding matrix of thermophysical properties
THERMOOPTICAL_1 = ["thermo-optical name","absorptivity (alpha)","emissivity
(epsilon)"]; % Global holding matrix of thermooptical properties
CONDUCTANCES_1 = []; % Global holding matrix of conductance values.
CONDUCTANCES_2 = {}; % Cell array that holds contactor information.

%% Surface mesh (create)
% Surface mesh inputs
surface_number = 1;
surf_method = 1; % 0 = global, 1 = relative to origin
origin = [-0.15;-0.05;0.10];
vec_x11 = [0.30;0;0];
vec_x12 = [0;0.10;0];
n_x1 = 7;
n_x2 = 3;
start_node = 1;
[MESHGRIDS_1,SURFACES_2] =
surface_mesh_create_R01(surface_number,origin,surf_method,vec_x11,vec_x12,n_x1,n_x2
,start_node,MESHGRIDS_1,SURFACES_2);

surface_number = 2;
surf_method = 1;
origin = [-0.15;-0.05;-0.10];
```

```
vec_x21 = [0.30;0;0];
vec_x22 = [0;0;0.20];
n_x1 = 7;
n_x2 = 5;
start_node = 22;
[MESHGRIDS_1,SURFACES_2] =
surface_mesh_create_R01(surface_number,origin,surf_method,vec_x21,vec_x22,n_x1,n_x2
,start_node,MESHGRIDS_1,SURFACES_2);

surface_number = 3;
surf_method = 1;
origin = [0.15;-0.05;-0.10];
vec_x31 = [0;0.10;0];
vec_x32 = [0;0;0.20];
n_x1 = 3;
n_x2 = 5;
start_node = 57;
[MESHGRIDS_1,SURFACES_2] =
surface_mesh_create_R01(surface_number,origin,surf_method,vec_x31,vec_x32,n_x1,n_x2
,start_node,MESHGRIDS_1,SURFACES_2);

surface_number = 4;
surf_method = 1;
origin = [0.15;0.05;-0.10];
vec_x41 = [-0.30;0;0];
vec_x42 = [0;0;0.20];
n_x1 = 7;
n_x2 = 5;
start_node = 72;
[MESHGRIDS_1,SURFACES_2] =
surface_mesh_create_R01(surface_number,origin,surf_method,vec_x41,vec_x42,n_x1,n_x2
,start_node,MESHGRIDS_1,SURFACES_2);

surface_number = 5;
surf_method = 1;
origin = [-0.15;0.05;-0.10];
vec_x51 = [0;-0.10;0];
vec_x52 = [0;0;0.20];
n_x1 = 3;
n_x2 = 5;
start_node = 107;
[MESHGRIDS_1,SURFACES_2] =
surface_mesh_create_R01(surface_number,origin,surf_method,vec_x51,vec_x52,n_x1,n_x2
,start_node,MESHGRIDS_1,SURFACES_2);

surface_number = 6;
surf_method = 1;
origin = [-0.15;0.05;-0.10];
vec_x61 = [0.30;0;0];
vec_x62 = [0;-0.10;0];
n_x1 = 7;
n_x2 = 3;
start_node = 122;
```

```
[MESHGRIDS_1,SURFACES_2] =
surface_mesh_create_R01(surface_number,origin,surf_method,vec_x61,vec_x62,n_x1,n_x2
,start_node,MESHGRIDS_1,SURFACES_2);

fprintf('Surface mesh created.\n')

%% Thermophysical Properties
thermophysical_name = "aluminum";
rho = 2711; % Density [kg/m3]
k = 179.690; % Thermal conductivity [W/m/K]
% k = 237;
cp = 896; % Specific heat [J/kg/K]

THERMOPHYSICAL_1 =
thermophysical_create_edit_R01(thermophysical_name,rho,k,cp,THERMOPHYSICAL_1);
fprintf('Thermophsyical properties created.\n')

%% Thermo-optical Properties
thermooptical_name_01 = "Thermo_Optical_01";
alpha = 0.15; % Solar absorptivity
epsilon = 0.8; % Infrared emissivity
THERMOOPTICAL_1 =
thermooptical_create_edit_R01(thermooptical_name_01,alpha,epsilon,THERMOOPTICAL_1);

thermooptical_name_02 = "Thermo_Optical_02";
alpha = 0.3; % Solar absorptivity
epsilon = 0.5; % Infrared emissivity
THERMOOPTICAL_1 =
thermooptical_create_edit_R01(thermooptical_name_02,alpha,epsilon,THERMOOPTICAL_1);

fprintf('Thermo-optical properties created.\n')

%% Surface Properties
T_init = 273.15; % Initial temperature [K]
thickness = 0.003; % Surface thickness [m]

nodes2edit = -1;
surfaces2edit = [1,2,3,4,5,6];

% Assign surface properties to nodes
[MESHGRIDS_1,SURFACES_1] =
surface_properties_assign_R02(1,"aluminum","Thermo_Optical_01","Thermo_Optical_01",
vec_x11,vec_x12,thickness,MESHGRIDS_1,SURFACES_1,SURFACES_2);
[MESHGRIDS_1,SURFACES_1] =
surface_properties_assign_R02(2,"aluminum","Thermo_Optical_01","Thermo_Optical_01",
vec_x21,vec_x22,thickness,MESHGRIDS_1,SURFACES_1,SURFACES_2);
[MESHGRIDS_1,SURFACES_1] =
surface_properties_assign_R02(3,"aluminum","Thermo_Optical_01","Thermo_Optical_01",
vec_x31,vec_x32,thickness,MESHGRIDS_1,SURFACES_1,SURFACES_2);
[MESHGRIDS_1,SURFACES_1] =
surface_properties_assign_R02(4,"aluminum","Thermo_Optical_01","Thermo_Optical_01",
vec_x41,vec_x42,thickness,MESHGRIDS_1,SURFACES_1,SURFACES_2);
[MESHGRIDS_1,SURFACES_1] =
surface_properties_assign_R02(5,"aluminum","Thermo_Optical_01","Thermo_Optical_01",
vec_x51,vec_x52,thickness,MESHGRIDS_1,SURFACES_1,SURFACES_2);
```

```
[MESHGRIDS_1,SURFACES_1] =
surface_properties_assign_R02(6,"aluminum","Thermo_Optical_01","Thermo_Optical_01",
vec_x61,vec_x62,thickness,MESHGRIDS_1,SURFACES_1,SURFACES_2);
fprintf('Surface properties assigned to nodes.\n')

% Assign thermal masses to nodes
MESHGRIDS_1 =
set_nodal_thermal_mass_R01(surfaces2edit,thermophysical_name,THERMOPHYSICAL_1,MESHG
RIDS_1,SURFACES_1,SURFACES_2);
fprintf('Thermal masses assigned to nodes.\n')

% Assign initial temperature to nodes
MESHGRIDS_1 =
set_nodal_temperature_R02(T_init,nodes2edit,surfaces2edit,MESHGRIDS_1,SURFACES_2);
fprintf('Thermal masses assigned to nodes.\n')

%% Create HL
% Apply sample heat load along edge of surface 1
HL_nodes = 1:21;
HL_01 = zeros(height(MESHGRIDS_1),1);
HL = 0;
HL_node_count = length(HL_nodes);
HL_01(HL_nodes) = HL/HL_node_count;

fprintf('Heat load(s) created.\n')

%% Define Elements
ELEMENTS_1 = elements_define_R01(MESHGRIDS_1,SURFACES_2);
fprintf('Elements defined.\n')

%% Nodal Lengths
SURFACELENGTHS_1 = Inf*ones(height(MESHGRIDS_1)); % Initialize holding matrix after
MESHGRIDS_1 is created
SURFACELENGTHS_1 =
nodal_lengths_R02(surfaces2edit,MESHGRIDS_1,ELEMENTS_1,SURFACELENGTHS_1);
fprintf('Surface Lengths calculated.\n')

%% Surface Conductances
CONDUCTANCES_1 =
surface_conductance_R02(1,CONDUCTANCES_1,SURFACES_1,SURFACES_2,THERMOPHYSICAL_1,SUR
FACELENGTHS_1);
CONDUCTANCES_1 =
surface_conductance_R02(2,CONDUCTANCES_1,SURFACES_1,SURFACES_2,THERMOPHYSICAL_1,SUR
FACELENGTHS_1);
CONDUCTANCES_1 =
surface_conductance_R02(3,CONDUCTANCES_1,SURFACES_1,SURFACES_2,THERMOPHYSICAL_1,SUR
FACELENGTHS_1);
CONDUCTANCES_1 =
surface_conductance_R02(4,CONDUCTANCES_1,SURFACES_1,SURFACES_2,THERMOPHYSICAL_1,SUR
FACELENGTHS_1);
CONDUCTANCES_1 =
surface_conductance_R02(5,CONDUCTANCES_1,SURFACES_1,SURFACES_2,THERMOPHYSICAL_1,SUR
FACELENGTHS_1);
```

```
CONDUCTANCES_1 =
surface_conductance_R02(6,CONDUCTANCES_1,SURFACES_1,SURFACES_2,THERMOPHYSICAL_1,SUR
FACELENGTHS_1);
fprintf('Conductance matrix created.\n')

%% Merge nodes or contact
merge = 1;
if merge == 1 % Merge nodes at an edge
    % Check mass beforehand
    mass_check = sum(MESHGRIDS_1(:,2));
    fprintf('Thermal mass before merge: %f\n',mass_check)

    % Do merge
    range_tol = 1e-4;
    [MESHGRIDS_1,ELEMENTS_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
merge_nodes_R02(1,31,2,33,range_tol,0,MESHGRIDS_1,SURFACES_2,ELEMENTS_1,CONDUCTANCE
S_1,CONDUCTANCES_2); % s1 South to s2 North
    [MESHGRIDS_1,ELEMENTS_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
merge_nodes_R02(1,32,3,33,range_tol,0,MESHGRIDS_1,SURFACES_2,ELEMENTS_1,CONDUCTANCE
S_1,CONDUCTANCES_2); % s1 East to s3 North
    [MESHGRIDS_1,ELEMENTS_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
merge_nodes_R02(1,33,4,33,range_tol,0,MESHGRIDS_1,SURFACES_2,ELEMENTS_1,CONDUCTANCE
S_1,CONDUCTANCES_2); % s1 North to s4 North
    [MESHGRIDS_1,ELEMENTS_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
merge_nodes_R02(1,34,5,33,range_tol,0,MESHGRIDS_1,SURFACES_2,ELEMENTS_1,CONDUCTANCE
S_1,CONDUCTANCES_2); % s1 West to s5 North
    [MESHGRIDS_1,ELEMENTS_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
merge_nodes_R02(2,32,3,34,range_tol,0,MESHGRIDS_1,SURFACES_2,ELEMENTS_1,CONDUCTANCE
S_1,CONDUCTANCES_2); % s2 East to s3 West
    [MESHGRIDS_1,ELEMENTS_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
merge_nodes_R02(3,32,4,34,range_tol,0,MESHGRIDS_1,SURFACES_2,ELEMENTS_1,CONDUCTANCE
S_1,CONDUCTANCES_2); % s3 East to s4 West
    [MESHGRIDS_1,ELEMENTS_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
merge_nodes_R02(4,32,5,34,range_tol,0,MESHGRIDS_1,SURFACES_2,ELEMENTS_1,CONDUCTANCE
S_1,CONDUCTANCES_2); % s4 East to s5 West
    [MESHGRIDS_1,ELEMENTS_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
merge_nodes_R02(5,32,2,34,range_tol,0,MESHGRIDS_1,SURFACES_2,ELEMENTS_1,CONDUCTANCE
S_1,CONDUCTANCES_2); % s5 East to s2 West
    [MESHGRIDS_1,ELEMENTS_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
merge_nodes_R02(2,31,6,33,range_tol,0,MESHGRIDS_1,SURFACES_2,ELEMENTS_1,CONDUCTANCE
S_1,CONDUCTANCES_2); % s2 South to s6 North
    [MESHGRIDS_1,ELEMENTS_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
merge_nodes_R02(3,31,6,32,range_tol,0,MESHGRIDS_1,SURFACES_2,ELEMENTS_1,CONDUCTANCE
S_1,CONDUCTANCES_2); % s3 South to s6 East
    [MESHGRIDS_1,ELEMENTS_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
merge_nodes_R02(4,31,6,31,range_tol,0,MESHGRIDS_1,SURFACES_2,ELEMENTS_1,CONDUCTANCE
S_1,CONDUCTANCES_2); % s4 South to s6 South
    [MESHGRIDS_1,ELEMENTS_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
merge_nodes_R02(5,31,6,34,range_tol,0,MESHGRIDS_1,SURFACES_2,ELEMENTS_1,CONDUCTANCE
S_1,CONDUCTANCES_2); % s5 South to s6 West

    % Check mass again
    mass_check = sum(MESHGRIDS_1(:,2));
    fprintf('Thermal mass after merge: %f\n',mass_check)
```

```
elseif merge == 2 % Create contactor at the edge instead
    priority_switch = 1;
    CONDUCTANCES_2 =
contactor_create_R01(1,1,30,2,30,100,3,0,0.5,priority_switch,MESHGRIDS_1,SURFACES_1
,SURFACES_2,CONDUCTANCES_2);
end

%% Rad Groups
% Choose which surfaces and what sides experience external radiation (into
satellite)
% (ID,topbot,group_ID,role,sn,SURFACES_1)
% ID - Surface ID
% topbot - Select both (1), topside (2), bottomside (3)
% group_ID - Group number
% role - Role of the surface (0=ignored, 1=active+blocker, 2=blocker)
% sn - Boolean for turning on/off spacenode vision (0=off, 1=on)
SURFACES_1 = rad_groups_assign_R01(1,2,1,1,1,SURFACES_1); % Assign topside of
surface 1 to external group. Can see sn.
SURFACES_1 = rad_groups_assign_R01(1,3,2,1,0,SURFACES_1); % Assign bottomside of
surface 1 to internal group. Cannot see sn.
SURFACES_1 = rad_groups_assign_R01(2,2,1,1,1,SURFACES_1); % Assign topside of
surface 2 to external group. Can see sn.
SURFACES_1 = rad_groups_assign_R01(2,3,2,1,0,SURFACES_1); % Assign bottomside of
surface 2 to internal group Cannot see sn.
SURFACES_1 = rad_groups_assign_R01(3,2,1,1,1,SURFACES_1); % Assign topside of
surface 1 to external group. Can see sn.
SURFACES_1 = rad_groups_assign_R01(3,3,2,1,0,SURFACES_1); % Assign bottomside of
surface 1 to internal group. Cannot see sn.
SURFACES_1 = rad_groups_assign_R01(4,2,1,1,1,SURFACES_1); % Assign topside of
surface 2 to external group. Can see sn.
SURFACES_1 = rad_groups_assign_R01(4,3,2,1,0,SURFACES_1); % Assign bottomside of
surface 2 to internal group Cannot see sn.
SURFACES_1 = rad_groups_assign_R01(5,2,1,1,1,SURFACES_1); % Assign topside of
surface 1 to external group. Can see sn.
SURFACES_1 = rad_groups_assign_R01(5,3,2,1,0,SURFACES_1); % Assign bottomside of
surface 1 to internal group. Cannot see sn.
SURFACES_1 = rad_groups_assign_R01(6,2,1,1,1,SURFACES_1); % Assign topside of
surface 2 to external group. Can see sn.
SURFACES_1 = rad_groups_assign_R01(6,3,2,1,0,SURFACES_1); % Assign bottomside of
surface 2 to internal group Cannot see sn.

fprintf('Radiation groups defined.\n');

%% Plotting
% Plot nodes
marker_area = 500;
back_switch = 1;
normal_switch = 1;
surfaces2plot = [1,2,3,4,5,6];

if pre_plotting == 1
    figure(1)

surface_pre_visualization_R01(surfaces2plot,back_switch,marker_area,MESHGRIDS_1,SUR
FACES_2)
```

```
    xlabel('X'); ylabel('Y');zlabel('Z');

    figure(2)
    mesh_visualization_R01(back_switch,normal_switch,MESHGRIDS_1,ELEMENTS_1);
    xlabel('X'); ylabel('Y');zlabel('Z');

    fprintf('Pre-visualization complete.\n')

end

%% Reset timer
toc
tic

%% Save
bulk_data='checkout_04.blk';
HEATLOADS_1 = HL_01;

bulk_data_process_R02(bulk_data,MESHGRIDS_1,SURFACES_1,SURFACES_2,THERMOPHYSICAL_1,
THERMOOPTICAL_1,ELEMENTS_1,CONDUCTANCES_1,CONDUCTANCES_2,HEATLOADS_1)

%% Solve
% clear

bulk_data_file='checkout_04.blk'; % Name/path of the bulk data file
RK = 89; % 1-Forward Euler, 4-RK4, 15-ode15s, 45-ode45, 89-ode89
dt = 30; % Time step size [s]
tf = 32400; % Final time step
t_data = [dt,tf]; % Inputs for time step and final time if not defined by the input
file
plot_switch = 0; % Turn off/on plotting during the simulation
rad_switches = [1,1,1]; % Turn on/off radiation out, environmental radiation,
surface-to-surface radiation
env_rad_type = 1; % Switch between saved data (0) or concurrent FF file (1)
input_file='C:\Users\qilim2\Box\Thesis\FreeFlyer Missions\output_test_06.txt'; %
Name/path of saved data
FreeFlyerPath = 'C:\Program Files\a.i. solutions, Inc\FreeFlyer 7.8.0.56841 (64-
Bit)\'; % Path to the FF executable
PathToMissionPlanFolder = 'C:\Users\qilim2\Box\Thesis\FreeFlyer Missions\'; % Path
to the folder where the mission plan is saved
MissionPlanName = 'checkout_interface_06.MissionPlan'; % Name of the mission plan
file
if env_rad_type == 0
    env_data_paths = input_file;
elseif env_rad_type == 1
    env_data_paths = {FreeFlyerPath,PathToMissionPlanFolder,MissionPlanName};
end
R_body = 6378e3; % Radius of body [m]
T_body = 255; % Temperature of body [k]
AF = 0.36; % Albedo factor. Use 0.36 per Thornton book
env_rad_data = [R_body,T_body,AF];
cutoff = 1e-3; % Cutoff value for reflections
ray_count = 2e3; % Temporary value for ray count
s2s_rad_data = [cutoff,ray_count];
workspace_name = 'base';
```

```
[full_out,node_IDs,epoch_keep,Qdot_tot_keep,MESHGRIDS_11,elem_nodes_map] =
solver_integrated_R08(bulk_data_file,RK,t_data,plot_switch,rad_switches,env_rad_typ
e,env_data_paths,env_rad_data,s2s_rad_data,workspace_name);
fprintf('Total number of timesteps: %i\n',length(epoch_keep));
t_keep = epoch_keep-epoch_keep(1);

%% End of main script
toc

%% Plot
figure(4)
title('Nodal temperature over time')
plot(t_keep,full_out(11,:));
grid on; hold on;
plot(t_keep,full_out(39,:));
plot(t_keep,full_out(88,:));
% plot(epoch_keep-epoch_keep(1),full_out(1,:));
% plot(epoch_keep-epoch_keep(1),full_out(28,:));
legend('11 (+z)','39 (-y)','132 (*88) (-z)');
xlim([t_keep(1),t_keep(end)]);
xlabel('Time (s)'); ylabel('Temperature (K)');

figure(5)
title('Solar flux over time (element on -z face)')
plot(t_keep(1:188),Qdot_solar_elems_keep(85,1:188)/0.0025);
grid on;
xlabel('Time (s)'); ylabel('Flux (W/m2)');
xlim([t_keep(1),t_keep(188)]);

figure(6)
title('Albedo flux over time (element on +z face)')
plot(t_keep(1:188),Qdot_albedo_elems_keep(9,1:188)/0.0025);
grid on;
xlabel('Time (s)'); ylabel('Flux (W/m2)');
xlim([t_keep(1),t_keep(188)]);

figure(7)
title('IR flux over time (element on +z face)')
plot(t_keep(1:188),Qdot_IR_elems_keep(9,1:188)/0.0025);
xlabel('Time (s)'); ylabel('Flux (W/m2)');
xlim([t_keep(1),t_keep(188)]);

figure(8)
timestep_number = 1079;
surface_post_visualization_R03(timestep_number,MESHGRIDS_11,full_out,elem_nodes_map
)


%% Save
save('results_04_03.mat')
```

bulk_data_parser_R02.m

```
function [overview, MESHGRIDS_11, SURFACES_11, SURFACES_21, THERMOPHYSICAL_11,
THERMOOPTICAL_11, ELEMENTS_11, CONDUCTANCES_11, HEATLOADS_11] =
bulk_data_parser_R02(filename)
    % Parses data inside the bulk data file and converts to MATLAB vars.
    % Version 2.0 completed 10/26/2023

    % Open the file for reading
    fid = fopen(filename, 'r');

    % Read the file line by line
    ii = 1;
    while ~feof(fid)
        tline = fgetl(fid); % Get next line

        %fprintf('Round %i\n',ii)
        ii = ii + 1;

        % Find matrix sizes and preallocate matrices
        if contains(tline,'OVERVIEW')
            fgetl(fid); tline = fgetl(fid); % Go down 2 lines
            overview = str2num(tline); % Create overview matrix
            %fprintf('Overview: %s\n',mat2str(overview)) % Check output

            MESHGRIDS_11 = zeros(overview(1),6);
            ELEMENTS_11 = zeros(overview(2),9);
            THERMOPHYSICAL_11 = zeros(overview(4),4);
            THERMOOPTICAL_11 = zeros(overview(5),3);
            CONDUCTANCES_11 = zeros(overview(1),overview(1));
            SURFACES_11 = zeros(overview(3),15);
            SURFACES_21 = {};
        end

        % Read and store nodes
        if contains(tline,'NODES')
            fgetl(fid); tline = fgetl(fid); % Go down 2 lines
            for jj = 1:overview(1)
                MESHGRIDS_11(jj,:) = str2num(tline);
                tline = fgetl(fid);
            end
            %fprintf('%s\n',mat2str(MESHGRIDS_11));
        end

        % Read and store elements
        if contains(tline,'ELEMENTS')
            fgetl(fid); tline = fgetl(fid); % Go down 2 lines
            for jj = 1:overview(2)
                ELEMENTS_11(jj,:) = str2num(tline);
                tline = fgetl(fid);
            end
            %fprintf('%s\n',mat2str(ELEMENTS_1));
        end
```

```
        % Read and store thermophysical properties
        if contains(tline,'THERMOPHYSICAL PROPERTIES')
            fgetl(fid); tline = fgetl(fid); % Go down 2 lines
            for jj = 1:overview(4)
                THERMOPHYSICAL_11(jj,:) = str2num(tline);
                tline = fgetl(fid);
            end
            %fprintf('%s\n',mat2str(THERMOPHYSICAL_11));
        end

        % Read and store thermo-optical properties
        if contains(tline,'THERMO-OPTICAL PROPERTIES')
            fgetl(fid); tline = fgetl(fid); % Go down 2 lines
            for jj = 1:overview(5)
                THERMOOPTICAL_11(jj,:) = str2num(tline);
                tline = fgetl(fid);
            end
            %fprintf('%s\n',mat2str(THERMOOPTICAL_11));
        end

        % Read and store surfaces properties
        if contains(tline,'SURFACES PROPERTIES')
            fgetl(fid); tline = fgetl(fid); % Go down 2 lines
            for jj = 1:overview(3)
                SURFACES_11(jj,:) = str2num(tline);
                tline = fgetl(fid);
            end
            %fprintf('%s\n',mat2str(SURFACES_11));
        end

        % Read and store surfaces definitions
        if contains(tline,'SURFACES DEFINITIONS')
            fgetl(fid); tline = fgetl(fid); % Go down 2 lines
            for jj = 1:overview(3)
                ID = str2num(tline);
                SURFACES_21{jj}.ID = ID(1);
                SURFACES_21{jj}.nodes = ID(2:end);
                elems = find(ELEMENTS_11(:,1)==ID);
                SURFACES_21{jj}.elems = elems;
                tline = fgetl(fid);
            end
            %fprintf('%s\n',mat2str(SURFACES_21));
        end

        % Read and store conductances
        if contains(tline,'CONDUCTANCES')
            fgetl(fid); tline = fgetl(fid); % Go down 2 lines
            for jj = 1:overview(1)
                CONDUCTANCES_11(jj,:) = str2num(tline);
                tline = fgetl(fid);
            end
            %fprintf('%s\n',mat2str(CONDUCTANCES_11));
        end

        % Read and store heatloads
```

```
        if contains(tline,'USER DEFINED HEAT LOADS')
            fgetl(fid); tline = fgetl(fid); % Go down 2 lines
            HEATLOADS_11 = str2num(tline);
            %fprintf('%s\n',mat2str(HEATLOADS_11));
        end
    end

    % Close the file
    fclose(fid);
end
```

bulk_data_process_R02.m

```
function
bulk_data_process_R02(filename,MESHGRIDS_1,SURFACES_1,SURFACES_2,THERMOPHYSICAL_1,T
HERMOOPTICAL_1,ELEMENTS_1,CONDUCTANCES_1,CONDUCTANCES_2,HEATLOADS_1,~)
% Processes surfaces properties and sends information to nodes. This is
% done just before the actual solving. Packages the information into an
% input file.

% Version 2 runs with fprintf instead of writelines.
% Version 2.1 adds CONDUCTANCES_2

% Needs to also process heat loads and determine if they are valid (i.e.,
% are placed on existing nodes or surfaces)

% NECESSARY INFORMATION (for solving and plotting)
% Nodes and Temperatures (MESHGRIDS_1)
% Conductances (CONDUCTANCES_1)
% Elements (ELEMENTS_1)
% Surfaces and Thermo-optical Properties (SURFACES_1, SURFACES_2, THERMOOPTICAL_1)

% Version 2.1 completed 12/11/2023


%% Merge CONDUCTANCES_2 with CONDUCTANCES_1
for ii = 1:length(CONDUCTANCES_2) % Look through CONDUCTANCES_2 cell array
    if ~isempty(CONDUCTANCES_2{ii}) % Find nonempty cells
        mat = CONDUCTANCES_2{ii}; % Extract data from the current cell
        for jj = 1:height(mat) % Loop through each row of the data matrix
            CONDUCTANCES_1(mat(jj,1),mat(jj,2)) = mat(jj,3); % Set the conductance
value between the two nodes
        end
        %disp(CONDUCTANCES_1)
    end
end

%% Reduce matrices and vectors
nodes = find(MESHGRIDS_1(:,1)~=0);
MESHGRIDS_11 = MESHGRIDS_1(nodes,:);
CONDUCTANCES_11 = CONDUCTANCES_1(nodes,nodes);
HEATLOADS_11 = HEATLOADS_1(nodes);
```

```
%% Check MESHGRIDS for open spots
temp = MESHGRIDS_11(:,1);
check1 = isnan(MESHGRIDS_11(:,2)); % Find remaining NaN values
check2 = isnan(MESHGRIDS_11(:,3)); % Find remaining NaN values
if sum(check1) > 0
    check11 = temp(check1); % List node IDs for NaN values
    fprintf('ERROR (bulk_data_process): Thermal mass at node ID(s) %s not
defined.\n',mat2str(check11));
    return
end
if sum(check2) > 0
    check21 = temp(check2); % List node IDs for NaN values
    fprintf('ERROR (bulk_data_process): Initial temperature at node ID(s) %s not
defined.\n',mat2str(check21));
    return
end

%% Store properties
for ii = 2:height(THERMOPHYSICAL_1)
    THERMOPHYSICAL_11(ii-1,:) = [ii-1,str2double(THERMOPHYSICAL_1(ii,2:4))];
end
for ii = 2:height(THERMOOPTICAL_1)
    THERMOOPTICAL_11(ii-1,:) = [ii-1,str2double(THERMOOPTICAL_1(ii,2:3))];
end

%% Reprocess surfaces and reassign properties
jj=1;
for ii = 1:length(SURFACES_1)
    if ~isempty(SURFACES_1{ii})
        SURFACES_11(jj,1) = SURFACES_1{ii}.ID;
        SURFACES_11(jj,2) = SURFACES_1{ii}.area;
        indices = find(contains(THERMOPHYSICAL_1,SURFACES_1{ii}.material));
        SURFACES_11(jj,3) = indices(1)-1;
        indices = find(contains(THERMOOPTICAL_1,SURFACES_1{ii}.optical_top));
        SURFACES_11(jj,4) = indices(1)-1;
        indices = find(contains(THERMOOPTICAL_1,SURFACES_1{ii}.optical_bot));
        SURFACES_11(jj,5) = indices(1)-1;
        SURFACES_11(jj,6) = SURFACES_1{ii}.thickness;
        SURFACES_11(jj,7:9) = SURFACES_2{ii}.vec_x3';
        SURFACES_11(jj,10) = SURFACES_1{ii}.gtop;
        SURFACES_11(jj,11) = SURFACES_1{ii}.gtop_role;
        SURFACES_11(jj,12) = SURFACES_1{ii}.gtop_sn;
        SURFACES_11(jj,13) = SURFACES_1{ii}.gbot;
        SURFACES_11(jj,14) = SURFACES_1{ii}.gbot_role;
        SURFACES_11(jj,15) = SURFACES_1{ii}.gbot_sn;
        jj=jj+1;
    end
end

%% Get other element info
[elem_areas, elem_normals, ~] =
nodal_area_elem_normals_R01(MESHGRIDS_1,ELEMENTS_1);
```

```
%% Store in bulk data file with headers
% Create file
fid = fopen(filename,'w');

% Header
fprintf(fid,['THERMAL SOLVER BULK DATA FILE\n' ...
    'Qi Lim - University of Illinois Urbana-Champaign - M.S. Aerospace Engineering
thesis\n' ...
    '%s\n'],char(datetime));

% Overview
fprintf(fid,'\nOVERVIEW\n');
fprintf(fid,'Node Count, Element Count, Surface Count, Thermophysical Properties
Count, Thermo-optical Properties Count\n');
overview =
[length(nodes),height(ELEMENTS_1),height(SURFACES_11),height(THERMOPHYSICAL_11),hei
ght(THERMOOPTICAL_11)];
fprintf(fid,'%s\n',mat2str(overview));
fprintf(fid,'END OF SECTION\n');

% Nodes
fprintf(fid,'\nNODES\n');
fprintf(fid,'Node ID, Thermal Mass, Initial Temperature, X Coordinate, Y
Coordinate, Z Coordinate\n');
for ii = 1:height(MESHGRIDS_11)
    node_info = MESHGRIDS_11(ii,:);
    fprintf(fid,'%s\n',mat2str(node_info));
end
fprintf(fid,'END OF SECTION\n');

% Elements
fprintf(fid,'\nELEMENTS\n');
fprintf(fid,'Parent Surface, Node A ID, Node B ID, Node C ID, Node D ID, Elem Area,
Elem Norm X, Elem Norm Y, Elem Norm Z\n');
for ii = 1:height(ELEMENTS_1)
    elem_info = [ELEMENTS_1(ii,:), elem_areas(ii), elem_normals(ii,:)];
    fprintf(fid,'%s\n',mat2str(elem_info));
end
fprintf(fid,'END OF SECTION\n');

% Thermophysical properties
fprintf(fid,'\nTHERMOPHYSICAL PROPERTIES\n');
fprintf(fid,'Thermophysical Property ID, Density, Conductivity, Specific Heat\n');
for ii = 1:height(THERMOPHYSICAL_11)
    thermo_info = THERMOPHYSICAL_11(ii,:);
    fprintf(fid,'%s\n',mat2str(thermo_info));
end
fprintf(fid,'END OF SECTION\n');

% Thermo-optical properties
fprintf(fid,'\nTHERMO-OPTICAL PROPERTIES\n');
fprintf(fid,'Thermo-optical Property ID, Solar Absorptivity, IR Emissivity\n');
for ii = 1:height(THERMOOPTICAL_11)
    optical_info = THERMOOPTICAL_11(ii,:);
    fprintf(fid,'%s\n',mat2str(optical_info));
```

```
end
fprintf(fid,'END OF SECTION\n');

% Surfaces
%   Surface properties
fprintf(fid,'\nSURFACES PROPERTIES\n');
fprintf(fid,['Surface ID, Surface Area, Thermophysical Material, Topside Thermo-
optical Property, ' ...
    'Bottomside Thermo-optical Property, Surface Thickness, Surface Normal X,
Surface Normal Y, ' ...
    'Surface Normal Z, Topside Radiation Group, Topside Rad Group Role, Topside
Space Node, ' ...
    'Bottomside Radiation Group, Bottomside Rad Group Role, Bottomside Space
Node\n']);
for ii = 1:height(SURFACES_11)
    surf_info = SURFACES_11(ii,:);
    fprintf(fid,'%s\n',mat2str(surf_info));
end
fprintf(fid,'END OF SECTION\n');
%   Surface definitions
fprintf(fid,'\nSURFACES DEFINITIONS\n');
fprintf(fid,'Surface ID, List of nodes in the surface\n');
for ii = 1:length(SURFACES_2)
    if ~isempty(SURFACES_2{ii})
        surf_info = [SURFACES_2{ii}.ID SURFACES_2{ii}.nodes'];
        fprintf(fid,'%s\n',mat2str(surf_info));
    end
end
fprintf(fid,'END OF SECTION\n');

% Conductances
fprintf(fid,'\nCONDUCTANCES\n');
fprintf(fid,'Conductances associated with every other node. Row defines which node
is the source.\n');
for ii = 1:height(CONDUCTANCES_11)
    cond_info = CONDUCTANCES_11(ii,:);
    fprintf(fid,'%s\n',mat2str(cond_info));
end
fprintf(fid,'END OF SECTION\n');

% User heat loads
fprintf(fid,'\nUSER DEFINED HEAT LOADS\n');
fprintf(fid,'Heat loads applied to each node.\n');
cond_info = HEATLOADS_11;
fprintf(fid,'%s\n',mat2str(cond_info));
fprintf(fid,'END OF SECTION\n');

% Radiation Group Labels
% fprintf(fid,'\nRAD GROUP LABELS\n');
% for ii = 1:height(RADGROUPS_1)
%     cond_info = CONDUCTANCES_11(ii,:);
%     fprintf(fid,'%s\n',mat2str(cond_info));
% end

% Close file
```

```
fprintf(fid,'\nEND FILE');
fclose(fid);
end
```

conductor_create_R01.m

```
function CONDUCTANCES_2 =
conductor_create_R02(ID,source,target,target_type,cond,cond_type,area,split,MESHGRI
DS_1,ELEMENTS_1,SURFACES_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2)
% Creates a conductance between a node and another entity (node, element,
% edge, surface).

% Source must be a node
% Target can be a node, an element, an edge, or a surface

% ID - ID of conductor
% source - Source node number
% target - Target ID
% target_type - Specify if the target is a node (1), an element (2), a surface
(30), or a surface edge (31, 32, 33, 34)
% cond - conduction value
% cond_type - Specify what type of conduction is used (1-conductivity (k), 2-
thermal/area conductance (h), 3-total conductance (G))
% area - Specify area to calculate total conductance. Required for cond_type 1 and
2. Input 0 or less for targets of type 2+ to have the function calculate this based
on element or surface area/thickness.
% split - Specify if conduction value is split between targets (1) or is per node
in the target (0)

% Version 2.0 completed 11/16/2023

if split == 1 && length(target) == 1
    fprintf('Warning: Split is set to 1 while there is only a single target.\n')
end

if split == 1 % Split divides cond value by number of targets, not necessarily
number of nodes.
    cond = cond/length(target);
elseif split ~= 0 && split ~=1
    fprintf('Invalid split input. Please choose 0 (no split) or 1 (split cond value
between the target nodes.\n')
    return
end

if cond_type ~= 1 && cond_type ~= 2 && cond_type ~= 3
    fprintf('Invalid conduction type. Please input 1, 2, or 3 for cond_type with
appropriate ancillary information.\n');
    return
end

if target_type == 1 && cond_type ~= 3 && area == 0
```

```
    fprintf('Warning: Area set to zero for a node-node connection without given
total conductance. The conductance value will be zero.\n')
end

if cond_type == 3 && area ~= 0
    fprintf('Note: Area given but not used. cond_type is set to 3, Total
Conductance.\n')
end

if (target_type == 2 || target_type == 31 || target_type == 32 || target_type == 33
|| target_type == 33) && area > 0 && cond_type ~= 3
    fprintf('Note: Area given and used in priority over existing element or surface
area. Input area=0 to use the existing area.\n')
end

if target_type == 1 % Target is node(s)
    mat1 = zeros(length(target),3);
    mat2 = zeros(length(target),3);
    for ii = 1:length(target)
        % Check target
        if source == target(ii)
            fprintf('Target node cannot be the same as the source node.\n')
            return
        end
        if height(MESHGRIDS_1) < target(ii) || target(ii) < 0
        %if isempty(isempty(find(MESHGRIDS_1(:,1)==target(ii), 1)))
            fprintf('Target node %i does not exist. Select a different target
node.\n',target(ii));
            return
        end

        if cond_type == 1 % Thermal conductivity
            L = norm(MESHGRIDS_1(source,4:6)-MESHGRIDS_1(target(ii),4:6)); %
Calculate length
            G = cond*area/L; % Calculate total conductance
        elseif cond_type == 2 % Thermal area conductance
            G = cond*area; % Calculate total conductance
        elseif cond_type == 3 % Total thermal conductance
            G = cond;
        end
        % CONDUCTANCES_1(source,target(ii)) = G;
        % CONDUCTANCES_1(target(ii),source) = G;
        mat1(ii,:) = [source,target(ii),G];
        mat2(ii,:) = [target(ii),source,G];
    end
    mat = [mat1;mat2];
elseif target_type == 2 % Target is element(s)
    % Check target
    for ii = 1:length(target)
        if target(ii) > height(ELEMENTS_1)
            fprintf('Target element %i does not exist. Select a different target
element.\n',target(ii));
            return
        end
    end
```

```
    areas = zeros(length(target));
    for ii = 1:length(target) % Iterate through target elements
        if area <= 0 % Calculate area of element if needed. If area is given, then
this overrides area of all selected elements.
            nodes = ELEMENTS_1(target(ii),2:5);
            if sum(find(nodes==source)) > 0 % Check if source is within element
                fprintf('Source node must not be within the target set.\n')
                return
            end
            vec_1 = MESHGRIDS_1(nodes(2),4:6)-MESHGRIDS_1(nodes(1),4:6);
            vec_2 = MESHGRIDS_1(nodes(4),4:6)-MESHGRIDS_1(nodes(1),4:6);
            areas(ii) = norm(cross(vec_1,vec_2));
        else
            areas(ii) = area;
        end
    end

    COND_temp = [];
    if cond_type == 1 % Thermal conductivity
        for ii = 1:length(target) % Iterate through target elements
            nodes = ELEMENTS_1(target(ii),2:5);
            for jj = 1:length(nodes) % Iterate through nodes
                L = norm(MESHGRIDS_1(source,4:6)-MESHGRIDS_1(nodes(jj),4:6)); %
Calculate length
                G = cond*areas(ii)/L; % Calculate total conductance
                COND_temp = [COND_temp; nodes(jj), G]; % Assign to temporary array
            end
        end
    elseif cond_type == 2 % Thermal area conductance
        for ii = 1:length(target) % Iterate through target elements
            nodes = ELEMENTS_1(target(ii),2:5);
            for jj = 1:length(nodes) % Iterate through nodes
                G = cond*areas(ii);
                COND_temp = [COND_temp; nodes(jj), G/length(nodes)]; % Assign to
temporary array
            end
        end
    elseif cond_type == 3 % Total thermal conductance
        for ii = 1:length(target) % Iterate through target elements
            nodes = ELEMENTS_1(target(ii),2:5);
            for jj = 1:length(nodes) % Iterate through nodes
                G = cond;
                COND_temp = [COND_temp; nodes(jj), G/length(nodes)]; % Assign to
temporary array
            end
        end
    end

    % Find duplicates and select greater value
    sorted_matrix = sortrows(COND_temp, -2); % Sort the input matrix by the second
column in descending order.
    [~, unique_indices, ~] = unique(sorted_matrix(:,1), 'stable'); % Find the
indices of the duplicate values in the first column of the sorted matrix.
```

```
    COND_temp = sorted_matrix(unique_indices, :); % Keep only the duplicate rows
with the highest corresponding value in the second column.
    %fprintf('Cond_temp = %s\n',mat2str(COND_temp));

    % Assign to matrix
    mat1 = zeros(height(COND_temp),3);
    mat2 = zeros(height(COND_temp),3);
    for ii = 1:height(COND_temp)
        % CONDUCTANCES_1(source,COND_temp(ii,1)) = COND_temp(ii,2);
        % CONDUCTANCES_1(COND_temp(ii,1),source) = COND_temp(ii,2);
        mat1(ii,:) = [source,COND_temp(ii,1),G];
        mat2(ii,:) = [COND_temp(ii,1),source,G];
    end
    mat = [mat1;mat2];

elseif target_type == 30 % Target is surface(s)

    areas = zeros(length(target),1);
    for ii = 1:length(target)
        current_target = target(ii);
        if area <= 0 % Calculate area of surface if needed. If area is given, then
this overrides area of all selected elements.
            areas(ii) = SURFACES_2{current_target}.area;
        else
            areas(ii) = area;
        end
    end

    % Iterate through targets
    for ii = 1:length(target)
        % Find nodes in target surface
        current_target = target(ii);
        nodes = SURFACES_2{current_target}.nodes;
        mat1 = [];
        mat2 = [];
        if cond_type == 1 % Thermal conductivity
            for jj = 1:length(nodes)
                L = norm(MESHGRIDS_1(source,4:6)-MESHGRIDS_1(nodes(jj),4:6)); %
Calculate length
                G = cond*areas(ii)/L/length(nodes); % Calculate total conductance
of each node in target
                % CONDUCTANCES_1(source,nodes(jj)) = G;
                % CONDUCTANCES_1(nodes(jj),source) = G;
                mat1 = [mat1; source,nodes(jj),G];
                mat2 = [mat2; nodes(jj),source,G];
            end
        elseif cond_type == 2 % Thermal area conductance
            cond = cond*areas(ii); % Convert h to G based on area of target
            for jj = 1:length(nodes)
                G = cond/length(nodes);
                % CONDUCTANCES_1(source,nodes(jj)) = cond/length(nodes);
                % CONDUCTANCES_1(nodes(jj),source) = cond/length(nodes);
                mat1 = [mat1; source,nodes(jj),G];
                mat2 = [mat2; nodes(jj),source,G];
            end
```

```
        elseif cond_type == 3 % Total thermal conductance
            for jj = 1:length(nodes)
                G = cond/length(nodes);
                % CONDUCTANCES_1(source,nodes(jj)) = cond/length(nodes);
                % CONDUCTANCES_1(nodes(jj),source) = cond/length(nodes);
                mat1 = [mat1; source,nodes(jj),G];
                mat2 = [mat2; nodes(jj),source,G];
            end
        end
    end
    mat = [mat1;mat2];
elseif target_type == 31 % Target is surface(s) south edge

    areas = zeros(length(target));
    for ii = 1:length(target) % Iterate through target edges
        current_target = target(ii);
        if area <= 0 % Calculate area of element if needed. If area is given, then
this overrides area of all selected elements.
            areas(ii) =
norm(SURFACES_2{current_target}.vec_x1)*SURFACES_1{current_target}.thickness;
        else
            areas(ii) = area;
        end
    end

    % Find nodes
    edge_nodes = [];
    n_x1 = zeros(length(target),1);
    for ii = 1:length(target)
        current_target = target(ii);
        surf_nodes = SURFACES_2{current_target}.nodes;
        n_x1(ii) = SURFACES_2{current_target}.n_x1;
        edge_nodes = [edge_nodes;surf_nodes(1:n_x1),
areas(ii)/n_x1(ii)*ones(n_x1(ii),1), n_x1(ii)*ones(n_x1(ii),1)]; % Includes area
for each node in second column
    end

    % Calculate conductance for each node
    if cond_type == 1
        G = zeros(height(edge_nodes),1);
        for ii = 1:length(edge_nodes)
            L = norm(MESHGRIDS_1(source,4:6)-MESHGRIDS_1(edge_nodes(ii,1),4:6)); %
Calculate length
            G(ii) = cond*edge_nodes(ii,2)/L;
        end
    elseif cond_type == 2
        G = zeros(height(edge_nodes),1);
        for ii = 1:length(edge_nodes)
            G(ii) = cond*edge_nodes(ii,2);
        end
    elseif cond_type == 3
        G = zeros(height(edge_nodes),1);
        for ii = 1:length(edge_nodes)
            G(ii) = cond/edge_nodes(ii,3);
        end
```

```
    end
    %disp(G)

    COND_temp = [edge_nodes(:,1), G]; % Concatenate into one matrix

    % Find duplicates and select greater value
    sorted_matrix = sortrows(COND_temp, -2); % Sort the input matrix by the second
column in descending order.
    [~, unique_indices, ~] = unique(sorted_matrix(:,1), 'stable'); % Find the
indices of the duplicate values in the first column of the sorted matrix.
    COND_temp = sorted_matrix(unique_indices, :); % Keep only the duplicate rows
with the highest corresponding value in the second column.

    % Assign to matrix
    mat1 = zeros(length(nodes),3);
    mat2 = zeros(length(nodes),3);
    for ii = 1:height(COND_temp)
        % CONDUCTANCES_1(source,COND_temp(ii,1)) = COND_temp(ii,2);
        % CONDUCTANCES_1(COND_temp(ii,1),source) = COND_temp(ii,2);
        mat1(ii) = [source,nodes(jj),G];
        mat2(ii) = [nodes(jj),source,G];
    end
    mat = [mat1;mat2];

elseif target_type == 32 % Target is surface(s) east edge
    areas = zeros(length(target));
    for ii = 1:length(target) % Iterate through target edges
        current_target = target(ii);
        if area <= 0 % Calculate area of surface if needed. If area is given, then
this overrides area of all selected elements.
            areas(ii) =
norm(SURFACES_2{current_target}.vec_x2)*SURFACES_1{current_target}.thickness;
        else
            areas(ii) = area;
        end
    end

    % Find nodes
    edge_nodes = [];
    n_x2 = zeros(length(target),1);
    for ii = 1:length(target)
        current_target = target(ii);
        surf_nodes = SURFACES_2{current_target}.nodes;
        n_x1 = SURFACES_2{current_target}.n_x1;
        n_x2(ii) = SURFACES_2{current_target}.n_x2;
        for jj = 1:n_x2
            edge_nodes = [edge_nodes;surf_nodes(jj*n_x1), areas(ii)/n_x2(ii),
n_x2(ii)];
        end
    end

    % Calculate conductance for each node
    if cond_type == 1
        G = zeros(height(edge_nodes),1);
        for ii = 1:length(edge_nodes)
```

```
            L = norm(MESHGRIDS_1(source,4:6)-MESHGRIDS_1(edge_nodes(ii,1),4:6)); %
Calculate length
            G(ii) = cond*edge_nodes(ii,2)/L;
        end
    elseif cond_type == 2
        G = zeros(height(edge_nodes),1);
        for ii = 1:length(edge_nodes)
            G(ii) = cond*edge_nodes(ii,2);
        end
    elseif cond_type == 3
        G = zeros(height(edge_nodes),1);
        for ii = 1:length(edge_nodes)
            G(ii) = cond/edge_nodes(ii,3);
        end
    end

    COND_temp = [edge_nodes(:,1), G]; % Concatenate into one matrix

    % Find duplicates and select greater value
    sorted_matrix = sortrows(COND_temp, -2); % Sort the input matrix by the second
column in descending order.
    [~, unique_indices, ~] = unique(sorted_matrix(:,1), 'stable'); % Find the
indices of the duplicate values in the first column of the sorted matrix.
    COND_temp = sorted_matrix(unique_indices, :); % Keep only the duplicate rows
with the highest corresponding value in the second column.

    % Assign to matrix
    mat1 = zeros(length(nodes),3);
    mat2 = zeros(length(nodes),3);
    for ii = 1:height(COND_temp)
        % CONDUCTANCES_1(source,COND_temp(ii,1)) = COND_temp(ii,2);
        % CONDUCTANCES_1(COND_temp(ii,1),source) = COND_temp(ii,2);
        mat1(ii) = [source,nodes(jj),G];
        mat2(ii) = [nodes(jj),source,G];
    end
    mat = [mat1;mat2];

elseif target_type == 33 % Target is surface(s) north edge
    areas = zeros(length(target));
    for ii = 1:length(target) % Iterate through target edges
        current_target = target(ii);
        if area <= 0 % Calculate area of element if needed. If area is given, then
this overrides area of all selected elements.
            areas(ii) =
norm(SURFACES_2{current_target}.vec_x1)*SURFACES_1{current_target}.thickness;
        else
            areas(ii) = area;
        end
    end

    % Find nodes
    edge_nodes = [];
    n_x1 = zeros(length(target),1);
    for ii = 1:length(target)
        current_target = target(ii);
```

```
        surf_nodes = SURFACES_2{current_target}.nodes;
        n_x1(ii) = SURFACES_2{current_target}.n_x1;
        n_x2 = SURFACES_2{current_target}.n_x2;
        edge_nodes = [edge_nodes;surf_nodes(n_x1*n_x2-n_x1+1:end),
areas(ii)/n_x1(ii)*ones(n_x1(ii),1), n_x1(ii)*ones(n_x1(ii),1)];
    end

    % Calculate conductance for each node
    if cond_type == 1
        G = zeros(height(edge_nodes),1);
        for ii = 1:length(edge_nodes)
            L = norm(MESHGRIDS_1(source,4:6)-MESHGRIDS_1(edge_nodes(ii,1),4:6)); %
Calculate length
            G(ii) = cond*edge_nodes(ii,2)/L;
        end
    elseif cond_type == 2
        G = zeros(height(edge_nodes),1);
        for ii = 1:length(edge_nodes)
            G(ii) = cond*edge_nodes(ii,2);
        end
    elseif cond_type == 3
        G = zeros(height(edge_nodes),1);
        for ii = 1:length(edge_nodes)
            G(ii) = cond/edge_nodes(ii,3);
        end
    end

    COND_temp = [edge_nodes(:,1), G]; % Concatenate into one matrix

    % Find duplicates and select greater value
    sorted_matrix = sortrows(COND_temp, -2); % Sort the input matrix by the second
column in descending order.
    [~, unique_indices, ~] = unique(sorted_matrix(:,1), 'stable'); % Find the
indices of the duplicate values in the first column of the sorted matrix.
    COND_temp = sorted_matrix(unique_indices, :); % Keep only the duplicate rows
with the highest corresponding value in the second column.

    % Assign to matrix
    mat1 = zeros(length(nodes),3);
    mat2 = zeros(length(nodes),3);
    for ii = 1:height(COND_temp)
        % CONDUCTANCES_1(source,COND_temp(ii,1)) = COND_temp(ii,2);
        % CONDUCTANCES_1(COND_temp(ii,1),source) = COND_temp(ii,2);
        mat1(ii) = [source,nodes(jj),G];
        mat2(ii) = [nodes(jj),source,G];
    end
    mat = [mat1;mat2];


elseif target_type == 34 % Target is surface(s) west edge
    areas = zeros(length(target));
    for ii = 1:length(target) % Iterate through target surfaces
        current_target = target(ii);
        if ~isempty(SURFACES_1{current_target})
```

```
            if area <= 0 % Calculate area of element if needed. If area is given,
then this overrides area of all selected elements.
                areas(ii) =
norm(SURFACES_2{current_target}.vec_x2)*SURFACES_1{current_target}.thickness;
            else
                areas(ii) = area;
            end
        end
    end

    % Find nodes
    edge_nodes = [];
    n_x2 = zeros(length(target),1);
    for ii = 1:length(target)
        current_target = target(ii);
        surf_nodes = SURFACES_2{current_target}.nodes;
        n_x1 = SURFACES_2{current_target}.n_x1;
        n_x2(ii) = SURFACES_2{current_target}.n_x2;
        for jj = 1:n_x2
            edge_nodes = [edge_nodes;surf_nodes(jj*n_x1-n_x1+1),
areas(ii)/n_x2(ii),n_x2(ii)];
        end
    end
    %disp(edge_nodes)

    % Calculate conductance for each node
    if cond_type == 1
        G = zeros(height(edge_nodes),1);
        for ii = 1:length(edge_nodes)
            L = norm(MESHGRIDS_1(source,4:6)-MESHGRIDS_1(edge_nodes(ii,1),4:6)); %
Calculate length
            G(ii) = cond*edge_nodes(ii,2)/L;
        end
    elseif cond_type == 2
        G = zeros(height(edge_nodes),1);
        for ii = 1:length(edge_nodes)
            G(ii) = cond*edge_nodes(ii,2);
        end
    elseif cond_type == 3
        G = zeros(height(edge_nodes),1);
        for ii = 1:length(edge_nodes)
            G(ii) = cond/edge_nodes(ii,3);
        end
    end

    COND_temp = [edge_nodes(:,1), G]; % Concatenate into one matrix

    % Find duplicates and select greater value
    sorted_matrix = sortrows(COND_temp, -2); % Sort the input matrix by the second
column in descending order.
    [~, unique_indices, ~] = unique(sorted_matrix(:,1), 'stable'); % Find the
indices of the duplicate values in the first column of the sorted matrix.
    COND_temp = sorted_matrix(unique_indices, :); % Keep only the duplicate rows
with the highest corresponding value in the second column.
```

```
    % Assign to matrix
    mat1 = zeros(length(nodes),3);
    mat2 = zeros(length(nodes),3);
    for ii = 1:height(COND_temp)
        % CONDUCTANCES_1(source,COND_temp(ii,1)) = COND_temp(ii,2);
        % CONDUCTANCES_1(COND_temp(ii,1),source) = COND_temp(ii,2);
        mat1(ii) = [source,nodes(jj),G];
        mat2(ii) = [nodes(jj),source,G];
    end
    mat = [mat1;mat2];

else
    fprintf('Invalid target type. Please input 1 for node(s), 2 for element(s), 30
for surface(s),');
    fprintf('31 for surface(s) south edge, 32 for surface(s) east edge, 33 for
surface(s) north edge, 34 for surface(s) west edge.\n');
    return
end

CONDUCTANCES_2{ID} = mat;

end
```

contactor_create_R01.m

```
function CONDUCTANCES_2 =
contactor_create_R01(ID,source_input,source_type,target_input,target_type,cond,cond
_type,area,range_tolerance,priority_switch,MESHGRIDS_1,SURFACES_1,SURFACES_2,CONDUC
TANCES_2)
% Creates a contact conductance between two surfaces or edges. This value
% is partitioned out to the participating nodes. Turn on priority_switch to
% force each source node to seek out only the closest target node within
% tolerance range. Area value uses the following logic:
% area = -2: use area of target surface
% area = -1: use area of source surface
% area = 0: find and use smaller of the two areas
% area > 0: use this specified area in any calculations requiring it

% Result is CONDUCTANCES 2 which holds: [node 1, node 2, conductance,
% length between nodes].

% Version 1.0 completed 12/11/2023


%% Pre-check
if source_input == target_input
    fprintf('ERROR (contactor_create): Source and target cannot be the same.\n')
    return
end
```

```
%% Surface existence check
if source_type == 30 || source_type == 31 || source_type == 32 || source_type == 33
|| source_type == 34
    for ii = 1:length(source_input)
        if isempty(SURFACES_2{source_input(ii)})
            fprintf('ERROR (contactor_create): Surface %i does not
exist.\n',source_input(ii))
            return
        end
    end
end

if target_type == 30 || target_type == 31 || target_type == 32 || target_type == 33
|| target_type == 34
    for ii = 1:length(target_input)
        if isempty(SURFACES_2{target_input(ii)})
            fprintf('ERROR (contactor_create): Surface %i does not
exist.\n',target_input(ii))
            return
        end
    end
end

%% Source acquisition
n_x1 = SURFACES_2{source_input}.n_x1;
n_x2 = SURFACES_2{source_input}.n_x2;
surf_nodes = SURFACES_2{source_input}.nodes;
if source_type == 30 % Surface
    source_nodes = surf_nodes;
    source_area = SURFACES_2{source_input}.area;
elseif source_type == 31 % South edge
    source_nodes = surf_nodes(1:n_x1);
    source_area = SURFACES_1{source_input}.thickness*n_x1;
elseif source_type == 32 % East edge
    source_nodes = zeros(n_x2,1);
    for ii = 1:n_x2
        source_nodes(ii) = surf_nodes(jj*n_x1);
    end
    source_area = SURFACES_1{source_input}.thickness*n_x2;
elseif source_type == 33 % North edge
    source_nodes = surf_nodes(n_x1*n_x2-n_x1+1:end);
    source_area = SURFACES_1{source_input}.thickness*n_x1;
elseif source_type == 34 % West edge
    source_nodes = zeros(n_x2,1);
    for ii = 1:n_x2
        source_nodes(ii) = surf_nodes(ii*n_x1-n_x1+1);
    end
    source_area = SURFACES_1{source_input}.thickness*n_x2;
else % Error handling
    fprintf(['ERROR (contactor_create): Source_type does not exist. ' ...
        'Viable types are 30 (surf), 31 (south edge), ' ...
        '32 (east edge), 33 (north edge), 34 (west edge).\n'])
    return
end
```

```
%% Target acquisition
n_x1 = SURFACES_2{target_input}.n_x1;
n_x2 = SURFACES_2{target_input}.n_x2;
surf_nodes = SURFACES_2{target_input}.nodes;
if target_type == 30 % Surface
    target_nodes = surf_nodes;
    target_area = SURFACES_2{target_input}.area;
elseif target_type == 31 % South edge
    target_nodes = surf_nodes(1:n_x1);
    target_area = SURFACES_1{target_input}.thickness*n_x1;
elseif target_type == 32 % East edge
    target_nodes = zeros(n_x2,1);
    for ii = 1:n_x2
        target_nodes(ii) = surf_nodes(jj*n_x1);
    end
    target_area = SURFACES_1{target_input}.thickness*n_x2;
elseif target_type == 33 % North edge
    target_nodes = surf_nodes(n_x1*n_x2-n_x1+1:end);
    target_area = SURFACES_1{target_input}.thickness*n_x1;
elseif target_type == 34 % West edge
    target_nodes = zeros(n_x2,1);
    for ii = 1:n_x2
        target_nodes(ii) = surf_nodes(ii*n_x1-n_x1+1);
    end
    target_area = SURFACES_1{target_input}.thickness*n_x2;
else % Error handling
    fprintf(['ERROR (contactor_create): target_type does not exist. ' ...
        'Viable types are 30 (surf), 31 (south edge), ' ...
        '32 (east edge), 33 (north edge), 34 (west edge).\n'])
    return
end

%% Area handling
if area > 0
    faying_area = area;
elseif area == 0
    if source_area < target_area
        faying_area = source_area;
    else
        faying_area = target_area;
    end
elseif area == -1
    faying_area = source_area;
elseif area == -2
    faying_area = target_area;
else
    fprintf('ERROR (contactor_create): Area input issue. Please input a different
value.\n')
    return
end

%% Conduction type handling
if cond_type == 2 % Area conductance, h
    G = cond*faying_area;
elseif cond_type == 3 % Total conductance, G
```

```
    G = cond;
else % Error handling
    fprintf('ERROR (contactor_create): Cond_type not accepted. Use 2 for area
conductance [W/m2] or 3 for total conductance [W/K].')
end

%% Connection finding
% Find nodal coords
source_coords = MESHGRIDS_1(source_nodes,4:6);
target_coords = MESHGRIDS_1(target_nodes,4:6);

% Check each source node against the target nodes
dist = custom_pdist2(source_coords, target_coords);

% Check against range tolerance
[u,v] = find(dist <= range_tolerance);
w = zeros(length(u),1);
for ii = 1:length(u)
    w(ii) = dist(u(ii),v(ii));
end
u = source_nodes(u); % Source nodes numbers in range
v = target_nodes(v); % Target nodes numbers in range
dup_check = u ~= v;
u = u(dup_check); v = v(dup_check); w = w(dup_check); % Remove self-self
connections if they exist
G_split = G/length(u); % Conductance value to be assigned to each connection
result = [u,v,G_split*ones(length(u),1),w];

if priority_switch == 1 % Link only to shortest node pair that can be found
    % Find unique values of first column
    [unique_vals, ~, ~] = unique(result(:,1), 'stable');

    % Run through each unique value and compare to the duplicates
    rows2keep = [];
    for ii = 1:length(unique_vals)
        % Find indices of duplicates
        dup_indices = find(result(:,1)==unique_vals(ii));

        % Compare fourth value of unique val and the duplicates
        dist_vals = result(dup_indices,4);
        [~,I] = min(dist_vals); % Find the subindex of the minimum of the
duplicates distances amongst the duplicates
        min_index = dup_indices(I); % Get the actual index of the row
        rows2keep = [rows2keep;min_index];
    end
    result = result(rows2keep,:);
end

% Add the diagonal flip
result = [result; result(:,2),result(:,1),result(:,3:4)];

CONDUCTANCES_2{ID} = result; % Assign to contactor cell array


end
```

```
function distances = custom_pdist2(X, Y)
    % Check input dimensions
    [m, n] = size(X);
    [p, q] = size(Y);

    if n ~= q
        error('Input matrices must have the same number of columns.');
    end

    % Compute pairwise distances using Euclidean distance formula
    distances = sqrt(sum((reshape(X, m, 1, n) - reshape(Y, 1, p, n)).^2, 3));
end
```

create_rotmat_R01.m

```
function R = create_rotmat_R01(v1,v2)
    % Generates a rotation matrix to rotate from vector 1 to vector 2.
    v1 = v1/norm(v1); % Normalize
    v2 = v2/norm(v2); % Normalize
    e = cross(v1,v2); % Create rotation vector
    if sum(boolean(e))==0
        fprintf('Parallel vectors. Input intermediate rotation to clarify.\n')
        return
    end
    e = e/norm(e); % Normalize
    angle = acos(dot(v1,v2)); % Find angle
    c = cos(angle); s = sin(angle);
    % Rodriques' rotation formula
    K = [0,-e(3),e(2);e(3),0,-e(1);-e(2),e(1),0];
    R = eye(3)+s*K+(1-c)*(K^2);
end
```

elem2nodes_R01.m

```
function Qdot_nodes = elem2nodes_R01(Qdot_elems,elem_nodes_map,node_count)
% Spreads heat for each element out to the nodes

Qdot_nodes = zeros(node_count,1);
for ii = 1:length(Qdot_elems)
    Qdot_add = Qdot_elems(ii)/4;
    element_nodes = elem_nodes_map(ii,:);
    Qdot_nodes(element_nodes) = Qdot_add + Qdot_nodes(element_nodes);
end
end
```

elements_define_R01.m

```
function ELEMENTS_1 = elements_define_R01(MESHGRIDS_1,SURFACES_2)
% Creates a matrix of elements agnostic of the surfaces.
ELEMENTS_1 = NaN*ones(height(MESHGRIDS_1),5);
current_elem = 1;

% Version 1.0 completed 10/2/2023

for current_surf = 1:length(SURFACES_2) % Iterate through all the surfaces to find
the relevant nodes
    if ~isempty(SURFACES_2{current_surf}) % Check if the container is empty

        % Extract information from SURFACES_2 about the current surface
        relevant_nodes = SURFACES_2{current_surf}.nodes;
        n_x1 = SURFACES_2{current_surf}.n_x1;
        n_x2 = SURFACES_2{current_surf}.n_x2;

        % Identify node a of each element
        indices2remove = ones(n_x2-1,1); % Initialize without index in top row
        for i = 1:n_x2-1
            indices2remove(i) = i*n_x1;
        end
        relevant_indices = 1:length(relevant_nodes);
        relevant_indices(end-n_x1+1:end) = []; % Remove top row
        relevant_indices(indices2remove) = []; % Remove last column

        % Create elements
        for i=1:length(relevant_indices)
            current_index = relevant_indices(i);
            node_a = relevant_nodes(current_index);
            node_b = relevant_nodes(current_index+1);
            node_c = relevant_nodes(current_index+1+n_x1);
            node_d = relevant_nodes(current_index+n_x1);
            ELEMENTS_1(current_elem,:) =
[current_surf,node_a,node_b,node_c,node_d];
            current_elem = current_elem + 1; % Iterate up
        end
    end
end

for i = height(ELEMENTS_1):-1:1
    if isnan(ELEMENTS_1(i,1))
        ELEMENTS_1(i,:) = [];
    end
end


end
```

heat_flux_create_R01.m

```
function HL_vec = heat_flux_create_R01(HF,HF_surfaces,MESHGRIDS_1,SURFACES_2)
% Creates a heat load on either a node or a surface. Outputs as a vector
% representing the heat on each node as produced by this heat load.
% HF - Heat flux [W/m2]
% Version 1.1 completed 9/25/2023


% Check if surfaces exist
if isempty(SURFACES_2)
    fprintf('ERROR (create_heat_flux): No surfaces currently exist.\n');
    return
end

mesh_grids_size = size(MESHGRIDS_1); % Find size of mesh
nodes_total = mesh_grids_size(1); % Find total nodes in the mesh
HL_vec = [MESHGRIDS_1(:,1), zeros(nodes_total,1)]; % Initialize vector of zeros
representing heat loads on each node matched with node number


for i=1:length(HF_surfaces)
    surface_number = HF_surfaces(i);
    %fprintf('Adding heat flux to surface %i\n',surface_number);
    % Check if surface exists
    if isempty(SURFACES_2{surface_number})
        % Error handling
        fprintf('ERROR (create_heat_flux): Surface %i does not
exist.\n',surface_number);
        return
    else
        surface_area = SURFACES_2{surface_number}.area; % Obtain the surface area
of the chosen surface
        HL = HF*surface_area; % Create a heat load using the heat flux multiplied
by the surface area
        relevant_node_indeces = SURFACES_2{surface_number}.nodes; % Find relevant
indeces as described in the surface
        for j=1:length(relevant_node_indeces) % Iterate through the list of
relevant node indeces
            HL_vec(relevant_node_indeces(j),2) = HL/length(relevant_node_indeces);
% Apply portion of the heat load to the nodes in the surface
        end
    end
end


end
```

heat_load_create_R02.m

```
function HL_vec =
heat_load_create_R02(HL,HL_nodes,HL_surfaces,HL_surface_method,MESHGRIDS_1,SURFACES
_2)
% Creates a heat load on either a node or a surface. Outputs as a vector
% representing the heat on each node as produced by this heat load.
% HL - Heat load [W]
% Set HL_nodes, HL_surface = -1, HL_surface_method if not used
% HL_surface method: 1 to set all nodes in the surface as HL, 2 to divide HL among
all nodes in the surface
% Version 2.0 completed 9/25/2023

nodes_total = height(MESHGRIDS_1); % Find total nodes in the mesh
HL_vec = [MESHGRIDS_1(:,1), zeros(nodes_total,1)]; % Initialize vector of zeros
representing heat loads on each node matched with node number

if HL_nodes ~= -1
    for i = 1:length(HL_nodes)
        relevant_node = HL_nodes(i);
        if MESHGRIDS_1(relevant_node,1) == 0 || relevant_node > height(MESHGRIDS_1)
            % Error handling
            fprintf('ERROR (create_heat_load): Node %i does not
exist.\n',relevant_node);
            return
        else
            HL_vec(relevant_node,2) = HL;
        end
    end
end
if HL_surfaces ~= -1
    for i=1:length(HL_surfaces)
        surface_number = HL_surfaces(i);
        % Check if surface exists
        if isempty(SURFACES_2{surface_number})
            % Error handling
            fprintf('ERROR (create_heat_load): Surface %i does not
exist.\n',surface_number);
            return
        else
            relevant_node_indeces = SURFACES_2{surface_number}.nodes; % List
relevant indeces as described in the surface
            for j=1:length(relevant_node_indeces) % Iterate through the list of
relevant node indeces
                if HL_surface_method == 1
                    HL_vec(relevant_node_indeces(j),2) = HL; % Apply identical heat
load to all nodes in the surface
                elseif HL_surface_method == 2
                    HL_vec(relevant_node_indeces(j),2) =
HL/length(relevant_node_indeces); % Apply portion of the heat load to the nodes in
the surface
                else
                    % Error handling
```

```
                    fprintf('ERROR (create_heat_load): Heat load application method
not specified correctly.\n');
                    fprintf('Input -1 if a surface is not specified.\n');
                    fprintf('Input 1 to apply the same heat load across all nodes
in the surface.\n');
                    fprintf('Input 2 to divide the heat load equivalently across
the nodes in the surface\n')
                end
            end
        end
    end
end

end
```

input_file_parser_R01.m

```
function [epoch,vS,vBody,eclipse] = input_file_parser_R01(input_file)
% Reads the input file given path and name of file.
% Gets:
% Epoch of each time step
% Vector from s/c to Sun in s/c body frame
% Vector from s/c to nearby celestial body in s/c body frame

% Open the file for reading
fid = fopen(input_file, 'r');

% Read the file line by line
ii = 1;
while ~feof(fid)
    tline = fgetl(fid); % Get next line
    % Read and store information
    if contains(tline,'output_All')
        fgetl(fid); tline = fgetl(fid); % Go down 2 lines
    end
    values = str2num(tline); % Convert string to vector
    if ~isempty(values) % Skip empty lines
        data(ii,:) = values;
    end
    ii = ii + 1;
end

% Close the file
fclose(fid);

epoch = data(2:end,1);
vS = data(2:end,2:4).*1000;
vBody = data(2:end,5:7).*1000;
eclipse = data(2:end,8);
end
```

internal_view_factors_R02.m

```
function [VF_t,VF_b,FB_t,FB_b] =
internal_view_factors_R02(g,ecoords,ray_count,rotms)
% Calculates the view factor between elements in a radiation group. Output
% is a matrix giving the view factor of the source element (row) of the
% target element (column).

% g = struct with the following information (group information)
% .elems = list of element IDs in this group
% .tsn = boolean vector of length(.elems) indicating which elems have sn-active
topsides
% .bsn = boolean vector of length(.elems) indicating which elems have sn-active
bottomsides
% .trole = vector of length(.elems) indicating the role of each element topside
% .brole = vector of length(.elems) indicating the role of each element bottomside
% .centers = Nx3 matrix of coordinates of each element center

% ecoords = coordinates of all elements
% ray_count = number of rays to be shot
% rotms = cell array of rotation matrices that rotate from s/c body frame to local
element reference frame

% This version creates a VF matrix with front and back sides of each
% element handled separately. This allows for the radiation proportionality
% matrix to handle elements with sides of different emissivities.

% Version 2.0 completed 2/21/2024

elems1 = g.elems;
N1 = length(elems1); % Get number of elements in the rad group

% Pare down list of elements to only elements that are active or blocking
role_check = g.trole == 1 | g.trole == 2 | g.brole == 1 | g.brole == 2;
% elems2 = elems1(role_check);
% N2 = length(elems2); % Get number of elements to check
% gcoords = ecoords(elems2); % Get coordinates of only the elements in the rad
group that are active or blocking
% grotms = rotms{elems2};


% Iterate through list of elements in the group
VF_t = zeros(N1); VF_b = zeros(N1);
FB_t = zeros(N1); FB_b = zeros(N1);
for ii = 1:N1
    selem = elems1(ii); % Get current element ID
    if g.trole(ii) == 1 || g.brole(ii) == 1 % Only perform calcs for elements that
radiate out

        scoords = ecoords{selem}; % Get coordinates of the source element
        srotm = rotms{selem}; % Get the rotation matrix of the source element
```

```
        telems = elems1; telems(ii) = NaN;
        telems = telems(role_check); % Keep only elements that can block or absorb
        telems(isnan(telems)) = []; % Exclude source element
        telems_indices = 1:N1; telems_indices(ii) = NaN;
        telems_indices = telems_indices(role_check);
        telems_indices(isnan(telems_indices)) = []; % Get list of indices relating
telems to list of elems in rad group

        tcoords = ecoords(telems); % Get coordinates of the target elements
(elements that can possibly block or absorb radiation)
        trotms = rotms(telems); % Get the rotation matrices of the target elements
        % tcoords(ii) = []; % Remove coordinates of the source element from the
list of target elements coords
        % trotms = grotms; % Get the rotation matrices of the target elements
        % trotms(ii) = []; % Remove the rotation matrix of the source element from
the list of target element rotms

        [vf_t, ~, ~, frontback_of_topside] =
mcrt_R03(scoords,tcoords,ray_count,srotm,trotms,1); % Get source-topside view
factors
        [vf_b, ~, ~, frontback_of_backside] =
mcrt_R03(scoords,tcoords,ray_count,srotm,trotms,-1); % Get source-backside view
factors
        % frontback indicates if the source element sees the front or the back
        % side of the target elements. This must be used with the condition of
        % the view factor (vf1/2) since it does not care if the view factor is
        % zero.
        vf_t(isnan(vf_t)) = 0; vf_b(isnan(vf_b)) = 0;
        vf_t(isinf(vf_t)) = 0; vf_b(isinf(VF_b)) = 0;

        for jj = 1:length(telems) % Iterate through list of outputs (same length as
list of target elements)
            telem_index = telems_indices(jj); % Get index of current target
element, relative to list of elements in rad group
            VF_t(ii,telem_index) = vf_t(jj);
            VF_b(ii,telem_index) = vf_b(jj);
            FB_t(ii,telem_index) = frontback_of_topside(jj);
            FB_b(ii,telem_index) = frontback_of_backside(jj);
        end

        % This sets VF as a matrix specifically with elements of the group and
        % handles self viewing
        % if ii > 1 && ii < N1
        %     tempvf1t = vf_t(1:ii); tempvf2t = vf_t(ii+1:end);
        %     tempvf1b = vf_b(1:ii); tempvf2b = vf_b(ii+1:end);
        %     tempfb1t = frontback_of_topside(1:ii); tempfb2t =
frontback_of_topside(ii+1:end);
        %     tempfb1b = frontback_of_backside(1:ii); tempfb2b =
frontback_of_backside(ii+1:end);
        %     VF_t(ii,:) = [tempvf1t, false, tempvf2t];
        %     VF_b(ii,:) = [tempvf1b, false, tempvf2b];
        %     FB_t(ii,:) = [tempfb1t, false, tempfb2t];
        %     FB_b(ii,:) = [tempfb1b, false, tempfb2b];
        % elseif ii == 1
        %     VF_t(ii,:) = [false, vf_t];
```

```
        %     VF_b(ii,:) = [false, vf_b];
        %     FB_t(ii,:) = [false, frontback_of_topside];
        %     FB_b(ii,:) = [false, frontback_of_backside];
        % elseif ii == N1
        %     VF_t(ii,:) = [vf_t, false];
        %     VF_b(ii,:) = [vf_b, false];
        %     FB_t(ii,:) = [frontback_of_topside, false];
        %     FB_b(ii,:) = [frontback_of_backside, false];
        % end
    end
end


end
```

internodal_area_R03.m

```
function INTERNODAL_AREA_1 = internodal_area_R03(surface_number, SURFACES_1,
SURFACES_2)
% Calculates the internodal area between each node in a surface.
% Version 3 specifically creates a matrix of the same size as the surface input.
% Solves without for-loops or if-else statements.
% Version 3.0 completed 10/20/2023

% Extract data from relevant holding matrices
nodes = SURFACES_2{surface_number}.nodes;
n_x1 = SURFACES_2{surface_number}.n_x1;
n_x2 = SURFACES_2{surface_number}.n_x2;
v_x1 = SURFACES_2{surface_number}.vec_x1;
v_x2 = SURFACES_2{surface_number}.vec_x2;
thickness = SURFACES_1{surface_number}.thickness;

% Create widths
dx1 = norm(v_x1)/(n_x1-1);
dx2 = norm(v_x2)/(n_x2-1);

% Preallocate the matrix
node_count = length(nodes);
INTERNODAL_AREA_1 = zeros(node_count, node_count);

% Identify edge nodes for horizontal connections
nedge_horz = [1:n_x1, node_count-n_x1+1:node_count];

% Identify edge nodes for vertical connections
nedge_vert = [1:n_x1:node_count, n_x1:n_x1:node_count];

% Horizontal
n_horz = 1:node_count-1;
edge_horz = ismember(n_horz, nedge_horz);
n_horz_p1 = n_horz + 1;

% Input into mat
```

```
INTERNODAL_AREA_1(sub2ind([node_count, node_count], n_horz(edge_horz),
n_horz_p1(edge_horz))) = thickness*dx2/2;
INTERNODAL_AREA_1(sub2ind([node_count, node_count], n_horz_p1(edge_horz),
n_horz(edge_horz))) = thickness*dx2/2;

INTERNODAL_AREA_1(sub2ind([node_count, node_count], n_horz(~edge_horz),
n_horz_p1(~edge_horz))) = thickness*dx2;
INTERNODAL_AREA_1(sub2ind([node_count, node_count], n_horz_p1(~edge_horz),
n_horz(~edge_horz))) = thickness*dx2;

% For vertical connections
n_vert = 1:node_count-n_x1;
edge_vert = ismember(n_vert, nedge_vert);
n_vert_p1 = n_vert + n_x1;

INTERNODAL_AREA_1(sub2ind([node_count, node_count], n_vert(edge_vert),
n_vert_p1(edge_vert))) = thickness*dx1/2;
INTERNODAL_AREA_1(sub2ind([node_count, node_count], n_vert_p1(edge_vert),
n_vert(edge_vert))) = thickness*dx1/2;

INTERNODAL_AREA_1(sub2ind([node_count, node_count], n_vert(~edge_vert),
n_vert_p1(~edge_vert))) = thickness*dx1;
INTERNODAL_AREA_1(sub2ind([node_count, node_count], n_vert_p1(~edge_vert),
n_vert(~edge_vert))) = thickness*dx1;

end
```

mcrt_R03.m

```
function [vf, o_final, r_final, frontback] =
mcrt_R03(scoords,tcoords,ray_count,srotm,trotms,topbot)
% Performs Monte Carlo Ray Tracing (MCRT) to find the view factor between
% the source element and the target elements. This includes the shadowing
% of each target element upon the others.

%%% Inputs
% scoords = Source element coordinate matrix given as a 3x4 matrix with row
%           1 giving the x coordinates of each point, row 2 giving the y
coordinates
%           of each poinot, and row 3 giving the z coordinates of each point.
Points
%           are defined in order of lower left then ccw around the rectangular
%           element.
% tcoords = Target element coordinate cell array containing the coordinates
%           of each node in 3x4 matrix form/
%           Example: tcoords{elem_ID}.a = [xcoord;ycoord;zcoord]
% ray_count = Number of rays to be shot from the source element.
% srotms = Rotation matrix to rotate position vectors into the source
%          element's frame of reference.
% trotms = Cell array giving the rotation matrix to rotate position vectors
%          from s/c frame of reference into each target element's frame of
%          reference. Same length as tcoords.
% topbot = Choose to shoot rays from topside or bottomside of the element
```

```
%          (1 or -1)

%%% Outputs
% vf = View factors of each target element.
% o_final = Coordinates of each ray origin that survives.
% r_final = Location of intersect of each ray with each blocker.
% frontback = Indicates which side is seen (1=front, 0=back). Must be used
%             in conjunction with vf since it does not care if the view factor is
zero.

% Version 1.1 completed 2/19/2024

%% Prep
A = norm(scoords(:,1) - scoords(:,2)); % Length - Node a to node b
B = norm(scoords(:,1) - scoords(:,4)); % Length - Node a to node d
tcount = length(tcoords);

%% Generate rays
% Ray origin point
u = A*rand(1,ray_count); % 1xray_count random u pos
v = B*rand(1,ray_count); % 1xray_count random v pos
s = [u;v;zeros(1,ray_count)]; % Position vectors of each source point in the
reference frame of the source element.
o = srotm'*s + repmat(scoords(:,1),1,ray_count);
% o = bsxfun(@plus,srotm'*s,scoords(:,1)); % Rotate position vectors to s/c body
frame. Add values to origin of source element to get position.

% Ray vector
angle1 = asin(sqrt(rand(1,ray_count))); % Get phi angle
angle2 = rand(1,ray_count)*2*pi; % Get theta angle
u_x = cos(angle2).*sin(angle1); % Unit vector x
u_y = sin(angle2).*sin(angle1); % Unit vector y
u_z = cos(angle1); % Unit vector z
ray_unit_vec = topbot*[u_x;u_y;u_z]./vecnorm([u_x;u_y;u_z]); % Ray unit vector in
the reference frame of the source element.
d = srotm'*ray_unit_vec; % Rotate ray unit vector into the s/c body frame.
%d = reshape(d,3,ray_count); % Reshape to get 3xN matrix of unit vectors where each
column is a different, random unit vector.


%% Ray hit/intersection checks
%%% Find t for each element
t_keep = zeros(tcount,ray_count);
r_keep = zeros(3,ray_count,tcount);
tn = zeros(tcount,3);
parfor ii = 1:tcount
    tc = tcoords{ii}; % Get coordinates of target element
    p = tc(:,1); % Get a point on the target element
    n = cross(tc(:,2)-tc(:,1),tc(:,4)-tc(:,1)); % Get normal vector of the target
element
    n = n/norm(n); % Make into unit vector
    tn(ii,:) = n'; % Save normal vector into Nx3 matrix
    tr = trotms{ii}; % Get rotation matrix of target element
    [t_out,r_out,~] = ray_elem_intersect_R04(o,d,p,n,tc,tr); % Perform element hit
check
```

```
    %   t output in 1 x ray_count vector format with Infs at indices that fail or
do not hit.
    %   r output in 3 x ray_count matrix format

    t_keep(ii,:) = t_out;
    r_keep(:,:,ii) = r_out;
end

%%% Check which has lower t
%   t_keep is a tcount x ray_count matrix
[val_check,min_target] = min(t_keep,[],1); % Identify which was the first target
element hit for each ray
%[target_elems,~,~] = unique(min_target)

% Split out ray hits and count
hit_count = zeros(1,tcount);
parfor ii = 1:tcount
    look_for = ii; % Element to look for
    indices_bool = min_target == look_for; % Get indices of min_target that match
current element
    vals_at_indices = val_check(indices_bool); % Get values of t for the rays
hitting the current element
    vals_at_indices(isinf(vals_at_indices) | isnan(vals_at_indices)) = []; % Prune
out Infs and NaNs
    hit_count(ii) = numel(vals_at_indices); % Get count of elements that are not
Infs that hit the element
    indices_keep(ii,:) = indices_bool;
end
vf = hit_count./ray_count;

%% Visualization prep
o_final = cell(1,ray_count);
r_final = cell(1,ray_count);
parfor ii = 1:tcount
    indices_bool = indices_keep(ii,:); % Indices of relevant rays

    o_final{ii} = o(:,indices_bool); % Get origins of rays that hit each element,
filtered to each element
    r_temp = r_keep(:,:,ii);
    r_final{ii} = r_temp(:,indices_bool);
end

%% See topside or bottomside?
% Check if the source element sees the topside or the bottomside of each
% target element

% Get source element normal vector
sx1 = scoords(:,2) - scoords(:,1);
sx2 = scoords(:,4) - scoords(:,1);
sn = topbot*cross(sx1,sx2);
sn = repmat(sn,1,tcount)';
inc_angles = vector_angles_R03(sn,tn);
frontback = (inc_angles > pi/2)';

end
```

merge_nodes_R02.m

```
function [MESHGRIDS_1,ELEMENTS_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
merge_nodes_R02(source_input,source_type,target_input,target_type,range_tolerance,k
eep_switch,MESHGRIDS_1,SURFACES_2,ELEMENTS_1,CONDUCTANCES_1,CONDUCTANCES_2)
% This function takes in a two sets of inputs: sources and targets. Source
% nodes are checked against the target nodes using the range_tolerance,
% merging the target with the source or vice-versa depending on the
% keep_switch (0 = keep lower indexed node, 1 = keep higher indexed node).
% Source and target types are needed to identify what the sources and the
% targets are.
% 0 = all currently active nodes
% 1 = nodes
% 2 = elements
% 30 = surface
% 31 = surface south edge (input is the surface ID)
% 32 = surface east edge (input is the surface ID)
% 33 = surface north edge (input is the surface ID)
% 34 = surface west edge (input is the surface ID)

% Version 2.0 completed 12/7/2023
% Version 2.1 completed 12/11/2023

%% Pre-process
% Identify existing nodes and surfaces
nodes = find(MESHGRIDS_1(:,1)~=0);

%% Find source nodes
if source_type == 0 % All nodes currently active
    source_nodes = nodes;
elseif source_type == 1 % Nodes
    % Check if values exist in input
    for ii = 1:length(source_input)
        existcheck = MESHGRIDS_1(:,1) == source_input(ii);
        if sum(existcheck) < 1
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
    end
    source_nodes = source_input;
elseif source_type == 2 % Elements
    source_nodes = [];
    for ii = 1:length(source_input) % Grab nodes from each element and append to
list
        existcheck = ELEMENTS_1(:,1) == source_input(ii);
        if sum(existcheck) < 1
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
        source_nodes = [source_nodes, ELEMENTS_1(source_input(ii),2:5)];
```

```
    end
    source_nodes = unique(source_nodes)'; % Remove duplicates and sort
elseif source_type == 30 % Entire surface
    source_nodes = [];
    for ii = 1:length(source_input)
        if isempty(SURFACES_2{source_input(ii)})
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
        source_nodes = [source_nodes; SURFACES_2{source_input(ii)}.nodes];
    end
    source_nodes = unique(source_nodes); % Remove duplicates and sort
elseif source_type == 31 % South edge of surface
    source_nodes = [];
    n_x1 = zeros(length(source_input),1);
    for ii = 1:length(source_input)
        if isempty(SURFACES_2{source_input(ii)})
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
        current_source = source_input(ii);
        surf_nodes = SURFACES_2{current_source}.nodes;
        n_x1(ii) = SURFACES_2{current_source}.n_x1;
        source_nodes = [source_nodes;surf_nodes(1:n_x1)];
    end
    source_nodes = unique(source_nodes); % Remove duplicates and sort
elseif source_type == 32 % East edge of surface
    source_nodes = [];
    n_x2 = zeros(length(source_input),1);
    for ii = 1:length(source_input)
        if isempty(SURFACES_2{source_input(ii)})
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
        current_source = source_input(ii);
        surf_nodes = SURFACES_2{current_source}.nodes;
        n_x1 = SURFACES_2{current_source}.n_x1;
        n_x2(ii) = SURFACES_2{current_source}.n_x2;
        for jj = 1:n_x2
            source_nodes = [source_nodes;surf_nodes(jj*n_x1)];
        end
    end
    source_nodes = unique(source_nodes); % Remove duplicates and sort
elseif source_type == 33 % North edge of surface
    source_nodes = [];
    n_x1 = zeros(length(source_input),1);
    for ii = 1:length(source_input)
        if isempty(SURFACES_2{source_input(ii)})
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
```

```
        current_source = source_input(ii);
        surf_nodes = SURFACES_2{current_source}.nodes;
        n_x1(ii) = SURFACES_2{current_source}.n_x1;
        n_x2 = SURFACES_2{current_source}.n_x2;
        source_nodes = [source_nodes;surf_nodes(n_x1*n_x2-n_x1+1:end)];
    end
    source_nodes = unique(source_nodes); % Remove duplicates and sort
elseif source_type == 34 % West edge of surface
    source_nodes = [];
    n_x2 = zeros(length(source_input),1);
    for ii = 1:length(source_input)
        if isempty(SURFACES_2{source_input(ii)})
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
        current_source = source_input(ii);
        surf_nodes = SURFACES_2{current_source}.nodes;
        n_x1 = SURFACES_2{current_source}.n_x1;
        n_x2(ii) = SURFACES_2{current_source}.n_x2;
        for jj = 1:n_x2
            source_nodes = [source_nodes;surf_nodes(jj*n_x1-n_x1+1)];
        end
    end
else
    fprintf('ERROR (merge_nodes): Input_type invalid. Valid input_types are 0, 1,
2, 30, 31, 32, 33, 34.\n')
    return
end
%disp(source_nodes)

%% Find target nodes
if target_type == 0 % All nodes currently active
    target_nodes = nodes;
elseif target_type == 1 % Nodes
    for ii = 1:length(source_input)
        existcheck = MESHGRIDS_1(:,1) == target_input(ii);
        if sum(existcheck) < 1
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
    end
    target_nodes = target_input;
elseif target_type == 2 % Elements
    target_nodes = [];
    for ii = 1:length(target_input) % Grab nodes from each element and append to
list
        existcheck = ELEMENTS_1(:,1) == target_input(ii);
        if sum(existcheck) < 1
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
        target_nodes = [target_nodes, ELEMENTS_1(target_input(ii),2:5)];
```

```
    end
    target_nodes = unique(target_nodes)'; % Remove duplicates and sort
elseif target_type == 30 % Entire surface
    target_nodes = [];
    for ii = 1:length(target_input)
        if isempty(SURFACES_2{target_input(ii)})
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
        target_nodes = [target_nodes; SURFACES_2{target_input(ii)}.nodes];
    end
    target_nodes = unique(target_nodes); % Remove duplicates and sort
elseif target_type == 31 % South edge of surface
    target_nodes = [];
    n_x1 = zeros(length(target_input),1);
    for ii = 1:length(target_input)
        if isempty(SURFACES_2{target_input(ii)})
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
        current_target = target_input(ii);
        surf_nodes = SURFACES_2{current_target}.nodes;
        n_x1(ii) = SURFACES_2{current_target}.n_x1;
        target_nodes = [target_nodes;surf_nodes(1:n_x1)];
    end
    target_nodes = unique(target_nodes); % Remove duplicates and sort
elseif target_type == 32 % East edge of surface
    target_nodes = [];
    n_x2 = zeros(length(target_input),1);
    for ii = 1:length(target_input)
        if isempty(SURFACES_2{target_input(ii)})
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
        current_target = target_input(ii);
        surf_nodes = SURFACES_2{current_target}.nodes;
        n_x1 = SURFACES_2{current_target}.n_x1;
        n_x2(ii) = SURFACES_2{current_target}.n_x2;
        for jj = 1:n_x2
            target_nodes = [target_nodes;surf_nodes(jj*n_x1)];
        end
    end
    target_nodes = unique(target_nodes); % Remove duplicates and sort
elseif target_type == 33 % North edge of surface
    target_nodes = [];
    n_x1 = zeros(length(target_input),1);
    for ii = 1:length(target_input)
        if isempty(SURFACES_2{target_input(ii)})
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
```

```
        current_target = target_input(ii);
        surf_nodes = SURFACES_2{current_target}.nodes;
        n_x1(ii) = SURFACES_2{current_target}.n_x1;
        n_x2 = SURFACES_2{current_target}.n_x2;
        target_nodes = [target_nodes;surf_nodes(n_x1*n_x2-n_x1+1:end)];
    end
    target_nodes = unique(target_nodes); % Remove duplicates and sort
elseif target_type == 34 % West edge of surface
    target_nodes = [];
    n_x2 = zeros(length(target_input),1);
    for ii = 1:length(target_input)
        if isempty(SURFACES_2{target_input(ii)})
            fprintf('ERROR (merge_nodes): Value in input does not exist in the
mesh.\n')
            return
        end
        current_target = target_input(ii);
        surf_nodes = SURFACES_2{current_target}.nodes;
        n_x1 = SURFACES_2{current_target}.n_x1;
        n_x2(ii) = SURFACES_2{current_target}.n_x2;
        for jj = 1:n_x2
            target_nodes = [target_nodes;surf_nodes(jj*n_x1-n_x1+1)];
        end
    end
else
    fprintf('ERROR (merge_nodes): Input_type invalid. Valid input_types are 0, 1,
2, 30, 31, 32, 33, 34.\n')
    return
end
%disp(target_nodes)

%% Find nodes to keep and nodes to replace
source_count = length(source_nodes); % Get count of source nodes
source_dup = source_nodes; % Save original list of source nodes for later
comparison
target_count = length(target_nodes); % Get count of target nodes
target_dup = target_nodes; % Save original list of target nodes for later
comparison

% Keeping the higher index
if keep_switch == 1
    for ii = source_count:-1:1 % Iterate through source nodes
        current_source = source_nodes(ii);

        % Check if the range_tolerance is greater than 1/10th of the source node's
parent element internodal lengths
        [ind1,ind2] = find(ELEMENTS_1(:,2:5)==current_source);
        ind2 = ind2 + 1; % Note: index is relative to ELEMENTS_1
        if ind2 < 5
            checknode = ind2+1; % Check against next node in element
        else
            checknode = ind2-1; % Check against previous node in element
        end
        checknode = ELEMENTS_1(ind1,checknode); % Get actual node number of the
checknode
```

```
        temp = norm(MESHGRIDS_1(current_source,4:6)-MESHGRIDS_1(checknode,4:6));
        if range_tolerance > 0.1*temp
            fprintf('ERROR (merge_nodes): Range tolerance is too large and may find
nodes within a source node''s surface. Please input a range tolerance less than
%f.\n',0.1*temp);
            return
        end

        % Iterate through target nodes for the current source node
        for jj = target_count:-1:1
            current_target_node = target_nodes(jj);
            % Check if target node is within range
            nodal_length = norm(MESHGRIDS_1(current_source,4:6)-
MESHGRIDS_1(current_target_node,4:6));
            if nodal_length <= range_tolerance
                if current_target_node > current_source
                    % Update thermal mass
                    MESHGRIDS_1(current_target_node,2) =
MESHGRIDS_1(current_target_node,2) + MESHGRIDS_1(current_source,2);

                    % Keep the target node, replacing the source node in
                    % the list of source nodes and the list of target nodes
                    source_nodes(source_nodes == current_source) =
current_target_node;
                    target_nodes(target_nodes == current_source) =
current_target_node;
                elseif current_target_node < current_source
                    % Update thermal mass
                    MESHGRIDS_1(current_source,2) = MESHGRIDS_1(current_source,2) +
MESHGRIDS_1(current_target_node,2);

                    % Keep the source node, replacing the target node in
                    % the list of target nodes and the list of source nodes
                    source_nodes(source_nodes == current_target_node) =
current_source;
                    target_nodes(target_nodes == current_target_node) =
current_source;
                end
            end
        end
    end
elseif keep_switch == 0
    for ii = 1:source_count % Iterate through source nodes
        current_source = source_nodes(ii);

        % Check if the range_tolerance is greater than 1/10th of the source node's
parent element internodal lengths
        [ind1,ind2] = find(ELEMENTS_1(:,2:5)==current_source);
        ind2 = ind2 + 1; % Note: index is relative to ELEMENTS_1
        if ind2 < 5
            checknode = ind2+1; % Check against next node in element
        else
            checknode = ind2-1; % Check against previous node in element
        end
```

```
        checknode = ELEMENTS_1(ind1,checknode); % Get actual node number of the
checknode
        temp = norm(MESHGRIDS_1(current_source,4:6)-MESHGRIDS_1(checknode,4:6));
        if range_tolerance > 0.1*temp
            fprintf('ERROR (merge_nodes): Range tolerance is too large and may find
nodes within a source node''s surface. Please input a range tolerance less than
%f.\n',0.1*temp);
            fprintf('If a connection between distant nodes/elements/surfaces are
desired, consider using a Contactor or a Conductor.\n')
            return
        end

        % Iterate through target nodes for the current source node
        for jj = 1:target_count
            current_target_node = target_nodes(jj);
            % Check if target node is within range
            nodal_length = norm(MESHGRIDS_1(current_source,4:6)-
MESHGRIDS_1(current_target_node,4:6));
            if nodal_length <= range_tolerance
                if current_target_node < current_source
                    % Update thermal mass
                    MESHGRIDS_1(current_target_node,2) =
MESHGRIDS_1(current_target_node,2) + MESHGRIDS_1(current_source,2);

                    % Keep the target node, replacing the source node in
                    % the list of source nodes and the list of target nodes
                    source_nodes(source_nodes == current_source) =
current_target_node;
                    target_nodes(target_nodes == current_source) =
current_target_node;
                elseif current_target_node > current_source
                    % Update thermal mass
                    MESHGRIDS_1(current_source,2) = MESHGRIDS_1(current_source,2) +
MESHGRIDS_1(current_target_node,2);

                    % Keep the source node, replacing the target node in
                    % the list of target nodes and the list of source nodes
                    source_nodes(source_nodes == current_target_node) =
current_source;
                    target_nodes(target_nodes == current_target_node) =
current_source;
                end
            end
        end
    end
else
    fprintf('ERROR (merge_nodes): Keep_switch invalid. Valid keep_switch values are
1 and 0.\n')
    return
end
%source_replace = [source_nodes,source_dup];
%target_replace = [target_nodes,target_dup];
%disp(source_replace)
%disp(target_replace)
```

```
%% Full replacement
% Replace nodes in MESHGRIDS_1, ELEMENTS_1, SURFACES_2, CONDUCTANCES_1
for ii = 1:source_count
    if source_nodes(ii) ~= source_dup(ii)
        % Find source_dup(ii) in MESHGRIDS_1, ELEMENTS_1, SURFACES_2,
        % CONDUCTANCES_1 and make the replacement or removal. Ignore if
        % already done.

        % Find source_dup(ii) in MESHGRIDS_1 and remove
        MESHGRIDS_1(MESHGRIDS_1(:,1)==source_dup(ii),:) = 0;

        % Find source_dup(ii) in ELEMENTS_1 and replace
        elem_surfs = ELEMENTS_1(:,1); % Keep first column since this may be
overwritten in the next step
        change_indices = ELEMENTS_1==source_dup(ii); % This can be used to identify
which elements have been changed, and therefore which surfaces have been changed
        ELEMENTS_1(change_indices) = source_nodes(ii);
        ELEMENTS_1(:,1) = elem_surfs; % Fix the surface IDs in the ELEMENTS_1
matrix

        % Find source_dup(ii) in SURFACES_1 and replace
        change_rows = sum(change_indices,2) > 0; % Find which elements have been
changed
        change_surfs = elem_surfs(change_rows); % Extract the surfaces that have
been changed
        %disp(change_surfs)
        for jj = 1:length(change_surfs) % Iterate through each changed surface and
change nodes
            surf_nodes = SURFACES_2{change_surfs(jj)}.nodes;
            surf_nodes(surf_nodes==source_dup(ii))=source_nodes(ii);
            SURFACES_2{change_surfs(jj)}.nodes = surf_nodes;
        end

        % Find source_dup(ii) in CONDUCTANCES_1 and remove and replace.
        % Note that replacement only happens if the conductance between the
        % two nodes is zero.
        zero_locs_horz = CONDUCTANCES_1(source_nodes(ii),:) == 0;
        zero_locs_vert = CONDUCTANCES_1(:,source_nodes(ii)) == 0;
        CONDUCTANCES_1(source_nodes(ii),:) = CONDUCTANCES_1(source_nodes(ii),:) +
CONDUCTANCES_1(source_dup(ii),:).*zero_locs_horz;
        CONDUCTANCES_1(:,source_nodes(ii)) = CONDUCTANCES_1(:,source_nodes(ii)) +
CONDUCTANCES_1(:,source_dup(ii)).*zero_locs_vert;
        CONDUCTANCES_1(source_dup(ii),:) = 0;
        CONDUCTANCES_1(:,source_dup(ii)) = 0;

        % Find source_dup(ii) in CONDUCTANCES_2 and remove and replace.
        for jj = 1:length(CONDUCTANCES_2)
            if ~isempty(CONDUCTANCES_2{jj})
                mat = CONDUCTANCES_2{jj}; % Get data from cell
                [ind1,ind2,~] = find(mat(:,1:2) == source_dup(ii)); % Find where
the to-be-replaced node IDs are
                mat(ind1,ind2) = source_nodes(ii); % Replace the node IDs
                CONDUCTANCES_2{jj} = mat; % Set values in holding matrix
            end
```

```
        end
    end
end

for ii = 1:target_count
    if target_nodes(ii) ~= target_dup(ii)
        % Find target_dup(ii) in MESHGRIDS_1, ELEMENTS_1, SURFACES_2,
        % CONDUCTANCES_1 and make the replacement or removal. Ignore if
        % already done.

        % Find target_dup(ii) in MESHGRIDS_1 and remove
        MESHGRIDS_1(MESHGRIDS_1(:,1)==target_dup(ii),:) = 0;

        % Find target_dup(ii) in ELEMENTS_1 and replace
        elem_surfs = ELEMENTS_1(:,1); % Keep first column since this may be
overwritten in the next step
        change_indices = ELEMENTS_1==target_dup(ii); % This can be used to identify
which elements have been changed, and therefore which surfaces have been changed
        ELEMENTS_1(change_indices) = target_nodes(ii);
        ELEMENTS_1(:,1) = elem_surfs; % Fix the surface IDs in the ELEMENTS_1
matrix

        % Find target_dup(ii) in SURFACES_1 and replace
        change_rows = sum(change_indices,2) > 0; % Find which elements have been
changed
        change_surfs = elem_surfs(change_rows); % Extract the surfaces that have
been changed
        %disp(change_surfs)
        for jj = 1:length(change_surfs) % Iterate through each changed surface and
change nodes
            surf_nodes = SURFACES_2{change_surfs(jj)}.nodes;
            surf_nodes(surf_nodes==target_dup(ii))=target_nodes(ii);
            SURFACES_2{change_surfs(jj)}.nodes = surf_nodes;
        end

        % Find target_dup(ii) in CONDUCTANCES_1 and remove and replace
        CONDUCTANCES_1(target_nodes(ii),:) = CONDUCTANCES_1(target_nodes(ii),:) +
CONDUCTANCES_1(target_dup(ii),:);
        CONDUCTANCES_1(:,target_nodes(ii)) = CONDUCTANCES_1(:,target_nodes(ii)) +
CONDUCTANCES_1(:,target_dup(ii));
        CONDUCTANCES_1(target_dup(ii),:) = 0;
        CONDUCTANCES_1(:,target_dup(ii)) = 0;

        % Find source_dup(ii) in CONDUCTANCES_2 and remove and replace.
        for jj = 1:length(CONDUCTANCES_2)
            if ~isempty(CONDUCTANCES_2{jj})
                mat = CONDUCTANCES_2{jj}; % Get data from cell
                [ind1,ind2,~] = find(mat(:,1:2) == target_dup(ii)); % Find where
the to-be-replaced node IDs are
                mat(ind1,ind2) = target_nodes(ii); % Replace the node IDs
                CONDUCTANCES_2{jj} = mat; % Set values in holding matrix
            end
        end
    end
end
```

```
end
```

mesh_visualization_R01.m

```
function mesh_visualization_R01(back_switch,normal_switch,MESHGRIDS_1,ELEMENTS_1)
% Plots the entire mesh and displays normal vectors and backfaces upon request.

% Plot frontface
for i = 1:height(ELEMENTS_1)
    relevant_nodes = ELEMENTS_1(i,:);
    x_coords = MESHGRIDS_1(relevant_nodes(2:5),4);
    y_coords = MESHGRIDS_1(relevant_nodes(2:5),5);
    z_coords = MESHGRIDS_1(relevant_nodes(2:5),6);
    c = [1;1;1;1];
    fill3(x_coords,y_coords,z_coords,c);
    hold on

    % Plot backface
    if back_switch == 1
        vec_x1 = [x_coords(2)-x_coords(1);y_coords(2)-y_coords(1);z_coords(2)-
z_coords(1)];
        vec_x2 = [x_coords(4)-x_coords(1);y_coords(4)-y_coords(1);z_coords(4)-
z_coords(1)];
        vec_x3 = cross(vec_x1,vec_x2);
        unit_x3 = vec_x3/norm(vec_x3);

        back_off = 1e3; % Scale factor backface offset
        back_x = x_coords - unit_x3(1)/back_off;
        back_y = y_coords - unit_x3(2)/back_off;
        back_z = z_coords - unit_x3(3)/back_off;

        c = [0.5 0.5 0.5 0.5];
        backface = fill3(back_x,back_y,back_z,c);
        backface.FaceAlpha = 0.5;
    end

    % Plot normal vector
    if normal_switch == 1
        vec_x1 = [x_coords(2)-x_coords(1);y_coords(2)-y_coords(1);z_coords(2)-
z_coords(1)];
        vec_x2 = [x_coords(4)-x_coords(1);y_coords(4)-y_coords(1);z_coords(4)-
z_coords(1)];
        vec_x3 = cross(vec_x1,vec_x2);
        arrow_scale = 1;
        unit_x3 = vec_x3/norm(vec_x3)*arrow_scale;

        x_mid = mean(x_coords);
        y_mid = mean(y_coords);
        z_mid = mean(z_coords);
        quiver3(x_mid,y_mid,z_mid,unit_x3(1),unit_x3(2),unit_x3(3),'r'); % Plot
normal vector
    end
```

```
end
pbaspect([1 1 1]) % Set aspect ratio of plot to be 1:1:1
colorbar
axis equal
end
```

nodal_area_elem_normals_R01.m

```
function [elem_areas, elem_normals, nodal_areas] =
nodal_area_elem_normals_R01(MESHGRIDS_1,ELEMENTS_1)
% Calculates the surface area partitioned to each node in the listed
% elements. Also calculates normals for each element.

% Version 1.0 completed 11/16/2023

%{
%% Method 1
% Calculate area of each element using area from surfaces 1
% Partition area of each element to each node, summing up with values from other
elements

% Find non-empty surfaces
surface_list = find(~cellfun(@isempty,SURFACES_1));

% Extract areas of surfaces and input into holding matrix at the matching index
surface_areas = zeros(length(SURFACES_1));
for ii = 1:length(surface_list)
    current_surface = surface_list(ii);
    surface_areas(current_surface) = SURFACES_1{current_surface}.area;
end

% List surfaces the elements belong to and sort, then count number of elements
share each surface
elems_temp = sortrows(ELEMENTS_1(:,1));
elems_per_surf = accumarray(elems_temp,1);

% Partition areas to each element
%}

%% Method 2
% Calculate area of each element using the element nodes

% Find nodes a, b, and d or each element
node_a = ELEMENTS_1(:,2);
node_b = ELEMENTS_1(:,3);
node_d = ELEMENTS_1(:,5);

% Find point info of nodes
a_points = MESHGRIDS_1(node_a,4:6);
b_points = MESHGRIDS_1(node_b,4:6);
d_points = MESHGRIDS_1(node_d,4:6);
```

```
% Get vectors
ab_vecs = b_points-a_points;
ad_vecs = d_points-a_points;

% Create unit vectors or element normal
elem_normals = cross(ab_vecs,ad_vecs);
elem_areas = vecnorm(elem_normals,3,2); % Calculate element areas (norm of normal
vector)
elem_normals = rdivide(elem_normals,elem_areas); % Normalize to unit vectors

% Assign area to each node
quarter_areas = elem_areas/4; % Quarter of each element area
nodal_areas = zeros(height(MESHGRIDS_1),1);

for ii = 1:height(ELEMENTS_1)
    for jj = 2:5
        current_node = ELEMENTS_1(ii,jj);
        nodal_areas(current_node) = nodal_areas(current_node) + quarter_areas(ii);
    end
end

end
```

nodal_lengths_R02.m

```
function SURFACELENGTHS_1 =
nodal_lengths_R02(surfaces2edit,MESHGRIDS_1,ELEMENTS_1,SURFACELENGTHS_1)
% Generates a holding matrices of the lengths between each node. Works only
% within a surface.
% Only calculates for nodes that connect.

% Version 2.0 completed 10/19/2023

for kk = 1:length(surfaces2edit)
    surface_number = surfaces2edit(kk);
    relevant_elems = find(ELEMENTS_1(:,1)==surface_number);
    for ii = 1:length(relevant_elems)
        elem_nodes = [ELEMENTS_1(relevant_elems(ii),2:5),
ELEMENTS_1(relevant_elems(ii),2)]; % Element nodes with node_a wrapped to the back
too
        for jj = 1:4
            L = norm(MESHGRIDS_1(elem_nodes(jj),4:6)-
MESHGRIDS_1(elem_nodes(jj+1),4:6));
            SURFACELENGTHS_1(elem_nodes(jj),elem_nodes(jj+1)) = L;
            SURFACELENGTHS_1(elem_nodes(jj+1),elem_nodes(jj)) = L;
        end
    end
end
end
```

node_create_R01.m

```
function MESHGRIDS_1 =
node_create_R01(ID,thermal_mass,T_init,x_coord,y_coord,z_coord,MESHGRIDS_1)
% Creates an individual, unlinked node.
% Input the ID, the thermal mass, the initial temperature, and the global
% x, y, z coordinates.
% Version 1.0 completed 10/2/2023

if height(MESHGRIDS_1) >= ID
    if MESHGRIDS_1(ID,1) == ID
        fprintf('ERROR (create_node): Node ID already exists.\n')
        return
    end
end
MESHGRIDS_1(ID,:) = [ID, thermal_mass, T_init, x_coord, y_coord, z_coord];
end
```

node_edit_R01.m

```
function [MESHGRIDS_1,ELEMENTS_1,CONDUCTANCES_1,CONDUCTANCES_2] =
node_edit_R01(ID,action,overwrite,input,MESHGRIDS_1,ELEMENTS_1,CONDUCTANCES_1,CONDU
CTANCES_2)
% This function edits node properties and can also be used to change the
% node ID. This function does NOT allow for movement of the node. That can
% be performed with node_move.
%
% Several different actions can be performed:
% action = 0: Delete the node. Overwrite must be set to 1.
% action = 1: Change node ID
% action = 2: Change thermal mass
% action = 3: Change initial temperature
%
% If there is a nonzero value already at the position, overwrite must be
% set to 1 to confirm overwrite.
%
% Version 1.0 completed 12/14/2024


%% Check if node exists
check = find(MESHGRIDS_1(:,1)==ID, 1);
if isempty(check)
    fprintf('ERROR (node_edit): Node does not exist.\n')
    return
end

%% Make edits
if action == 0 % Delete
    % Check if node is part of an element (and also a surface)
    indices = find(ELEMENTS_1(:,2:5)==ID, 1);
    if ~isempty(indices)
        fprintf('ERROR (node_edit): Node is part of an element and cannot be
deleted.\n')
```

```
        return
    end

    % Check overwrite
    if overwrite ~= 1
        fprintf('ERROR (node_edit): Overwrite must be set to 1 to confirm deletion
of node.\n')
        return
    end

    % MESHGRIDS_1
    MESHGRIDS_1(ID,:) = 0;

    % CONDUCTANCES_1
    CONDUCTANCES_1(ID,:) = 0;
    CONDUCTANCES_1(:,ID) = 0;

    % CONDUCTANCES_2
    if ~isempty(CONDUCTANCES_2)
        for ii = 1:length(CONDUCTANCES_2)
            if ~isempty(CONDUCTANCES_2{ii})
                mat = CONDUCTANCES_2{ii};
                [ind1,~] = find(mat(:,1:2)==ID);
                mat(ind1,:) = [];
                CONDUCTANCES_2{ii} = mat;
            end
        end
    end


elseif action == 1 % Node ID
    % Check if node exists
    check = find(MESHGRIDS_1(:,1)==input,1);
    if ~isempty(check) && overwrite ~= 1
        fprintf('ERROR (node_edit): New node ID already exists. Set overwrite=1 to
overwrite the existing node.\n')
        return
    end

    % MESHGRIDS_1
    temp = MESHGRIDS_1(ID,:);
    MESHGRIDS_1(ID,:) = 0;
    MESHGRIDS_1(input,:) = temp;

    % ELEMENTS_1
    [ind1,ind2] = find(ELEMENTS_1(:,2:5)==ID);
    ind2 = ind2+1; % Fix the index since it skips one
    ELEMENTS_1(ind1,ind2) = input;

    % CONDUCTANCES_1
    temp1 = CONDUCTANCES_1(ID,:);
    temp2 = CONDUCTANCES_1(:,ID);
    CONDUCTANCES_1(ID,:) = 0;
    CONDUCTANCES_1(:,ID) = 0;
    CONDUCTANCES_1(input,:) = temp1;
```

```
    CONDUCTANCES_1(:,input) = temp2;

    % CONDUCTANCES_2
    if ~isempty(CONDUCTANCES_2)
        for ii = 1:length(CONDUCTANCES_2)
            if ~isempty(CONDUCTANCES_2{ii})
                mat = CONDUCTANCES_2{ii};
                [ind1,ind2] = find(mat(:,1:2)==ID);
                mat(ind1,ind2) = input;
                CONDUCTANCES_2{ii} = mat;
            end
        end
    end

elseif action == 2 % Thermal mass
    % Check if value is nonzero
    check = MESHGRIDS_1(ID,2);
    if check ~= 0 && overwrite ~= 1
        fprintf('ERROR (node_edit): Thermal mass already exists. Set overwrite=1 to
overwrite the existing value.\n')
        return
    end
    MESHGRIDS_1(ID,2) = input;
elseif action == 3 % Initial temperature
    % Check if value is nonzero
    check = MESHGRIDS_1(ID,3);
    if check ~= 0 && overwrite ~= 1
        fprintf('ERROR (node_edit): Initial temperature already exists. Set
overwrite=1 to overwrite the existing value.\n')
        return
    end
    MESHGRIDS_1(ID,3) = input;
else
    fprintf('ERROR (node_edit): Invalid action. Use 1 to change the node ID, 2 to
change the thermal mass, 3 to change the initial temperature.\n')
    return
end


end
```

node_move_R01.m

```
function MESHGRIDS_1 = node_move_R01(ID,relative,x1,x2,x3,MESHGRIDS_1,ELEMENTS_1)
% Moves a node as long as it is not part of a surface.
% ID - Node ID
% relative - Switch between relative movement from current origin (1) and absolute
position of origin (0)
% x1 - movement in x1 [m]
% x2 - movement in x2 [m]
% x3 - movement in x3 [m]

% Version 1.0 completed 12/12/2023
```

```
%% Check if node exists
exist = MESHGRIDS_1(:,1) == ID;
exist = sum(exist,"all");
if exist == 0
    fprintf('ERROR(node_move): Node selected does not exist.\n')
    return
end

%% Check if node is part of a surface
in_surf = ELEMENTS_1(:,2:5) == ID;
in_surf = sum(in_surf,"all");
if in_surf ~= 0
    fprintf('ERROR (node_move): Node selected is part of a surface and cannot be
moved.\n')
    return
end

%% Move
if relative == 0
    MESHGRIDS_1(ID,4:6) = [x1,x2,x3];
elseif relative == 1
    MESHGRIDS_1(ID,4:6) = MESHGRIDS(ID,4:6) + [x1,x2,x3];
else
    fprintf('ERROR (node_move): Invalid input for relative. Choose 1 for movement
relative to initial position or 0 for absolute position.\n')
    return
end


end
```

rad_group_assign_R01.m

```
function SURFACES_1 = rad_groups_assign_R01(ID,topbot,group_ID,role,sn,SURFACES_1)
% Assigns surfaces with radiation groups.
% ID - Surface ID
% topbot - Select both (1), topside (2), bottomside (3)
% group_ID - Group number
% role - Role of the surface (0=ignored, 1=active+blocker, 2=blocker)
% sn - Boolean for turning on/off spacenode vision (0=off, 1=on)

% Version 1.0 completed 1/18/2024

if role ~= 0 && role ~= 1 && role ~= 2
    fprintf('ERROR (rad_groups_assign): role must be 9 (ignored), 1
(active++blocker) or 2 (blocker).\n')
    return
end

if ID > length(SURFACES_1) || isempty(SURFACES_1{ID}) || ID < 1
    fprintf('ERROR (rad_groups_assign): Invalid surface ID.\n')
    return
```

```
end

if sn ~= 1 && sn ~= 0
    fprintf('ERROR (rad_groups_assign): Space node boolean should be 0 (off) or 1
(on). This describes the surface side(s) capabilitiy to see the space node.\n')
    % Note: Does not necessarily need to be active. Can be a blocker and
    % therefore prevents light from reaching other objects.
    return
end

% Assign surface to group
if topbot == 1
    SURFACES_1{ID}.gtop = group_ID;
    SURFACES_1{ID}.gtop_role = role;
    SURFACES_1{ID}.gtop_sn = sn;
    SURFACES_1{ID}.gbot = group_ID;
    SURFACES_1{ID}.gbot_role = role;
    SURFACES_1{ID}.gbot_sn = sn;
elseif topbot == 2
    SURFACES_1{ID}.gtop = group_ID;
    SURFACES_1{ID}.gtop_role = role;
    SURFACES_1{ID}.gtop_sn = sn;
elseif topbot == 3
    SURFACES_1{ID}.gbot = group_ID;
    SURFACES_1{ID}.gbot_role = role;
    SURFACES_1{ID}.gbot_sn = sn;
else
    fprintf('ERROR (rad_groups_assign): topbot should be 1 (both sides), 2
(topside), 3 (bottomside).\n')
    return
end
end
```

rad_proportions_R01.m

```
function RPM = rad_proportions_R01(eps,F,cutoff)
% Calculates the radiation proportionality matrix for a set of elements for
% reflections under the cutoff value.

% Version 1.0 completed 2/20/2024

[N,~] = size(F);

% Reflectivities
rho = 1 - eps;
drF = diag(rho)*F;

temp = eps .* F;
RPM = zeros(N);
diff = Inf;
while diff > cutoff
    diff = max(temp,[],"all");
    RPM = RPM + temp;
```

```
    temp = drF * temp;
end
end
```

radiation_environmental_R03.m

```
function [Qdot_total_S,Qdot_total_A,Qdot_total_I,S] =
radiation_environmental_R03(T_body,AF,vS,v_body,g,enormals,ecoords,rotms,alpha,epsi
lon,sab,elem_solar_proj_area,illum_t,illum_b,elem_body_t_VF,elem_body_b_VF,eclipse,
elem_areas,R_body)
% Calculates the heating rate of elements based on incoming radiation
% (solar, albedo, Earth IR/planetshine). Solves one radiation group at a
% time.

% vS = vector from s/c center to Sun
% v_body = vector from s/c center to orbital body
% enormals = matrix of normal vectors for ALL elements
% rotms = cell array of rotation matrices needed to flatten each element to 2D

% g = struct with the following information (group information)
% .elems = list of element IDs in this group
% .tsn = boolean vector of length(.elems) indicating which elems have sn-active
topsides
% .bsn = boolean vector of length(.elems) indicating which elems have sn-active
bottomsides
% .trole = vector of length(.elems) indicating the role of each element topside
% .brole = vector of length(.elems) indicating the role of each element bottomside
% .centers = Nx3 matrix of coordinates of each element center

% sab = solar_angle_bool = boolean vector indicating which side of an element is
illuminated by the sun

% ecoords = cell array of the coordinates of each element in 3x4 matrix format

% Version 3.0 completed 2/26/2024

%% Initialization
%TE = 255; % Earth average temperature [K]
SBC = 5.670374419e-8; % Stefan-Boltzmann Constant [W/m2/K4]
S = solar_flux_R01(norm(vS)); % Solar flux [W/m2]
vRayS = repmat(vS,length(g.elems),1) - g.centers; % From element to sun
vRay_body = repmat(v_body,length(g.elems),1) - g.centers; % From element to earth
dS = vRayS./vecnorm(vRayS,2,2); % Get unit vectors along rays
dbody = vRay_body./vecnorm(vRay_body,2,2); % Get unit vectors along rays

%% Element Prep
% Normal unit vectors
n_t = enormals(g.elems,:); % Get normal unit vectors of each element
% n_b = -n_t; % Reverse normal unit vectors for bottomside
sab = boolean(sab(g.elems)); % Reduce to relevant elements (solar angle bool)
illum_t = illum_t(g.elems); % Reduce to relevant elements (need top and bottom for
orbiting body)
illum_b = illum_b(g.elems); % Reduce to relevant elements
```

```
%%% Solar
% Pare down number of elements needed to do shadowing checks by removing those that
are not sn-active and illuminated
elems_t_bool_1 = boolean(g.tsn) & sab & ~boolean(eclipse); % Create boolean vector
indicating if each element topside is both sn-active and illuminated
elems_b_bool_1 = boolean(g.bsn) & ~sab & ~boolean(eclipse); % Create boolean vector
indicating if each element bottomside is both sn-active and illuminated
elems_t_solar_shadecheck = g.elems(elems_t_bool_1); % Reduce list of elements to
only the elements that are both sn-active and illuminated
elems_b_solar_shadecheck = g.elems(elems_b_bool_1); % Reduce list of elements to
only the elements that are both sn-active and illuminated

%%% Orbital body
% Pare down number of elements needed to do shadowing checks by removing those that
are not sn-active and illuminated
elems_t_bool_2 = boolean(g.tsn) & illum_t; % Create boolean vector indicating if
each element topside is both sn-active and illuminated
elems_b_bool_2 = boolean(g.bsn) & illum_b; % Create boolean vector indicating if
each element bottomside is both sn-active and illuminated
elems_t_body_shadecheck = g.elems(elems_t_bool_2); % Reduce list of elements to
only the elements that are both sn-active and illuminated
elems_b_body_shadecheck = g.elems(elems_b_bool_2); % Reduce list of elements to
only the elements that are both sn-active and illuminated

%%% Blockers
blockers = g.elems(g.trole ~= 0 | g.brole ~= 0); % Get list of blockers by finding
elements that are not ignored, including both topside or bottomside
blocker_centers = g.centers(blockers,:);

%%% Reduce vectors for shadowing calculations (also avoids broadcasting)
dS_t_1 = dS(elems_t_bool_1,:); % Reduce list of unit vectors
dS_b_1 = dS(elems_b_bool_1,:); % Reduce list of unit vectors
dbody_t_1 = dbody(elems_t_bool_2,:); % Reduce list of unit vectors
dbody_b_1 = dbody(elems_b_bool_2,:); % Reduce list of unit vectors

%% Solar
%%% Shadowing
parfor ii = 1:length(blockers) % Check through each source elem
    blocker = blockers(ii);
    tc = ecoords{blocker};
    tr = rotms{blocker};
    p = blocker_centers(ii,:)';
    n = n_t(ii,:)';

    o = g.centers(elems_t_solar_shadecheck,:)'; % Origin of the rays
    d = dS_t_1'; % Ray unit vector out to the sun

    [~,~,hit_check(ii,:)] = ray_elem_intersect_R04(o,d,p,n,tc,tr); % Perform
element hit check
end
unshadowed_t_solar = ~boolean(sum(hit_check,1)); % 0 = in shadow, 1 = not in shadow
hit_check = []; % Reset hit check

parfor ii = 1:length(blockers)
```

```
    blocker = blockers(ii);
    tc = ecoords{blocker};
    tr = rotms{blocker};
    p = blocker_centers(ii,:)';
    n = n_t(ii,:)';

    o = g.centers(elems_b_solar_shadecheck,:)'; % Origin of the rays
    d = dS_b_1'; % Ray unit vector out to the sun

    [~,~,hit_check(ii,:)] = ray_elem_intersect_R04(o,d,p,n,tc,tr); % Perform
element hit check
end
unshadowed_b_solar = ~boolean(sum(hit_check,1)); % 0 = in shadow, 1 = not in shadow
hit_check = []; % Reset hit check

% Elements that are unshadowed, have active-sn, have active-role (i.e. elements
that should have Qdot):
elems2calc_t_solar = elems_t_solar_shadecheck(boolean(unshadowed_t_solar));
elems2calc_b_solar = elems_b_solar_shadecheck(boolean(unshadowed_b_solar));

%%% Heating calculation
Qdot_t_solar =
S*alpha(elems2calc_t_solar,1).*elem_solar_proj_area(elems2calc_t_solar);
Qdot_b_solar =
S*alpha(elems2calc_b_solar,2).*elem_solar_proj_area(elems2calc_b_solar);

%% Orbital body
%%% Shadowing
parfor ii = 1:length(blockers) % Check through each source elem
    blocker = blockers(ii);
    tc = ecoords{blocker};
    tr = rotms{blocker};
    p = blocker_centers(ii,:)';
    n = n_t(ii,:)';

    % o = g.centers(elems_t_body_shadecheck,:)'; % Origin of the rays
    o = g.centers(elems_t_bool_2,:)';
    n_sources = enormals(elems_t_body_shadecheck,:)';
    body_edges = n_sources*R_body;
    d = (body_edges - o);
    d = d./(vecnorm(d')');
    % d = dbody_t_1'; % Ray unit vector out to the body

    [~,~,hit_check(ii,:)] = ray_elem_intersect_R04(o,d,p,n,tc,tr); % Perform
element hit check
end
unshadowed_t_body = ~boolean(sum(hit_check,1)); % 0 = in shadow, 1 = not in shadow
hit_check = []; % Reset hit check

parfor ii = 1:length(blockers)
    blocker = blockers(ii);
    tc = ecoords{blocker};
    tr = rotms{blocker};
    p = blocker_centers(ii,:)';
    n = n_t(ii,:)';
```

```
    % o = g.centers(elems_b_body_shadecheck,:)'; % Origin of the rays
    o = g.centers(elems_b_bool_2,:)';
    n_sources = -enormals(elems_b_body_shadecheck,:)';
    body_edges = n_sources*R_body;
    d = (body_edges - o);
    d = d./(vecnorm(d')');
    % d = dbody_b_1'; % Ray unit vector out to the sun

    [~,~,hit_check(ii,:)] = ray_elem_intersect_R04(o,d,p,n,tc,tr); % Perform
element hit check
end
unshadowed_b_body = ~boolean(sum(hit_check,1)); % 0 = in shadow, 1 = not in shadow


% Elements that are unshadowed, have active-sn, have active-role (i.e. elements
that should have Qdot)
elems2calc_t_body = elems_t_body_shadecheck(boolean(unshadowed_t_body));
elems2calc_b_body = elems_b_body_shadecheck(boolean(unshadowed_b_body));

% Bug fixing
% elems2calc_t_body = elems_t_body_shadecheck;
% elems2calc_b_body = elems_b_body_shadecheck;

%%% Heating calculation - Albedo
vSBody = v_body-vS;
reflection_angle = vector_angles_R03(v_body,vSBody);
Qdot_t_albedo =
S*AF*alpha(elems2calc_t_body,1).*elem_body_t_VF(elems2calc_t_body).*elem_areas(elem
s2calc_t_body)*abs(cos(reflection_angle))*(eclipse~=1);
Qdot_b_albedo =
S*AF*alpha(elems2calc_b_body,2).*elem_body_b_VF(elems2calc_b_body).*elem_areas(elem
s2calc_b_body)*abs(cos(reflection_angle))*(eclipse~=1);

%%% Heating calculation - IR
Qdot_t_IR =
SBC*(T_body^4)*epsilon(elems2calc_t_body,1).*elem_body_t_VF(elems2calc_t_body).*ele
m_areas(elems2calc_t_body);
Qdot_b_IR =
SBC*(T_body^4)*epsilon(elems2calc_b_body,2).*elem_body_b_VF(elems2calc_b_body).*ele
m_areas(elems2calc_b_body);


%% Qdot Sum
Qdot_total_S = zeros(height(enormals),1);
Qdot_total_A = zeros(height(enormals),1);
Qdot_total_I = zeros(height(enormals),1);
for ii = 1:length(elems2calc_t_solar)
    elem = elems2calc_t_solar(ii);
    Qdot_total_S(elem) = Qdot_total_S(elem) + Qdot_t_solar(ii); % Add to current
value
end
for ii = 1:length(elems2calc_b_solar)
    elem = elems2calc_b_solar(ii);
```

```
    Qdot_total_S(elem) = Qdot_total_S(elem) + Qdot_b_solar(ii); % Add to current
value
end
for ii = 1:length(elems2calc_t_body)
    elem = elems2calc_t_body(ii);
    Qdot_total_A(elem) = Qdot_total_A(elem) + Qdot_t_albedo(ii); % Add to current
value
end
for ii = 1:length(elems2calc_t_body)
    elem = elems2calc_t_body(ii);
    Qdot_total_I(elem) = Qdot_total_I(elem) + Qdot_t_IR(ii); % Add to current value
end
for ii = 1:length(elems2calc_b_body)
    elem = elems2calc_b_body(ii);
    Qdot_total_A(elem) = Qdot_total_A(elem) + Qdot_b_albedo(ii); % Add to current
value
end
for ii = 1:length(elems2calc_b_body)
    elem = elems2calc_b_body(ii);
    Qdot_total_I(elem) = Qdot_total_I(elem) + Qdot_b_IR(ii); % Add to current value
end

% Bug checking
% Qdot_total_S = Qdot_total_S/4;
% Qdot_total_B = Qdot_total_B/4;

end
```

radiation_internal_R01.m

```
function Qdot_e2e =
radiation_internal_R01(g,RPM,elem_count,Qdot_out_elems_top,Qdot_out_elems_bot)
% Calculates the surface-to-surface radiation heat transfer for a radiation
% group. Outputs to full matrix showing element to element radiation
% exchange rates.
% https://www.afs.enea.it/project/neptunius/docs/fluent/html/th/node116.htm

% RPM gives the radiation proportion matrix for each radiation group,
% listed only for the active elements. Isolate active elements using the
% trole and brole boolean vectors and the list of elements in each group.

% Version 1.0 completed 2/26/2024

elems = g.elems; % Get list of elements in rad group
tactive = g.trole == 1; bactive = g.brole == 1;
elems_t_active = elems(tactive); % Get elements that have topside active
elems_b_active = elems(bactive); % Get list of elements that have bottomside active
elems_active = [elems_t_active;elems_b_active];
Qdot_out = [Qdot_out_elems_top(elems_t_active);Qdot_out_elems_bot(elems_b_active)];


Qdot_temp = sum(RPM.*Qdot_out,1)';
Qdot_e2e = zeros(elem_count,1);
```

```
for ii = 1:length(elems_active)
    elem = elems_active(ii);
    Qdot_e2e(elem) = Qdot_e2e(elem) + Qdot_temp(ii);
end

end
```

radiation_internal_prep_R01.m

```
function out =
radiation_internal_prep_R01(g,ecoords,ray_count,rotms,epsilon,cutoff)
% Prepares element connections for internal radiation of each radiation
% group.

% g = struct giving information of the current rad group
% ecoords = coordinates of each node in the element
% ray_count = number of rays to be shot for the monte-carlo ray tracing
% (used for view factor calculations)
% epsilon = Nx2 matrix giving IR emissivity/absorptivity values for the top
% and bottom sides of each element N.
% cutoff = tolerance that kills reflection function once an emitted ray has
% negligible energy

% Get elements
elems = g.elems; % ID of elements in the radiation group
% N = length(elems); % Number of elements in the radiation group

% Get epsilon for only the group elements
eps = epsilon(elems,:);
eps_tb = [eps(:,1);eps(:,2)]'; % Concatenate top and bottom into one vector
% rho_tb = 1-eps_tb; % Get reflectivities

% Get view factors
[VF_t,VF_b,FB_t,FB_b] = internal_view_factors_R02(g,ecoords,ray_count,rotms);
% VF_t = view factor that the topside of each element in the group has of each
element in the group, including itself.
% VF_b = view factor that the bottomside of each element in the group has of each
element in the group, including itself.
% FB_t = (frontback) indicates if the view factor in VF_f is to the topside of an
element (1) or the bottomside (0) or itself (0)
% FB_b = (frontback) indicates if the view factor in VF_b is to the topside of an
element (1) or the bottomside (0) or itself (0)

assignin('base','VF_t',VF_t);
assignin('base','VF_b',VF_b);
assignin('base','FB_t',FB_t);
assignin('base','FB_b',FB_b);
% fprintf('View factors topsides of each element (uncaring of seeing top or
bottom)\n')
% checktop = [elems,sum(VF_t,2)]
% checktop_sum = sum(checktop)
% fprintf('View factors bottomsides of each element (uncaring of seeing top or
bottom)\n')
```

```
% checkbot = [elems,sum(VF_b,2)]
% checkbot_sum = sum(checkbot)

% Find only active elements that give and take radiation
tactive = g.trole == 1; bactive = g.brole == 1;
active = [tactive,bactive];

% Get matrices for topside viewing and bottomside viewing
% Four matrices of the same size each
F_tt = VF_t.*FB_t; % NxN matrix (same size as the VF matrices but now with zeros
for target bottomsides). Shows the view of each element topside to the topside of
other elements.
F_tb = VF_t.*(~FB_t); % NxN matrix (same size as the VF matrices but now with zeros
for target topsides). Shows the view of each element topside to the bottomside of
other elements.
F_bt = VF_b.*FB_b; % NxN matrix (same size as the VF matrices but now with zeros
for target bottomsides). Shows the view of each element bottomside to the topside
of other elements.
F_bb = VF_b.*(~FB_b); % NxN matrix (same size as the VF matrices but now with zeros
for target topsides). Shows the view of each element bottomside tot he bottomside
of other elements.

% Handle emissivity and reflectivity values
% F_tt_eps = F_tt.*repmat(eps(:,1)',N,1); % Get topside epsilons (Nx1) and rotate
to match the columns (1xN), then repeat to create NxN matrix, then multiply by
element.
% F_tb_eps = F_tb.*repmat(eps(:,2)',N,1);
% F_bt_eps = F_tt.*repmat(eps(:,1)',N,1);
% F_bb_eps = F_tb.*repmat(eps(:,2)',N,1);
% F_tt_rho = F_tt.*(1-repmat(eps(:,1)',N,1)); % Get topside epsilons (Nx1) and
rotate to match the columns (1xN), then repeat to create NxN matrix, then subtract
from 1 to get reflectivities, then multiply by element.
% F_tb_rho = F_tb.*(1-repmat(eps(:,2)',N,1));
% F_bt_rho = F_tt.*(1-repmat(eps(:,1)',N,1));
% F_bb_rho = F_tb.*(1-repmat(eps(:,2)',N,1));

% F_eps = [F_tt_eps, F_tb_eps; F_bt_eps, F_bb_eps]; % Concatenate to make
emissivity matrix
% F_rho = [F_tt_rho, F_tb_rho; F_bt_rho, F_bb_rho]; % Concatenate to make
reflectivity matrix
F = [F_tt,F_tb;F_bt,F_bb]; % Concatenate to get 2Nx2N matrix

% Run through reflections to get proportionality matrix
RPM = rad_proportions_R01(eps_tb,F,cutoff);
out = RPM(active,active); % Prune out inactive elements

end
```

radiation_out_R03.m

```
function [Qdot_out_nodes,Qdot_out_elems] =
radiation_out_R03(Tn1,pre_rad_out_coefficient,elem_nodes_map)
% Calculates Qdot for input elements depending on the given coefficient and
```

```
% current temperature of the nodes. This function MUST work with nodes
% since the temperature of each node may be different.

% Tn1 = Vector of current temperatures for each node.
% pre_rad_out_coefficient = Truncated vector of precalculated values for each
element. Repeated four times to match list of nodes.
% elem_nodes_map = Nx4 matrix of nodes listing the four nodes of each input
element.

% Version 3.0 completeed 4/3/2024

% Reshaped list of nodes that is in order based on elements
reshaped_nodes_vec = reshape(elem_nodes_map',[numel(elem_nodes_map),1]);

% Preallocate
Qdot_out_nodes = zeros(max(reshaped_nodes_vec),1);
Qdot_out_elems = zeros(height(elem_nodes_map),1);

% Solve
Qdot_tempval = pre_rad_out_coefficient.*(Tn1(reshaped_nodes_vec).^4);

% Sum values at nodes
for ii = 1:length(Qdot_tempval)
    % Assign to vector whose shape matches nodes vector
    current_node = reshaped_nodes_vec(ii); % Get node index
    Qdot_out_nodes(current_node,1) = Qdot_out_nodes(current_node,1) +
Qdot_tempval(ii); % Assign/add to overall vector

    % Assign to parent element
    current_elem = ceil(ii/4);
    Qdot_out_elems(current_elem,1) = Qdot_out_elems(current_elem,1) +
Qdot_tempval(ii);
end

end
```

ray_elem_intersect_R04.m

```
function [t_out,r_out,hit_check] = ray_elem_intersect_R04(o,d,p,n,tc,tr)
% Checks if the launched rays hit the target element.

% o = origins of rays
% d = unit vectors of rays
% p = point on target element
% n = normal vector of target element
% tc = coordinates of target element
% tr = rotation matrix of target element (s/c frame to target frame)

[s1,s2] = size(d); % Get size of unit vector matrix (need to know how many rays are
shot)
n = repmat(n,1,s2); % Repeat normal vector to form matrix of appropriate size
dotdn = dot(d,n); % Get dot product between each ray unit vector and the normal
vector
```

```
parallelism_check = abs(dotdn) < 1e-6; % Check if too small to create threshold for
numerical stability

t = dot(p-o,n)./dotdn; % Create t values for each ray (where each ray intersects
plane of the target element)
tbool = t>0; % Check if greater t is in front of source element
%ptbool = ~parallelism_check & tbool;
trep = repmat(t,s1,1); % Repeat t vector for easier elementwise multiplication
r = o + trep.*d; % Get r for each value t
r_2D = tr*r; % Transform position vector from s/c body frame to target element
frame
r_2D(3,:) = []; % Remove third row
tc_2D = tr*tc; % Transform coordinates of target element from s/c body frame to
target element frame
[in,on] = inpolygon(r_2D(1,:),r_2D(2,:),tc_2D(1,:),tc_2D(2,:)); % Check if points r
are within the bounds of the target element
in_on = in | on; % Combine inside points with points on the edge

hit_check = (~parallelism_check & tbool) & in_on; % Create boolean to indicate
which rays pass checks and hit the element
t_out = t;
t_out(~hit_check) = Inf; % Set failed rays
r_out = r;
r_out(:,~hit_check) = NaN; % Give bad values for bad rays
end
```

rect_elem_center_R01.m

```
function out = rect_elem_center_R01(elem_index,ELEMENTS_11,MESHGRIDS_11)
% Calculates center of a rectangular element given its index.
nodeA = ELEMENTS_11(elem_index,2); % Get node A ID
nodeC = ELEMENTS_11(elem_index,4); % Get node C ID
nodeAind = MESHGRIDS_11(:,1)==nodeA; % Get boolean index of node A
nodeCind = MESHGRIDS_11(:,1)==nodeC; % Get boolean index of node C
nodeAcoords = MESHGRIDS_11(nodeAind,4:6); % Get coords of node A
nodeCcoords = MESHGRIDS_11(nodeCind,4:6); % Get coords of node C
out = mean([nodeAcoords;nodeCcoords]); % Average
end
```

set_nodal_temperature_R02.m

```
function MESHGRIDS_1 =
set_nodal_temperature_R02(T,nodes2edit,surfaces2edit,MESHGRIDS_1,SURFACES_2)
% Sets the initial temperature of the nodes or the surfaces input
% Set nodes or surfaces = -1 if not used
% Version 2.0 completed 9/25/2023

if nodes2edit~=-1
    for i=1:length(nodes2edit)
        indeces = find(MESHGRIDS_1(:,1)==nodes2edit(i), 1);
        if isempty(indeces) % Check if node exists
```

```
            % Error handling
            fprintf('ERROR (set_nodal_temperature): Node %i does not
exist.\n',nodes2edit(i));
            return
        else
            MESHGRIDS_1(indeces(1,1),3) = T; % Set node at index with initial
temperature
        end
    end
end
if surfaces2edit~=-1
    % Check if surface exists
    if isempty(SURFACES_2)
        fprintf('ERROR (set_nodal_temperature): No surfaces currently exist.\n');
        return
    end

    for i=1:length(surfaces2edit)
        surface_number = surfaces2edit(i);
        % Check if surface exists
        if SURFACES_2{surface_number}.ID~=surface_number
            % Error handling
            fprintf('ERROR (set_nodal_temperature): Surface %i does not
exist.\n',surface_number);
            return
        else
            relevant_node_indeces = SURFACES_2{surface_number}.nodes; % Find
indeces in the grids matrix of the surface nodes relevant to this surface
            for j=1:length(relevant_node_indeces) % Iterate through the list of
relevant node indeces
                MESHGRIDS_1(relevant_node_indeces(j),3) = T; % Set the temperature
of each node
            end
        end
    end
end


end
```

set_nodal_thermal_mass_R01.m

```
function MESHGRIDS_1 =
set_nodal_thermal_mass_R01(surfaces2edit,thermophysical_name,THERMOPHYSICAL_1,MESHG
RIDS_1,SURFACES_1,SURFACES_2)
% Assigns thermal mass to the nodes given the surface and the material.
% Version 1.0 completed 9/25/2023

% Check if surfaces exist
if isempty(SURFACES_2)
    fprintf('ERROR (set_nodal_thermal_mass): No surfaces currently exist.\n');
    return
end
```

```
for i=1:length(surfaces2edit)

    surface_number = surfaces2edit(i);

    % Check if surface exists
    if surface_number <= length(SURFACES_2)
        if isempty(SURFACES_2{surface_number})
            fprintf('ERROR (set_nodal_thermal_mass): Surface %i does not
exist.\n',surface_number);
            return
        end
    else
        fprintf('ERROR (set_nodal_thermal_mass): Surface %i does not
exist.\n',surface_number);
        return
    end

    % Extract data from relevant holding matrices
    surface_area = SURFACES_2{surface_number}.area;
    n_x1 = SURFACES_2{surface_number}.n_x1;
    n_x2 = SURFACES_2{surface_number}.n_x2;
    indeces = find(contains(THERMOPHYSICAL_1,thermophysical_name)); % Find indices
of material in global storage matrix
    rho = str2double(THERMOPHYSICAL_1(indeces(1),2));
    cp = str2double(THERMOPHYSICAL_1(indeces(1),4));
    thickness = SURFACES_1{surface_number}.thickness;

    % Divide mass by nodes
    surf_node_total = n_x1*n_x2; % Total number of elements in the surface
    surface_mass = rho*surface_area*thickness; % Mass of surface [kg]
    int_node_total = (n_x1-2)*(n_x2-2); % Number of interior nodes
    corn_node_total = 4; % Number of corner nodes
    edge_node_total = surf_node_total-int_node_total-corn_node_total; % Number of
edge nodes
    divisions_total = (n_x1-1)*(n_x2-1)*4; % Total number of divisions of the
surface into quarter elements per node (num elems * 4)
    quarter_node_mass = surface_mass/divisions_total; % Mass per quarter node
    int_node_mass = 4*quarter_node_mass; % Mass of interior node
    corn_node_mass = quarter_node_mass; % Mass of corner node
    edge_node_mass = 2*quarter_node_mass; % Mass of edge node

    % Assign mass to nodes
    surface_nodes = SURFACES_2{surface_number}.nodes; % Find indeces in the grids
matrix of the surface nodes relevant to this surface
    node_mass_vec = zeros(surf_node_total,1);
    node_mass_vec(1) = corn_node_mass; % First node in surface
    node_mass_vec(surf_node_total) = corn_node_mass; % Last node in surface
    node_mass_vec(n_x1) = corn_node_mass; % Node in bottom right corner
    node_mass_vec((n_x2-1)*n_x1+1) = corn_node_mass; % Node in top left corner
    if edge_node_total ~= 0
        node_mass_vec(2:n_x1-1) = edge_node_mass; % Edge nodes along bottom
        for j = 1:n_x2-2 % Edge nodes along left side
            node_mass_vec(j*n_x1+1) = edge_node_mass;
        end
```

```
        for j = 2:n_x2-1 % Edge nodes along right side
            node_mass_vec(j*n_x1) = edge_node_mass;
        end
        node_mass_vec((n_x2-1)*n_x1+2:surf_node_total-1) = edge_node_mass; % Edge
nodes along top
    end
    if int_node_total ~= 0
        for k = 1:n_x2-2
            node_mass_vec(k*n_x1+2:k*n_x1+n_x1-1) = int_node_mass; % Interior nodes
        end
    end

    %disp(node_mass_vec)

    % Convert to thermal mass and assign to nodes in grid matrix
    for j=1:surf_node_total
        temp = surface_nodes(j);
        MESHGRIDS_1(temp,2) = node_mass_vec(j)*cp; % Set thermal mass of node
    end
end

end
```

solar_flux_R01.m

```
function S = solar_flux_R01(d)
% Outputs solar flux in W/m2 given distance from sun to target in meters.
% https://www.ess.uci.edu/~yu/class/ess200a/lecture.2.global.pdf
% Version 1.0 completed 1/5/2024

% Solar Luminosity
L = 3.83*10^26; % W

% Solar flux density
S = L./(4.*pi.*d.^2); % W/m2

end
```

solver_integrated_R08.m

```
function [full_out,nodes,epoch_keep,Qdot_tot_keep,MESHGRIDS_11,elem_nodes_map] =
solver_integrated_R08(bulk_data_file,RK,t_data,plot_switch,rad_switches,env_rad_typ
e,env_data_paths,env_rad_data,s2s_rad_data,workspace_name)
% Other saved values: Qdot_out_keep, Qdot_solar_nodes_keep, Qdot_solar_elems_keep,
Qdot_body_nodes_keep, Qdot_body_elems_keep, Qdot_e2e_keep, vS_keep, vBody_keep

% Solver of the transient thermal analysis problem. Takes a bulk data
% file, spits out output temperatures which are both saved and plotted.
% This version sockets with FF to output the temperatures at
% each time step.
```

```
% Version 8.0 completed 4/5/2024

fprintf('---SOLVER RUNNING---\n')

% Importing java libraries to create sockets
import java.net.*
import java.io.*

%% Solver Initialization

% bulk_data_file = path to the bulk data file (string)
% RK = choose RK1 (Forward Euler) or RK4 for stepping
% t_data = holds time inputs for time step size and final time if needed: [dt, tf]
% plot_switch = turn on plot (0 or 1)
% rad_switches = radiation out, environmental radiation, internal radiation vector
of 0s (off) and 1s (on)
% env_rad_type = define if the propagator information for the environmental
radiation is from a saved file (0) or from a concurrently run FF file (1)
% env_data_paths = path to the saved file or a string vector with: [FreeFlyerPath,
PathToMissionPlanFolder, MissionPlanName]
% env_rad_data = input data for environmental radiation (input ~ if not used):
[R_body, T_body, albedo_factor]
% s2s_rad_data = input data for internal radiation (input ~ if not used):
[reflection_cutoff, ray_count]

if rad_switches(1) == 0
    fprintf('Radiation rejection is turned OFF\n')
elseif rad_switches(1) == 1
    fprintf('Radiation rejection is turned ON\n')
else
    fprintf('Error with rad_switches element 1 input.\n')
    return
end
if rad_switches(2) == 0
    fprintf('Environmental radiation is turned OFF\n');
    t0 = 0; % Initial time
    dt = t_data(1);
    tf = t_data(2);
    t = t0:dt:tf;
    nt = length(t);
    fprintf('Simulation start time: t0 = %f\nSimulation end time: tf = %f\nInitial
step size: dt = %f\nNumber of time steps: nt = %i\n',t0,tf,dt,nt);
    vS = 0;
    vBody = 0;

elseif rad_switches(2) == 1
    fprintf('Environmental radiation is turned ON\n')
    R_body = env_rad_data(1);
    T_body = env_rad_data(2);
    albedo_factor = env_rad_data(3);

    AF = albedo_factor; % Albedo factor (recommended to use 0.36 per Thornton book)

    if env_rad_type == 0
        input_file = env_data_paths;
```

```
        % Read input file
        [epoch_vec,vS_mat,vBody_mat,eclipse_vec] =
input_file_parser_R01(input_file);
        t = epoch_vec - epoch_vec(1);
        t = t(2:end); % Vector of times
        dt_vec = epoch_vec(2:end) - epoch_vec(1:end-1); % Vector of time steps
        nt = length(dt_vec);
        fprintf('Set number of time steps: %i\n',nt)
        fprintf('Data successfully read in from text file.\n')
    elseif env_rad_type == 1
        FreeFlyerPath = env_data_paths{1};
        PathToMissionPlanFolder = env_data_paths{2};
        MissionPlanName = env_data_paths{3};
        t = 0;
    else
        fprintf('Error with rad_env_type in solver_integrated input.\n');
        return
    end
else
    fprintf('Error with rad_switches element 2 input.\n');
    return
end
if rad_switches(3) == 0
    fprintf('Surface-to-surface heat exchange is turned OFF\n');
elseif rad_switches(3) == 1
    fprintf('Surface-to-surface heat exchange is turned ON\n');
    reflection_cutoff = s2s_rad_data(1);
    ray_count = s2s_rad_data(2);

    cutoff = reflection_cutoff; % Cutoff value for reflections
else
    fprintf('Error with rad_switches element 3 input.\n');
end


SBC = 5.670374419*10^(-8); % Stefan-Boltzmann Constant [W/m2/K4]



%% Bulk Data Processing
% Read bulk data file
[overview, MESHGRIDS_11, SURFACES_11, ~, ~, THERMOOPTICAL_11, ELEMENTS_11,
CONDUCTANCES_11, HEATLOADS_11] = bulk_data_parser_R02(bulk_data_file);

% Overview
node_count = overview(1);
elem_count = overview(2);

% Prepare mass and temp
nodes = MESHGRIDS_11(:,1);
Cth = MESHGRIDS_11(:,2);
T = MESHGRIDS_11(:,3);

% Rename for clarity or brevity
G = CONDUCTANCES_11;
```

```
Qdot_HL = HEATLOADS_11;

% Store temperatures
% T_keep1 = Inf*ones(length(nodes),nt);
T_keep1(:,1) = T;
MESHGRIDS_12 = MESHGRIDS_11; % Hold temperatures and nodal information for the
current step

%%% Prep surface values
surf_gtop = SURFACES_11(:,10);
surf_gtop_sn = SURFACES_11(:,12);
surf_gbot = SURFACES_11(:,13);
surf_gbot_sn = SURFACES_11(:,15);

%%% Prep element info
elem_ID = (1:elem_count)';
elem_nodes = ELEMENTS_11(:,2:5);
[~,elem_nodes_map] = ismember(elem_nodes,nodes); % Map nodes within elements
elem_areas = ELEMENTS_11(:,6);
elem_normals = ELEMENTS_11(:,7:9);
elem_alpha = zeros(height(ELEMENTS_11),2);
elem_epsilon = zeros(height(ELEMENTS_11),2);
elem_gtop = zeros(height(ELEMENTS_11),1); elem_gtop_role = elem_gtop; elem_gtop_sn
= elem_gtop;
elem_gbot = zeros(height(ELEMENTS_11),1); elem_gbot_role = elem_gbot; elem_gbot_sn
= elem_gbot;
for ii = 1:elem_count
    surf_index = find(ELEMENTS_11(ii,1)==SURFACES_11(:,1));
    elem_alpha(ii,1) = THERMOOPTICAL_11(SURFACES_11(surf_index,4),2); % Topside
    elem_alpha(ii,2) = THERMOOPTICAL_11(SURFACES_11(surf_index,5),2); % Bottomside
    elem_epsilon(ii,1) = THERMOOPTICAL_11(SURFACES_11(surf_index,4),3); % Topside
    elem_epsilon(ii,2) = THERMOOPTICAL_11(SURFACES_11(surf_index,5),3); %
Bottomside
    elem_gtop(ii,1) = SURFACES_11(surf_index,10); % Group ID of topside for this
element
    elem_gtop_role(ii,1) = SURFACES_11(surf_index,11); % Role (0=ignored,
1=active+blocker, 2=blocker) of topside for this element
    elem_gtop_sn(ii,1) = SURFACES_11(surf_index,12); % Boolean indicator of
spacenode visibility for the topside for this element
    elem_gbot(ii,1) = SURFACES_11(surf_index,13); % Group ID of bottomside for this
element
    elem_gbot_role(ii,1) = SURFACES_11(surf_index,14); % Role (0=ignored,
1=active+blocker, 2=blocker) of bottomside for this element
    elem_gbot_sn(ii,1) = SURFACES_11(surf_index,15); % Boolean indicator of
spacenode visibility for the bottomside for this element
end
elem_gtop_sn = logical(elem_gtop_sn); % Convert to boolean/logical
elem_gbot_sn = logical(elem_gbot_sn); % Convert to boolean/logical

%% Radiation Preparation
if sum(rad_switches) > 0
    %%% Element center coordinates
    elem_center = zeros(elem_count,3);
    for ii = 1:elem_count
        elem_center(ii,:) = rect_elem_center_R01(ii,ELEMENTS_11,MESHGRIDS_11);
```

```
    end

    %%% Prep for rad_out
    % Isolate and map elements to go into rad_out function
    elem_top_rad_out = NaN; elem_bot_rad_out = NaN; % Initialize values
    pre_rad_out_top = NaN; pre_rad_out_bot = NaN; % Initialize values
    if sum(elem_gtop_role==1) ~= 0
        elem_top_rad_out = elem_ID(elem_gtop_role==1); % Get IDs of only active
elements. Needed for later remapping.
        % pre_rad_out_top =
SBC*elem_epsilon(elem_gtop_role==1,1).*elem_areas(elem_gtop_role==1); % Prepare
topside coefficient.
        % pre_rad_out_top = repelem(pre_rad_out_top/4,4); %
pre_rad_out_coefficients is by elements. The values in this vector need to be
repeated four times each (and divided by 4) to match the number of nodes.
        % elem_top_rad_out_mapped_nodes = elem_nodes_map(elem_gtop_role==1,:); %
Get relevant nodes with new indices
        % elem_top_rad_out = elem_gtop_role==1;
        pre_rad_out_top = SBC*elem_epsilon(:,1).*elem_areas; % Elemental
        pre_rad_out_top = pre_rad_out_top.*(elem_gtop_role==1); % Set values for
elems that don't radiate to zero
        pre_rad_out_top = repelem(pre_rad_out_top/4,4); % Convert to nodal

    end
    if sum(elem_gbot_role==1) ~= 0
        elem_bot_rad_out = elem_ID(elem_gbot_role==1); % Get IDs of only active
elements. Needed for later remapping.
        % pre_rad_out_bot =
SBC*elem_epsilon(elem_gbot_role==1,2).*elem_areas(elem_gbot_role==1); % Prepare
bottomside_coefficient
        % pre_rad_out_bot = repelem(pre_rad_out_bot/4,4); %
pre_rad_out_coefficients is by elements. The values in this vector need to be
repeated four times each (and divided by 4) to match the number of nodes.
        % elem_bot_rad_out_mapped_nodes = elem_nodes_map(elem_gbot_role==1,:); %
Get relevant nodes with new indices
        % elem_bot_rad_out = elem_gbot_role==1;
        pre_rad_out_bot = SBC*elem_epsilon(:,2).*elem_areas;
        pre_rad_out_bot = pre_rad_out_bot.*(elem_gbot_role==1);
        pre_rad_out_bot = repelem(pre_rad_out_bot/4,4);
    end


    %%% Prep for rad_environmental
    % Environmental radiation should implement radiation groups as there should be
    % shadowing of surfaces. Shadowing should be calculated on an element
    % basis if computation speed allows.
    % Environmental radiation is assumed ON if the space node boolean is activated
    % for the surface. This means that the surface, even if it is a blocker or
    % ignored within a radiation group can experience incoming external
    % radiation. The group role only determines the object's role within the
    % radiation group. Therefore, all elements part of a group within which at
    % least one elem/surf experiences external radiation need to be passed into
    % the external radiation function.
    if rad_switches(2) == 1 || rad_switches(3) == 1
        % Find groups with at least one surface that sees sn
```

```
        radgroups_rad_env = unique([surf_gtop(surf_gtop_sn ==
1);surf_gbot(surf_gbot_sn == 1)]); % Get list of rad group IDs
        g = cell(1,length(radgroups_rad_env));
        for ii = 1:length(radgroups_rad_env) % Iterate through each group
            % Get current group ID
            gID = radgroups_rad_env(ii);

            % Get list of elements
            f1 = find(elem_gtop==gID);
            f2 = find(elem_gbot==gID);
            elems = unique([f1;f2]);

            % Store info
            h.ID = gID;
            h.elems = elems;
            h.tsn = boolean(elem_gtop_sn(elems));
            h.bsn = boolean(elem_gbot_sn(elems));
            h.trole = elem_gtop_role(elems);
            h.brole = elem_gbot_role(elems);
            h.centers = elem_center(elems,:);

            % Store in group cell array
            g{ii} = h;
        end

        % Prepare rotation matrices for each element to transform the plane and any
        % points on the plane to a 2D projection (goes from s/c body frame to
        % element frame. To go from element frame to s/c body frame left multiply
by
        % the transpose of this matrix.)
        elem_rotms = cell(1,elem_count);
        for ii = 1:elem_count
            unit_norm = [0,0,1]; % Create a vector to align with
            if sum(cross(elem_normals(ii,:),unit_norm),"all")==0
                if dot(unit_norm,elem_normals(ii,:)) >= 0
                    R = eye(3);
                else
                    R = -eye(3);
                end
            else
                R = create_rotmat_R01(elem_normals(ii,:),unit_norm);
            end
            elem_rotms{ii} = R;
        end

        % Get coordinates of each point in the element
        ecoords = cell(1, elem_count); % Preallocate cell array
        for ii = 1:elem_count
            current_nodes = elem_nodes_map(ii,:);
            ecoords{ii} =
[MESHGRIDS_11(current_nodes(1),4:6)',MESHGRIDS_11(current_nodes(2),4:6)',...

MESHGRIDS_11(current_nodes(3),4:6)',MESHGRIDS_11(current_nodes(4),4:6)'];
        end
    end
```

```
    %%% Prep for radiation_internal
    if rad_switches(3) == 1
        % Radiation proportion calculations
        fprintf('Calculating internal view factors and RPM.\n');
        RPM = cell(1,length(g));
        for ii = 1:length(g)
            RPM{ii} =
radiation_internal_prep_R01(g{ii},ecoords,ray_count,elem_rotms,elem_epsilon,cutoff)
;
        end
        assignin(workspace_name,'RPMs',RPM)
    end
end
fprintf('Solver initialization complete.\n');

% check = sum(RPM{1},2)

%% FF connection
if rad_switches(2) == 1 && env_rad_type == 1

    showGUI = 0; % Open FF GUI if desired
    terminationCode = '10191980'; % Tells Matlab when to end Java read

    %%% Create 2 sockets per FF instance
    % Try to open socket 1 using the OS-determined port
    try
        socketServer(1) = ServerSocket(0); % JAVA command
    catch err
        error(strcat('Error: Unable to open socket. ', getReport(err)))
    end
    % Get the port number used for the above socket
    portNum(1) = socketServer(1).getLocalPort(); % JAVA command

    % Try to open socket 2 using the OS-determined port
    try
        socketServer(2) = ServerSocket(0); % JAVA command
    catch err
        error(strcat('Error: Unable to open socket. ', getReport(err)))
    end
    % Get the port number used for the above socket
    portNum(2) = socketServer(2).getLocalPort(); % JAVA command

    %%% Launch FF Instance
    % Create command-line string to be executed
    if boolean(showGUI)
       commandString = strcat('"', FreeFlyerPath, 'FreeFlyer.exe"', ...
           ' -r -mp "', PathToMissionPlanFolder, MissionPlanName,'"', ...
           sprintf(' -ui %d -ui %d -ui %s &', portNum(1), portNum(2),
terminationCode));
    else
       commandString = strcat('"', FreeFlyerPath, 'FF.exe"', ...
           ' -mp "', PathToMissionPlanFolder, MissionPlanName,'"', ...
           sprintf(' -ui %d -ui %d -ui %s  &', portNum(1), portNum(2),
terminationCode));
```

```
    end

    % Execute command at command-line.  Other execution commands are dos() and
    % !.  However, both have subtle differences from system()
    disp(commandString)
    system( commandString );

    % Wait for the FF client instance to connect.
    socketServer(1).setSoTimeout(int16(10000)); % JAVA command

    % Open first socket.
    try
        socketClient(1) = socketServer(1).accept(); % JAVA command
    catch err
        error('Error: Unable to accept MissionPlan as client')
    end

    socketServer(2).setSoTimeout(int16(10000)); % JAVA command

    %open second socket
    try
        socketClient(2) = socketServer(2).accept(); % JAVA command
    catch err
        error('Error: Unable to accept MissionPlan as client')
    end

    %%% Create write/read buffer
    outputStream = DataOutputStream(socketClient(1).getOutputStream()); % JAVA
command
    inputStream  = DataInputStream(socketClient(2).getInputStream()); % JAVA
command
    inputReader = BufferedReader(InputStreamReader(inputStream)); % JAVA command

    %%% Get initial set of data
    % Get information from orbital solver
    % Request data from FF client
    orbit_data_request = 1;
    outputStream.writeDouble(orbit_data_request); % JAVA command

    %%%  Wait for FF Instances to finish running
    %{
    This very simply listens to the data being input through the stream and
    only increments when non-null data is read in.  This reader will only break
    when it receives the end-loop command of '10191980'.
    %}
    index = 1;
    while 1
        JAVAdata = inputReader.readLine(); % JAVA command
        rawData{ index } = char(JAVAdata);

        if ( strcmp(terminationCode,rawData{ index }) == 1 )
            break
        end

        index = index + 1;
```

```
    end

    % Convert data
    received_data = str2num(rawData{1});
    epoch_keep = received_data(1);
end



%% Sim
fprintf('Beginning simulation...\n');
% fprintf('Total Time [s] = %2.1f\nCurrent time [s] = ',t(end)) % Set up timer
output
t0 = 0;
time_out = fprintf('%2.1f',t0); % Display initial time
loop_number = 2;
end_loop = 0;
while end_loop ~= 1

    Tn1 = T_keep1(:,loop_number-1); % Current temperature of all nodes

    %%% Get information from orbital solver
    if rad_switches(2) == 1 && env_rad_type == 0
        vS = vS_mat(loop_number-1,:);
        vBody = vBody_mat(loop_number-1,:);
        eclipse = eclipse_vec(loop_number-1,:);
        dt = dt_vec(loop_number-1);
    elseif rad_switches(2) == 1 && env_rad_type == 1
        % Request data from FF client
        orbit_data_request = 1;
        outputStream.writeDouble(orbit_data_request); % JAVA command

        %%%  Wait for FF Instances to finish running
        %{
        This very simply listens to the data being input through the stream and
        only increments when non-null data is read in.  This reader will only break
        when it receives the end-loop command of '10191980'.
        %}
        index = 1;
        while 1
            JAVAdata = inputReader.readLine(); % JAVA command
            rawData{ index } = char(JAVAdata);

            if ( strcmp(terminationCode,rawData{ index }) == 1 )
                break
            end

            index = index + 1;

        end

        % Convert data
        received_data = str2num(rawData{1});
        epoch_keep(loop_number) = received_data(1); % Keep epoch
```

```
        t(loop_number) = received_data(1) - epoch_keep(1); % Get sim time
        dt = epoch_keep(loop_number) - epoch_keep(loop_number-1);
        vS = received_data(2:4)*1000;
        vBody = received_data(5:7)*1000;
        eclipse = received_data(8);
        end_loop = received_data(9);
        if end_loop == 1
            break
        end
    end
    vS_keep(loop_number-1,:) = vS;
    vBody_keep(loop_number-1,:) = vBody;


    %%% Radiation out
    % Calculate Qdot_out for each node given the immediate area around it.
    % Use pre_rad_out_top and bot and divide to each node in element.
    % This requires finding each node per element and doing an assignment.
    % Check nodal masses function for calculating the area per node.
    % Sum total radiation out and store for later checks.
    if rad_switches(1) == 1 || rad_switches(3) == 1
        % Input active elements into rad_out function
        Qdot_out_nodes_top = zeros(node_count,1); Qdot_out_nodes_bot =
zeros(node_count,1); % Initialize both in case one is not solved.
        Qdot_out_elems_top = zeros(elem_count,1); Qdot_out_elems_bot =
zeros(elem_count,1);
        if ~isnan(elem_top_rad_out(1)) % Check if NaN by checking first value only.
            % Work with element topsides
            % Qdot_out_top =
radiation_out_R01(Tn1,pre_rad_out_top,elem_top_rad_out_mapped_nodes,node_count); %
Note: this should match new indices of nodes. Output is per node.
            % [Qdot_out_nodes_top,Qdot_out_elems_top] =
radiation_out_R02(Tn1,pre_rad_out_top,elem_top_rad_out_mapped_nodes,node_count,elem
_top_rad_out,elem_count);
            % [Qdot_out_nodes_top,Qdot_out_elems_top] =
radiation_out_R02(Tn1,pre_rad_out_top,elem_nodes_map,node_count,elem_top_rad_out,el
em_count);
            [Qdot_out_nodes_top,Qdot_out_elems_top] =
radiation_out_R03(Tn1,pre_rad_out_top,elem_nodes_map);
        end
        if ~isnan(elem_bot_rad_out(1)) % Check if NaN by checking first value only.
            % Work with element bottomsides
            % Qdot_out_bot =
radiation_out_R01(Tn1,pre_rad_out_bot,elem_bot_rad_out_mapped_nodes,node_count); %
Note: this should match new indices of nodes. Output is per node.
            % [Qdot_out_nodes_bot,Qdot_out_elems_bot] =
radiation_out_R02(Tn1,pre_rad_out_bot,elem_bot_rad_out_mapped_nodes,node_count,elem
_bot_rad_out,elem_count);
            [Qdot_out_nodes_bot,Qdot_out_elems_bot] =
radiation_out_R03(Tn1,pre_rad_out_bot,elem_nodes_map);
        end
        Qdot_out_nodes = Qdot_out_nodes_top + Qdot_out_nodes_bot; % Sum
        % Qdot_out_elems = Qdot_out_elems_top + Qdot_out_elems_bot;
    else
        Qdot_out_nodes = zeros(node_count,1);
```

```
    end

    Qdot_out_keep(:,loop_number-1) = Qdot_out_nodes;

    %%% Environmental radiation
    % Calculate Qdot_in for each element given its view of the space node.
    % Input vectors to sun and earth. Check against top and bottom sides of
    % elements that are part of active surfaces.
    if rad_switches(2) == 1
        % Vector to sun, vector to orbital body
        vSrep = repmat(vS,elem_count,1); % Prepare by repeating the vS vector for
each element to make elem_count x 3 matrix
        vbodyrep = repmat(vBody,elem_count,1); % Prepare by repeating the vbody
vector for each element to make elem_count x 3 matrix

        % Solar incidence angles
        solar_incidence_angles = vector_angles_R03(elem_normals,vSrep); % Get
angles of incidence for each element
        solar_angle_bool = (solar_incidence_angles < pi/2); % Get boolean
indicating if seeing top or bottomside
        elem_solar_proj_area = abs(elem_areas.*cos(solar_incidence_angles)); % Get
the projected area relative to the angle

        % Orbital body incidence angles and view factors (topside)
        theta_t = vector_angles_R03(elem_normals,vbodyrep); % Get angles of
incidence for each element
        sintheta_t = sin(theta_t);
        costheta_t = cos(theta_t);
        cottheta_t = cot(theta_t);

        % Orbital body incidence angles and view factors (bottomside)
        theta_b = vector_angles_R03(-elem_normals,vbodyrep);
        sintheta_b = sin(theta_b);
        costheta_b = cos(theta_b);
        cottheta_b = cot(theta_b);

        eta = asin(R_body/norm(vBody)); % Maximum angle subtended by Earth (angle
between nadir vector of s/c and line drawn from satellite to horizon of Earth)
        H = norm(vBody)/R_body;
        % elem_body_proj_area_t = abs(elem_areas.*costheta_t); % Get the projected
area relative to the topside angle
        % elem_body_proj_area_b = abs(elem_areas.*costheta_b); % Get the projected
area relative to the bottomside angle

        % Create boolean vectors indicating which sides are visible
        illum_topside_only = theta_t < (pi/2 - eta);
        illum_bottomside_only = theta_t > (pi/2 + eta);
        illum_both_sides = theta_t <= pi/2 + eta & theta_t > pi/2 - eta;
        illum_t = illum_topside_only | illum_both_sides; % Boolean true if topside
is illuminated either way
        illum_b = illum_bottomside_only | illum_both_sides;

        % Compose view factor vectors
        elem_body_t_VF = zeros(size(illum_topside_only));
        simple_t_VF = costheta_t/(H^2);
```

```
        complex_t_VF = (2/pi)*(pi/4 - 0.5*asin(sqrt(H^2-
1)./(H*sintheta_t))+(1/(2*H^2)).*(costheta_t.*acos(-sqrt(H^2-1).*cottheta_t)-
sqrt(H^2-1).*sqrt(1-H^2*(costheta_t.^2))));
        elem_body_t_VF(illum_topside_only) = simple_t_VF(illum_topside_only);
        elem_body_t_VF(illum_both_sides) = complex_t_VF(illum_both_sides);

        elem_body_b_VF = zeros(size(illum_bottomside_only));
        simple_b_VF = costheta_b/(H^2);
        complex_b_VF = (2/pi)*(pi/4 - 0.5*asin(sqrt(H^2-
1)./(H*sintheta_b))+(1/(2*H^2)).*(costheta_b.*acos(-sqrt(H^2-1).*cottheta_b)-
sqrt(H^2-1).*sqrt(1-H^2*(costheta_b).^2)));
        elem_body_b_VF(illum_bottomside_only) = simple_b_VF(illum_bottomside_only);
        elem_body_b_VF(illum_both_sides) = complex_b_VF(illum_both_sides);

        Qdot_solar_elems = zeros(elem_count,length(radgroups_rad_env));
        Qdot_albedo_elems = zeros(elem_count,length(radgroups_rad_env));
        Qdot_IR_elems = zeros(elem_count,length(radgroups_rad_env));
        parfor ii = 1:length(radgroups_rad_env)

[Qdot_solar_elems(:,ii),Qdot_albedo_elems(:,ii),Qdot_IR_elems(:,ii),S_to_keep(ii)]
=
radiation_environmental_R03(T_body,AF,vS,vBody,g{ii},elem_normals,ecoords,elem_rotm
s,elem_alpha,elem_epsilon,solar_angle_bool,elem_solar_proj_area,illum_t,illum_b,ele
m_body_t_VF,elem_body_b_VF,eclipse,elem_areas,R_body);
        end
        Qdot_solar_elems = sum(Qdot_solar_elems,2);
        Qdot_albedo_elems = sum(Qdot_albedo_elems,2);
        Qdot_IR_elems = sum(Qdot_IR_elems,2);
        S_keep(loop_number-1,1) = S_to_keep(1);
    else
        Qdot_solar_elems = zeros(elem_count,1);
        Qdot_albedo_elems = zeros(elem_count,1);
        Qdot_IR_elems = zeros(elem_count,1);
    end
    Qdot_solar_nodes = elem2nodes_R01(Qdot_solar_elems,elem_nodes_map,node_count);
    Qdot_solar_nodes_keep(:,loop_number-1) = Qdot_solar_nodes;
    Qdot_solar_elems_keep(:,loop_number-1) = Qdot_solar_elems;
    Qdot_albedo_nodes =
elem2nodes_R01(Qdot_albedo_elems,elem_nodes_map,node_count);
    Qdot_albedo_nodes_keep(:,loop_number-1) = Qdot_albedo_nodes;
    Qdot_albedo_elems_keep(:,loop_number-1) = Qdot_albedo_elems;
    Qdot_IR_nodes = elem2nodes_R01(Qdot_IR_elems,elem_nodes_map,node_count);
    Qdot_IR_nodes_keep(:,loop_number-1) = Qdot_IR_nodes;
    Qdot_IR_elems_keep(:,loop_number-1) = Qdot_IR_elems;


    %%% Internal radiation
    if rad_switches(3) == 1
        % Note: Should be entirely balanced within the satellite. Include a check.
May need to check against Qdot_out.
        Qdot_e2e = zeros(elem_count,length(radgroups_rad_env));
        parfor ii = 1:length(radgroups_rad_env)
            Qdot_e2e(:,ii) =
radiation_internal_R01(g{ii},RPM{ii},elem_count,Qdot_out_elems_top,Qdot_out_elems_b
ot);
```

```
        end
        Qdot_e2e = sum(Qdot_e2e,2);
    else
        Qdot_e2e = zeros(elem_count,1);
    end
    Qdot_e2e_nodes = elem2nodes_R01(Qdot_e2e,elem_nodes_map,node_count);
    Qdot_e2e_keep(:,loop_number-1) = Qdot_e2e_nodes;

    %%% Sum heats on each element (include env rad results) and separate out to
nodes
    % Note: already separated out to nodes for radiation out.
    % Qdot_elems = Qdot_env + Qdot_e2e;
    % Qdot_in = elem2nodes_R01(Qdot_elems,elem_nodes_map,node_count);

    %%% Sum nodal heating rates
    if rad_switches(1) == 0
        Qdot_out_nodes = 0;
    end
    if rad_switches(2) == 0
        Qdot_solar_nodes = 0;
        Qdot_body_nodes = 0;
    end
    if rad_switches(3) == 0
        Qdot_e2e_nodes = 0;
    end
    Qdot_tot = Qdot_solar_nodes + Qdot_albedo_nodes + Qdot_IR_nodes +
Qdot_e2e_nodes - Qdot_out_nodes;
    Qdot_tot_keep(:,loop_number-1) = Qdot_tot;


    %%% Solver
    if RK == 1
        % Heat transfer with RK1
        deltaT = Tn1.'-Tn1;
        k1 = (Qdot_HL + Qdot_tot + sum(G.*deltaT,2))./Cth;
        T_change = k1*dt;
        T = Tn1 + T_change;
        MESHGRIDS_12(:,3) = T;
        T_keep1(:,loop_number) = T;
    elseif RK == 4
        % Heat transfer with RK4
        deltaT = Tn1.'-Tn1;
        k1 = (Qdot_HL + Qdot_tot + sum(G.*deltaT,2))./Cth;
        deltaT = (Tn1+k1/2*dt).'-(Tn1+k1/2*dt);
        k2 = (Qdot_HL + Qdot_tot + sum(G.*deltaT,2))./Cth;
        deltaT = (Tn1+k2/2*dt).'-(Tn1+k2/2*dt);
        k3 = (Qdot_HL + Qdot_tot + sum(G.*deltaT,2))./Cth;
        deltaT = (Tn1+k3*dt).'-(Tn1+k3*dt);
        k4 = (Qdot_HL + Qdot_tot + sum(G.*deltaT,2))./Cth;
        T_change = (k1+2*k2+2*k3+k4)*dt/6;
        T = Tn1 + T_change;
        MESHGRIDS_12(:,3) = T;
        T_keep1(:,loop_number) = T;
    elseif RK == 15
        fun = @(T,Tn) (Qdot_HL + Qdot_tot + sum(G.*(Tn.'-Tn),2))./Cth;
```

```
        tspan = [t(loop_number-1),t(loop_number)];
        [~,T] = ode15s(fun,tspan,Tn1);
        MESHGRIDS_12(:,3) = T(end,:)';
        T_keep1(:,loop_number) = T(end,:)';
    elseif RK == 45
        fun = @(T,Tn) (Qdot_HL + Qdot_tot + sum(G.*(Tn.'-Tn),2))./Cth;
        tspan = [t(loop_number-1),t(loop_number)];
        [~,T] = ode45(fun,tspan,Tn1);
        MESHGRIDS_12(:,3) = T(end,:)';
        T_keep1(:,loop_number) = T(end,:)';
    elseif RK == 89
        fun = @(T,Tn) (Qdot_HL + Qdot_tot + sum(G.*(Tn.'-Tn),2))./Cth;
        tspan = [t(loop_number-1),t(loop_number)];
        [~,T] = ode89(fun,tspan,Tn1);
        MESHGRIDS_12(:,3) = T(end,:)';
        T_keep1(:,loop_number) = T(end,:)';
    else
        fprintf('ERROR (solver_integrated): Invalid input for RK. Use either RK=1,
RK=4, or RK=45.\n')
        return
    end

    %%% Plot
    if plot_switch == 1
        figure(3)
        surface_post_visualization_R02(MESHGRIDS_12,ELEMENTS_11) % Plot
    end

    %%% Gif
    %exportgraphics(gca,"checkout_surface_01_01.gif","Append",true)

    %%% Timer
    if rad_switches(2) == 1 && env_rad_type == 1
        fprintf(repmat('\b',1,time_out))
        time_out = fprintf('%2.1f',epoch_keep(loop_number)-epoch_keep(1));
    elseif rad_switches(2) == 1 && env_rad_type == 0
        fprintf(repmat('\b',1,time_out))
        time_out = fprintf('%2.1f',t(loop_number));
        epoch_keep = epoch_vec;
    else
        fprintf(repmat('\b',1,time_out))
        time_out = fprintf('%2.1f',t(loop_number));
        epoch_keep = t;
    end

    %%% Update loop number and check for end
    % Note that end_loop will handled by FF if using env_rad_type == 1
    if rad_switches(2) == 0
        % fprintf('Test 1. nt = %i, loop number = %i\n',nt,loop_number);
        if loop_number >= nt
            end_loop = 1;
        end
    elseif rad_switches(2) == 1
        if env_rad_type == 0
           if loop_number >= nt
```

```
                end_loop = 1;
            end

            % The ELSE is handled by FF passing end_loop=1 to matlab
        end
    end

    loop_number = loop_number + 1;
    % disp(end_loop)
end

fprintf('\n')

%% Output
full_out = T_keep1;
if rad_switches(1) == 1
    assignin(workspace_name,'Qdot_out_keep',Qdot_out_keep);
end
if rad_switches(2) == 1
    assignin(workspace_name,'Qdot_solar_nodes_keep',Qdot_solar_nodes_keep);
    assignin(workspace_name,'Qdot_solar_elems_keep',Qdot_solar_elems_keep);
    assignin(workspace_name,'Qdot_albedo_nodes_keep',Qdot_albedo_nodes_keep);
    assignin(workspace_name,'Qdot_albedo_elems_keep',Qdot_albedo_elems_keep);
    assignin(workspace_name,'Qdot_IR_nodes_keep',Qdot_IR_nodes_keep);
    assignin(workspace_name,'Qdot_IR_elems_keep',Qdot_IR_elems_keep);
    if env_rad_type == 1
        epoch_keep(end) = [];
        assignin(workspace_name,'S_keep',S_keep);
        assignin(workspace_name,'vS_keep',vS_keep);
        assignin(workspace_name,'vBody_keep',vBody_keep);
    elseif env_rad_type == 0
        epoch_keep(end) = [];
        assignin(workspace_name,'S_keep',S_keep);
        assignin(workspace_name,'vS_keep',vS_keep);
        assignin(workspace_name,'vBody_keep',vBody_keep);
    end
end
if rad_switches(3) == 1
    assignin(workspace_name,'Qdot_e2e_keep',Qdot_e2e_keep);
end
end
```

surface_conductance_R02.m

```
function CONDUCTANCES_1 =
surface_conductance_R02(surface_number,CONDUCTANCES_1,SURFACES_1,SURFACES_2,THERMOP
HYSICAL_1,SURFACELENGTHS_1)
% Creates a matrix listing all the conductances between each node in a
% surface.
% Version 1.0 completed 10/17/2023
```

```
material = SURFACES_1{surface_number}.material;
indices = find(contains(THERMOPHYSICAL_1,material)); % Find indices of material in
global storage matrix
k = str2double(THERMOPHYSICAL_1(indices,3)); % Ignore matlab's suggestion here
relevant_nodes = SURFACES_2{surface_number}.nodes;

% Internodal Area
A = internodal_area_R03(surface_number,SURFACES_1,SURFACES_2);
%fprintf('(Within surface_conductance) Internodal area calculated.\n')

% Internodal Lengths
L = SURFACELENGTHS_1(relevant_nodes,relevant_nodes);

% Calculate local Conductances
G_local = k*A./L;

% Assign local G to global CONDUCTANCES_1 matrix
CONDUCTANCES_1(relevant_nodes,relevant_nodes) = G_local;

end
```

surface_mesh_create_R01.m

```
function [MESHGRIDS_1,SURFACES_2] =
surface_mesh_create_R01(surface_number,origin,relative,vec_x1,vec_x2,n_x1,n_x2,star
t_node,MESHGRIDS_1,SURFACES_2)
% Generates a surface of nodes. The output matrix is of the following
% format:
% [Node number, Thermal mass, Current temperature, Parent surface, x1, x2, x3]
% Thermal mass and current temperature are defined separately.
% "Relative" is a switch that identifies whether the values given for
% vec_x1 and vec_x2 are nodal positions (not actually vectors) in the global
% CSYS or are relative vectors from the origin point. (0 is global, 1 is relative)
% Version 1.3 completed 9/29/2023

% Check if nodes exist and set warning
if ~isempty(SURFACES_2) % If SURFACES_2 has data inside...
    if surface_number <= length(SURFACES_2) % If the input surface ID is not
greater than the length of SURFACES_2...
        if ~isempty(SURFACES_2{surface_number}) % Check if the cell at the index
provided by the surface ID is already occupied.
            fprintf('Warning: Surface %i already exists. Utilize the EDIT function
to edit the surface. (NOTE to self: This is not yet implemented as of
9/29/2023.)\n',surface_number);
            return
        end
    end
end

if ~isempty(MESHGRIDS_1)
    temp = find(MESHGRIDS_1(:,1)==start_node, 1);
    if ~isempty(temp)
        % Error handling
```

```
        fprintf('ERROR (surface_mesh_create): Nodes overlap starting at %i. Please
enter a different start node.\n',temp);
        %start_node = input("Please enter a different start node.");
        return
    end
end

if n_x1 < 2 || n_x2 < 2
    % Check if node count is reasonable
    fprintf('ERROR (surface_mesh_create): Too few nodes to create a surface! Node
count along each edge must be 2 or greater.\n');
    return
else
    if relative ~= 1 % Convert nodal positions to relative vectors if global
positions are given
        vec_x1 = vec_x1-origin;
        vec_x2 = vec_x2-origin;
    end
    vec_x3 = cross(vec_x1,vec_x2);
    l_x1 = norm(vec_x1); % Length in x1 direction [m]
    l_x2 = norm(vec_x2); % Length in x2 direction [m]
    l_x3 = norm(vec_x3); % Create magnitude of vec_x3 for normalization
    unit_x1 = vec_x1/l_x1; % Unit vector in x1 direction
    unit_x2 = vec_x2/l_x2; % Unit vector in x2 direction
    unit_x3 = vec_x3/l_x3; % Unit vector in x3 direction
    inner_angle = acosd(dot(vec_x1,vec_x2)/(norm(vec_x1)*norm(vec_x2))); % Inner
angle of vectors
    if inner_angle ~= 90
        fprintf('ERROR (surface_mesh_create): Not a rectangular plate! Angle
between given vectors must be 90 degrees.\n')
        return
    end

    % Defining nodes
    surface_node_total = n_x1*n_x2; % Total number of nodes in surface 01
    ones_vec = ones(surface_node_total,1); % Ones vector sized for node count

    % Local coordinates
    coords_x1 = linspace(0,l_x1,n_x1)'; % x1 coordinates of nodes [m]
    coords_x2 = linspace(0,l_x2,n_x2)'; % x2 coordinates of nodes [m]

    % Create nodes
    nodes = (start_node:start_node+surface_node_total-1)'; % Create list of nodes
    %fprintf('surface_mesh_create - Surface %i node count:
%i\n',surface_number,surface_node_total);

    % Coords matrix
    current_local_node = 1; % Use for iterating through list of nodes
    coords_mat = ones(surface_node_total,3);
    for i=1:n_x2
        for j=1:n_x1
            coords_mat(current_local_node,:) = [coords_x1(j),coords_x2(i),0];
            current_local_node = current_local_node+1;
        end
    end
```

```
    % Convert local coords to global coords
    gcoords_mat = ones(3,surface_node_total); % Initialization
    for it = 1:surface_node_total
        gcoords_mat(:,it) =
local2globalcoord(coords_mat(it,:)','rr',origin,[unit_x1,unit_x2,unit_x3]);
    end
    gcoords_mat = gcoords_mat';

    % Grid Matrix: [Node number, Thermal mass, Current temperature, x1, x2, x3]
    surface_grids = [nodes, NaN*ones_vec, NaN*ones_vec,
gcoords_mat(:,1),gcoords_mat(:,2),gcoords_mat(:,3)];

    % Assign to MESHGRIDS_1
    for j = 1:height(surface_grids)
        MESHGRIDS_1(surface_grids(j,1),:) = surface_grids(j,:);
    end


    %MESHGRIDS_1 = [MESHGRIDS_1;surface_grids]; % Add surface grids to list of all
grids
    %MESHGRIDS_1 = sortrows(MESHGRIDS_1); % Sort grids by node number

    surface.ID = surface_number; % Store surface ID
    surface.nodes = nodes; % Store nodes related
    surface.n_x1 = n_x1;
    surface.n_x2 = n_x2;
    surface.vec_x1 = vec_x1;
    surface.vec_x2 = vec_x2;
    surface.vec_x3 = unit_x3;
    surface.area = norm(cross(vec_x1,vec_x2));
    SURFACES_2{surface_number} = surface; % Store surface

end
end
```

surface_mesh_edit.m

```
function
[MESHGRIDS_1,ELEMENTS_1,SURFACES_1,SURFACES_2,CONDUCTANCES_1,CONDUCTANCES_2] =
surface_mesh_edit_R01(ID,action,overwrite,input,MESHGRIDS_1,ELEMENTS_1,SURFACES_1,S
URFACES_2,CONDUCTANCES_1,CONDUCTANCES_2,THERMOPHYSICAL_1,THERMOOPTICAL_1)
% This function edits surface properties and can also be used to change the
% node IDs. This function does NOT allow for movement of the node. That can
% be performed with node_move.
%
% Several different actions can be performed:
% action = 0: Delete the surface.
% action = 1: Change surface ID.
% action = 2: Change starting node ID (this will renumber all following
% nodes sequentially from this node). If any of the nodes already exists
% the function will abort regardless of the overwrite state.
% action = 3: Change the thermophysical property.
```

```
% action = 4: Change the topside thermooptical property.
% action = 5: Change the bottomside thermooptical property.
%
% If there is a nonzero value already at the position, overwrite must be
% set to 1 to confirm overwrite.
%
% Version 1.0 completed 12/14/2024


%% Check if surface exists
if ID > length(SURFACES_1)
    fprintf('ERROR (surface_mesh_edit): Surface does not exist.\n')
    return
end
if isempty(SURFACES_1{ID})
    fprintf('ERROR (surface_mesh_edit): Surface does not exist.\n')
    return
end

%% Get nodes
nodes = SURFACES_2{ID}.nodes;
nodecount = length(nodes);

%% Make edits
if action == 0 % Delete
    % Check overwrite
    if overwrite ~= 1
        fprintf('ERROR (surface_mesh_edit): Overwrite must be set to 1 to confirm
deletion of surface.\n')
        return
    end

    % MESHGRIDS_1
    MESHGRIDS_1(nodes,:) = 0;

    % ELEMENTS_1
    [ind1,~] = find(ELEMENTS_1(:,1)==ID);
    ELEMENTS_1(ind1,:) = [];

    % SURFACES_1
    SURFACES_1{ID} = {};

    % SURFACES_2
    SURFACES_2{ID} = {};

    % CONDUCTANCES_1
    CONDUCTANCES_1(nodes,:) = 0;
    CONDUCTANCES_1(:,nodes) = 0;

    % CONDUCTANCES_2
    if ~isempty(CONDUCTANCES_2)
        for ii = 1:length(CONDUCTANCES_2)
            if ~isempty(CONDUCTANCES_2{ii})
                mat = CONDUCTANCES_2{ii};
```

```
                for jj = 1:length(nodes) % Iterate through list of nodes in the
surface
                    [ind1,~] = find(mat(:,1:2)==nodes(jj)); % Find the rows the
node exists in
                    mat(ind1,:) = []; % Delete the rows
                end
                CONDUCTANCES_2{ii} = mat;
            end
        end
    end

elseif action == 1 % Surface ID
    % Check if new surface ID exists
    if input <= length(SURFACES_1)
        if ~isempty(SURFACES_1{input}) && overwrite ~= 1
            fprintf('ERROR (surface_mesh_edit): New surface ID already exists. Set
overwrite=1 to overwrite the existing node.\n')
            return
        end
    end

    % Update surface number in ELEMENTS_1
    [ind1,ind2] = find(ELEMENTS_1(:,1)==ID);
    ELEMENTS_1(ind1,ind2) = input;

    % Update SURFACES_1
    SURFACES_1{input} = SURFACES_1{ID};
    SURFACES_1{ID} = {};

    % Update SURFACES_2
    SURFACES_2{input} = SURFACES_2{ID};
    SURFACES_2{ID} = {};

elseif action == 2 % Start node ID
    % Get list of new nodes
    new_nodes = (input:input+nodecount-1)';

    % Check if any of new nodes already exists. Throw error and list
    % violating nodes if so.
    check = MESHGRIDS_1(new_nodes,1) ~= 0;
    if sum(check) > 0
        existing_nodes = new_nodes(check);
        fprintf('ERROR (surface_mesh_edit): The following nodes already exist.
%s\n',mat2str(existing_nodes));
        return
    end

    % Assign nodes. Remove old nodes.
    MESHGRIDS_1(new_nodes,:) = MESHGRIDS_1(nodes,:);
    MESHGRIDS_1(nodes,:) = 0;
    SURFACES_2{ID}.nodes = new_nodes;

elseif action == 3 % Thermophysical property
    % Check if input is a string
    if ~isstring(input)
```

```
        fprintf('ERROR (surface_mesh_edit): Input must be a string.\n')
        return
    end

    % Check if input property exists
    indices = find(contains(THERMOPHYSICAL_1,input), 1); % Find indices in global
storage matrix
    if isempty(indices)
        fprintf('ERROR (surface_mesh_edit): Input property does not exist.\n')
        return
    end

    % Assign
    SURFACES_1{ID}.material = input;

elseif action == 4 % Topside thermooptical property
    % Check if input is a string
    if ~isstring(input)
        fprintf('ERROR (surface_mesh_edit): Input must be a string.\n')
        return
    end

    % Check if input property exists
    indices = find(contains(THERMOOPTICAL_1,input), 1); % Find indices in global
storage matrix
    if isempty(indices)
        fprintf('ERROR (surface_mesh_edit): Input property does not exist.\n')
        return
    end

    % Assign
    SURFACES_1{ID}.optical_top = input;

elseif action == 5 % Bottomside thermooptical property
    % Check if input is a string
    if ~isstring(input)
        fprintf('ERROR (surface_mesh_edit): Input must be a string.\n')
        return
    end

    % Check if input property exists
    indices = find(contains(THERMOOPTICAL_1,input), 1); % Find indices in global
storage matrix
    if isempty(indices)
        fprintf('ERROR (surface_mesh_edit): Input property does not exist.\n')
        return
    end

    % Assign
    SURFACES_1{ID}.optical_bot = input;

else
    fprintf('ERROR (node_edit): Invalid action. Use 1 to change the node ID, 2 to
change the thermal mass, 3 to change the initial temperature.\n')
    return
```

```
end

end
```

surface_move_R01.m

```
function MESHGRIDS_1 =
surface_move_R01(ID,relative,x1,x2,x3,r1,r2,r3,MESHGRIDS_1,SURFACES_2)
% Moves and rotates a surface given the inputs.
% ID - Surface ID
% relative - Switch between relative movement from current origin (1) and absolute
position of origin (0)
% x1 - movement in x1 [m]
% x2 - movement in x2 [m]
% x3 - movement in x3 [m]
% r1 - rotation about x1 [deg]
% r2 - rotation about x2 [deg]
% r3 - rotation about x3 [deg]
% Note that the x1, x2, and x3 axes are centered at the origin.

% Version 1.0 completed 12/12/2023

%% Check if surface exists
if ID > length(SURFACES_2)
    fprintf('ERROR (surface_move): Surface does not exist.\n')
    return
end
if isempty(SURFACES_2{ID})
    fprintf('ERROR (surface_move): Surface does not exist.\n')
    return
end

%% Translate
surf_nodes = SURFACES_2{ID}.nodes;
origin = surf_nodes(1);
origin_coords = MESHGRIDS_1(origin,4:6);
if relative == 0
    translation_vec = [x1,x2,x3] - origin_coords; % Find translation needed to get
to the new origin point
elseif relative == 1
    translation_vec = [x1,x2,x3];
else
    fprintf('ERROR (surface_move): Invalid input for relative. Choose 1 for
movement relative to initial position or 0 for absolute position.\n')
    return
end
for ii = 1:length(surf_nodes) % Iterate through each node in the surface and edit
the coordinates
    current_node = surf_nodes(ii);
    MESHGRIDS_1(current_node,4:6) = MESHGRIDS_1(current_node,4:6) +
translation_vec;
end
new_origin_coords = MESHGRIDS_1(origin,4:6);
```

```
%% Rotate
rotm = eul2rotm([deg2rad(r3),deg2rad(r2),deg2rad(r1)]); % Convert to rotation
matrix (order is ZYX).
% Find vector between each point and the surface origin and apply rotation matrix
for ii = 2:length(surf_nodes)
    current_node = surf_nodes(ii);
    node_coords = MESHGRIDS_1(current_node,4:6);
    node_vec = node_coords - new_origin_coords;
    rotated_vec = rotm*node_vec';
    new_coords = new_origin_coords + rotated_vec';
    MESHGRIDS_1(current_node,4:6) = new_coords;
end

end
```

surface_post_visualization_R02.m

```
function surface_post_visualization_R02(MESHGRIDS,ELEMENTS)
% Plots the elements. Display normal vector with the back_switch.
% This version is specifically for finding nodes whose indices do not match
% their node number.

% Version 2 completed 10/17/2023

reshaped_indices = zeros(4,1);
for i = 1:height(ELEMENTS)
    relevant_nodes = ELEMENTS(i,2:5);
    for j = 1:length(relevant_nodes)
        reshaped_indices(j) = find(relevant_nodes(j)==MESHGRIDS(:,1));
    end
    x_coords = MESHGRIDS(reshaped_indices(:,1),4);
    y_coords = MESHGRIDS(reshaped_indices(:,1),5);
    z_coords = MESHGRIDS(reshaped_indices(:,1),6);
    nodal_temperatures = MESHGRIDS(reshaped_indices(:,1),3);
    fill3(x_coords,y_coords,z_coords,nodal_temperatures);
    hold on
end
pbaspect([1 1 1])
colormap(jet(256))
colorbar
%clim([0,20])
axis equal
hold off
pause(0.000001)
end
```

surface_post_visualization_R03.m

```
function
surface_post_visualization_R03(timestep_number,MESHGRIDS_11,solver_temperature_outp
ut,elem_nodes_map)
% Plots elemental temperatures at the given timestep.

% MESHGRIDS_11 holds nodal coordinates and node numbers which do not match
% actual node indices.
% elem_nodes_map holds the indices of each node that maps to MESHGRIDS_11
% in each element.
% solver_temperature_output holds the temperatures at each instance in time

% Version 3.0 competed 4/8/2024

for i = 1:height(elem_nodes_map)
    relevant_node_indices = elem_nodes_map(i,:);
    x_coords = MESHGRIDS_11(relevant_node_indices,4);
    y_coords = MESHGRIDS_11(relevant_node_indices,5);
    z_coords = MESHGRIDS_11(relevant_node_indices,6);
    nodal_temperatures =
solver_temperature_output(relevant_node_indices,timestep_number);
    fill3(x_coords,y_coords,z_coords,nodal_temperatures);
    hold on
end
pbaspect([1 1 1])
colormap(jet(256))
colorbar
%clim([0,20])
axis equal
xlabel('X'); ylabel('Y');zlabel('Z');
hold off
pause(0.000001)
end
```

surface_pre_visualization_R01.m

```
function
surface_pre_visualization_R01(surfaces2plot,back_switch,marker_scale,MESHGRIDS_1,SU
RFACES_2)
% Plots the given surface. Display front/back with the back_switch.
% Only plots the surface. Does not display the elements.

% surface2plot: Desired surface to plot. Use a value <1 to select all surfaces.
% back_switch: Turns on the backface_color (0 for off, 1 for on)
% marker_scale: Size of the points in the plot

% Version 1.0 completed 10/2/2023
% Version 1.1 completed 12/4/2023 - Added input for multiple surfaces.
% Version 1.2 completed 12/8/2023 - Added input for selecting all surfaces.

if surfaces2plot < 1
    surfaces2plot = find(~cellfun(@isempty,SURFACES_2));
end
```

```
for ii = 1:length(surfaces2plot)
    surface2plot = surfaces2plot(ii);
    relevant_nodes = SURFACES_2{surface2plot}.nodes; % Make list of relevant nodes
    %disp(length(relevant_nodes))
    len = length(relevant_nodes); % Find length of list
    vec_x1 = SURFACES_2{surface2plot}.vec_x1;
    vec_x2 = SURFACES_2{surface2plot}.vec_x2;

    % Plot the nodes
    x_coords = zeros(length(relevant_nodes),1);
    y_coords = zeros(length(relevant_nodes),1);
    z_coords = zeros(length(relevant_nodes),1);
    for jj = 1:length(relevant_nodes)
        x_coords(jj) = MESHGRIDS_1(relevant_nodes(jj),4);
        y_coords(jj) = MESHGRIDS_1(relevant_nodes(jj),5);
        z_coords(jj) = MESHGRIDS_1(relevant_nodes(jj),6);
    end
    %disp(length(x_coords))
    scatter3(x_coords,y_coords,z_coords,marker_scale,'.');
    hold on
    offset_x = (x_coords(2)-x_coords(1))/3;
    offset_y = (y_coords(2)-y_coords(1))/3;
    offset_z = (z_coords(2)-z_coords(1))/3;

textscatter3(x_coords+offset_x,y_coords+offset_y,z_coords+offset_z,string(relevant_
nodes));
    %uistack(node_text,"top")

    % Plot frontface
    node_a = relevant_nodes(1);
    node_b = relevant_nodes(SURFACES_2{surface2plot}.n_x1);
    node_c = relevant_nodes(end);
    node_d = relevant_nodes(len-SURFACES_2{surface2plot}.n_x1+1);
    x_coords =
[MESHGRIDS_1(node_a,4),MESHGRIDS_1(node_b,4),MESHGRIDS_1(node_c,4),MESHGRIDS_1(node
_d,4)];
    y_coords =
[MESHGRIDS_1(node_a,5),MESHGRIDS_1(node_b,5),MESHGRIDS_1(node_c,5),MESHGRIDS_1(node
_d,5)];
    z_coords =
[MESHGRIDS_1(node_a,6),MESHGRIDS_1(node_b,6),MESHGRIDS_1(node_c,6),MESHGRIDS_1(node
_d,6)];
    c = [1 1 1 1];
    frontface = fill3(x_coords,y_coords,z_coords,c);
    frontface.FaceAlpha = 0.6;
    pbaspect([1 1 1]) % Set aspect ratio of plot to be 1:1:1

    % Plot backface
    if back_switch == 1

        vec_x3 = cross(vec_x1,vec_x2);
        unit_x3 = vec_x3/norm(vec_x3);

        back_off = 1e3; % Scale factor backface offset
        back_x = x_coords - unit_x3(1)/back_off;
```

```
        back_y = y_coords - unit_x3(2)/back_off;
        back_z = z_coords - unit_x3(3)/back_off;

        c = [0.5 0.5 0.5 0.5];
        backface = fill3(back_x,back_y,back_z,c);
        backface.FaceAlpha = 0.5;
    end

end
pbaspect([1 1 1]) % Set aspect ratio of plot to be 1:1:1
colorbar
axis equal

end
```

surface_properties_assign_R02.m

```
function [MESHGRIDS_1,SURFACES_1] =
surface_properties_assign_R02(surface_number,thermophysical_name,thermooptical_name
_top,thermooptical_name_bottom,vec_x1,vec_x2,thickness,MESHGRIDS_1,SURFACES_1,SURFA
CES_2)
% Initiates the selected surface with defined properties

% INPUTS:
% surface_number - the number identifying the surface
% thermophysical_name - a string which is the identifier of the thermophysical
property already created
% thermooptical_name_top - a string which is the identifier of the thermooptical
property already created, identifying the thermooptical property to be applied to
the top side of the surface
% thermooptical_name_bottom - a string which is the identifier of the thermooptical
property already created, identifying the thermooptical property to be applied to
the bottom side of the surface
% vec_x1 - the vector defining the x1 vector of the surface
% vec_x2 - the vector defining the x2 vector of the surface
% n_x1 - the number of nodes along the x1 direction
% n_x2 - the number of nodes along the x2 direction
% thickness - the thickness of the surface
% mesh_grids - global holding matrix of all nodes
% SURFACES_1 - global holding matrix of all surfaces
% THERMOPHYSICAL_1 - global holding matrix of all thermophysical properties
% THERMOOPTICAL_1 - global holding matrix of all thermo-optical properties

% OUTPUTS:
% mesh_grids - nodes are updated with thermal mass
% SURFACES_1 - surface properties are added to global holding matrix

% Version 2.0 completed 9/25/2023

if surface_number < 1
    fprintf('ERRROR (surface_properties_assign): Surface number must be 1 or
greater.')
    return
```

```
end

surface.ID = surface_number; % Set number as the ID

% Check if surface exists
if isempty(SURFACES_2)
    fprintf('ERROR (surface_properties_assign): No surfaces currently exist.\n');
    return
else
    %temp = SURFACES_2{:}.ID==surface_number;
    if surface_number <= length(SURFACES_2)
        if isempty(SURFACES_2{surface_number})
            % Error handling
            fprintf('ERROR (surface_properties_assign): Surface does not
exist.\n');
            return
        else
            % Create surface_props vector
            surface.area = norm(cross(vec_x1,vec_x2)); % Area of surface [m2]
            surface.material = thermophysical_name;
            surface.optical_top = thermooptical_name_top;
            surface.optical_bot = thermooptical_name_bottom;
            surface.thickness = thickness;
            surface.gtop = 1;
            surface.gbot = 1;
            surface.gtop_role = 0;
            surface.gbot_role = 0;
            surface.gtop_sn = 0;
            surface.gbot_sn = 0;
            SURFACES_1{surface.ID} = surface;
        end
    else
        fprintf('ERROR (surface_properties_assign): Surface does not exist.\n');
        return
    end
end
end
```

thermooptical_create_edit_R01.m

```
function THERMOOPTICAL_1 =
thermooptical_create_edit_R01(thermooptical_name,alpha,epsilon,THERMOOPTICAL_1)
% Creates a material thermo-optical property specifying the solar absorptivity
% (alpha) and infrared emissivity (epsilon).
% Version 1.0 completed ~9/19/2023.

material_thermooptical = [thermooptical_name,alpha,epsilon]; % Package the
properties

indeces = find(contains(THERMOOPTICAL_1,thermooptical_name)); % Find indeces in
global storage matrix
if isempty(indeces)
```

```
    THERMOOPTICAL_1 = [THERMOOPTICAL_1;material_thermooptical]; % If indeces cannot
be found (i.e. the material is not already created), then add the material
else
    THERMOOPTICAL_1(indeces(1),:) = material_thermooptical; % If indeces are found,
replace the material properties
end
end
```

thermophysical_create_edit_R01.m

```
function THERMOPHYSICAL_1 =
thermophysical_create_edit_R01(thermophysical_name,rho,k,cp,THERMOPHYSICAL_1)
% Creates a material thermophysical assignment specifying the density (rho
[kg/m3]),
% conductivity (k [W/m/K]), and specific heat of a material (cp [J/kg/K]).
% Version 1.0 completed ~9/19/2023.

material_thermophysical = [thermophysical_name,rho,k,cp]; % Package the properties

indeces = find(contains(THERMOPHYSICAL_1,thermophysical_name)); % Find indeces in
global storage matrix
if isempty(indeces)
    THERMOPHYSICAL_1 = [THERMOPHYSICAL_1;material_thermophysical]; % If indeces
cannot be found (i.e. the material is not already created), then add the material
else
    THERMOPHYSICAL_1(indeces(1),:) = material_thermophysical; % If indeces are
found, replace the material properties
end
end
```

vector_angles_R03.m

```
function out = vector_angles_R03(v1,v2)
% Takes in Nx3 matrices giving vector elements for v1 and v2. Each row of
% v1 and v2 is a new vector.

cv1v2 = cross(v1,v2,2);
s = vecnorm(cv1v2')';
c = dot(v1,v2,2);
out = atan2(s,c);
end
```

APPENDIX B: FREEFLYER CODE

This appendix presents the FreeFlyer code used for the test case in Chapter 5. This can be taken as an example of how to set up the socket connection between MATLAB and FreeFlyer. This code can also be accessed at https://github.com/qilim2/ThermalSolver.

checkout_interface_06.MissionPlan

```
// Variable Setup
String terminationCode;
terminationCode = FF_Preferences.UserInfo[2];

// Port number used by MATLAB, passed in through command-line call "-ui"
(user info)
Array portNum[2];
portNum[0] = StringToFloat(FF_Preferences.UserInfo[0]);
portNum[1] = StringToFloat(FF_Preferences.UserInfo[1]);

// Global loop-indexing variable
Variable i;

// Set request variable
Variable orbit_data_request;


// Create and Open Sockets

// Create client input socket object
Socket inSocket;

// IP address left blank if running on local machine
inSocket.IPAddress       = ""; // Requires specification if executed through
a server
inSocket.PortNumber      = portNum[0];
inSocket.SocketIsAscii   = 0; // Data input will be binary
inSocket.SocketTimeout   = 5;
inSocket.SocketType      = "client";
inSocket.RecvTranslation = "swap"; // Needed for binary communication
inSocket.SendTranslation = "swap"; // Needed for binary communication

// Create client output socket object
Socket outSocket;

// IPaddress left blank if running on local machine
outSocket.IPAddress       = ""; // Requires specification if executed
through a server.
outSocket.PortNumber      = portNum[1];
outSocket.SocketIsAscii   = 1; // Data output will be ASCII
outSocket.SocketTimeout   = -1; // Infinite
outSocket.SocketType      = "client";

// Open the input socket
```

```
Open inSocket;

// Open the output socket
Open outSocket;
Vector SunVec2;
Vector SunVec; // Vector from spacecraft to sun
Vector BodyVec; // Vector from spacecraft to orbital body
Vector SunVec_SCCSYS; // Vector from spacecraft to sun converted to
spacecraft coordinate system
Vector BodyVec_SCCSYS; // Vector from spacecraft to orbital body converted
to spacecraft coordinate system
String current_Epoch;
String output_SunVec;
String output_BodyVec;
String output_Eclipse;
String output_ToTextFile;

Variable end_loop;
end_loop = 0;
Array output_All[9]; // Final output string
SunVec2.BuildVector(9,Spacecraft1,Sun);
SunVec2.Active = 1;
SunVec2.Color = ColorTools.Yellow;
SunVec2.DrawMethod = 0;
ViewWindow1.AddObject(SunVec2);
While (Spacecraft1.ElapsedTime < TIMESPAN(32400 seconds));
   // FF to Matlab Section
   // Receive request for data from MATLAB
   Receive orbit_data_request from inSocket;
   If (orbit_data_request == 1);

      // Epoch (1)
      output_All[0] = Spacecraft1.Epoch.ToSeconds;

      // Sun Vector (2:4)
      SunVec.BuildVector(9,Spacecraft1,Sun); // Get vector from spacecraft
to sun in MJ2000 CSYS
      SunVec_SCCSYS.Element =
AttitudeConvert(0,3,Spacecraft1,SunVec.Element); // Convert from MJ2000 CSYS
to s/c body CSYS
      output_All[1:3] = SunVec_SCCSYS.Element;

      // Body Vector (5:7)
      BodyVec.BuildVector(9,Spacecraft1,Earth); // Get vector from
spacecraft to body in MJ2000 CSYS
      BodyVec_SCCSYS.Element =
AttitudeConvert(0,3,Spacecraft1,BodyVec.Element); // Convert from MJ2000
CSYS to s/c body CSYS
      output_All[4:6] = BodyVec_SCCSYS.Element;

      // In Eclipse (8)
      output_All[7] = Spacecraft1.InShadow;

      // Update end_loop (9)
      output_All[8] = end_loop;

      // Send data
```

```
      Send output_All to outSocket;
      Send terminationCode to outSocket; // Unique and arbitrary number
sequence signifying the final transmSpacecraft1ion

      // Flip flags
      orbit_data_request = 0;
   End;

   // ----------
   //
   // // FF to Text File Section
   // // Epoch (1)
   // current_Epoch = Spacecraft1.Epoch.ToString;
   //
   // // Sun Vector (2:4)
   // SunVec.BuildVector(9,Spacecraft1,Sun); // Get vector from spacecraft
to sun in MJ2000 CSYS
   // SunVec_SCCSYS.Element =
AttitudeConvert(0,3,Spacecraft1,SunVec.Element); // Convert from MJ2000 CSYS
to s/c body CSYS
   // output_SunVec =
StringConcat(SunVec_SCCSYS.Element[0].ToString,",",SunVec_SCCSYS.Element[1].
ToString,",",SunVec_SCCSYS.Element[2].ToString);
   //
   // // Body Vector (5:7)
   // BodyVec.BuildVector(9,Spacecraft1,Earth); // Get vector from
spacecraft to body in MJ2000 CSYS
   // BodyVec_SCCSYS.Element =
AttitudeConvert(0,3,Spacecraft1,BodyVec.Element); // Convert from MJ2000
CSYS to s/c body CSYS
   // output_BodyVec =
StringConcat(BodyVec_SCCSYS.Element[0].ToString,",",BodyVec_SCCSYS.Element[1
].ToString,",",BodyVec_SCCSYS.Element[2].ToString);
   //
   // // In Eclipse (8)
   // output_Eclipse = Spacecraft1.InShadow.ToString;
   //
   // // Final Concat and Report
   // output_ToTextFile =
StringConcat("[",current_Epoch,",",output_SunVec,",",output_BodyVec,",",outp
ut_Eclipse,"]");
   // Report output_ToTextFile to "C:\Users\qilim2\Box\Thesis\FreeFlyer
Missions\output_test_06.txt";
   // // ----------
   Step Spacecraft1;
   Update ViewWindow1;
End;
end_loop = 1;
// Final Concat and Report
output_All = {0,0,0,0,0,0,0,0,end_loop};

// Send data
//Call SendData(ARR, output_All, outSocket);
Send output_All to outSocket;
Send terminationCode to outSocket; // Unique and arbitrary number sequence
signifying the final transmission
```

```
Close inSocket;
Close outSocket;
```