Spacecraft Position and Attitude Estimation with Deep Learning

a project presented to
The Faculty of the Department of Aerospace Engineering
San José State University

in partial fulfillment of the requirements for the degree
Master of Science in Aerospace Engineering

by

Jesse W. Franklin

December 2024

approved by

Professor Jeanine Hunter
Faculty Advisor

ABSTRACT

Spacecraft Position and Attitude Estimation with Deep Learning

Jesse W. Franklin

Deep learning models and methods are purposed for estimating the position and attitude of a spacecraft deployed for a general space mission. The spacecraft is assumed to be instrumented with an onboard camera as well as the necessary computational hardware to provide predictions and estimates that inform its position and attitude. A position estimation system is established that uses the convolutional neural network AlexNet to identify the planet and approximate its altitude given an image captured by the spacecraft image. The images that form the dataset to train the position estimation system are synthetic images obtained from the software OpenSpace. The images are standardized so the planet is captured within the exact field of view for every planet. The raw image data is captured with dimensions $1920\times1080\times3$ and preprocessed to dimensions $224\times224\times3$ for compatibility with AlexNet. The position estimation system is able to classify planets within the images with an accuracy of 98.34% and estimate the planetary altitude with an average error or root-mean-squared error corresponding to roughly 2800 kilometers. A novel approach methodology for the attitude estimation system is proposed, leveraging optical flow estimation algorithms and machine learning models to obtain an estimate of the spacecraft angular velocities from video data. A use case for artificial intelligence in the context of spacecraft guidance, navigation, and control is made and demonstrated through preliminary means.

Acknowledgments

I would like to sincerely thank all of my friends and family in supporting, helping, and accompanying me on this long journey. To my father, mother, and three sisters, I am endlessly indebted to all of you for each of your plentiful and immeasurable contributions to my life, well-being, and academics during my every stage of life thus far. I aim constantly to be a brother and son not only to be proud of, but to return the acts of kindness, love, and care you have shown me. To Andreana Aquino: my unwavering companion and partner, I thank you for the years of support, understanding, and kindness you have offered me during my journey. To my friends Joseph Bruno, Grace Feng, Devin Jordao, Javaneh Keikha, Stanley Krzesniak, Jason Nguyen, Hieu Trinh, and Derek Ye, I am grateful to have shared my journey with each and every one of you and thank you for your support and companionship. To those whom I have acknowledged above, I wholeheartedly believe that no expanse of space and time within our universe is as valuable as the cherished memories and times we have shared.

To my esteemed advisor: Professor Jeanine Hunter, I thank you for your wisdom, guidance, and inspiration in completing this project. I also acknowledge and thank you for your faith in my capabilities through every stage of this project. I could not have achieved what I have during my journey and project without your courses and the outstanding learning environment you have consistently fostered.

To my professors: Dr. Long Lu and Dr. Yawo Ezunkpe, I thank you for the invaluable knowledge you have imparted me and for facilitating fruitful discussions with me regarding my questions and concerns during this journey. I also acknowledge and thank you for inspiring me to master topics that interest me through your courses, answering my many curiosities on the subject matter with grace.

To everyone I met during my journey, I thank you for crossing paths with me and being a part of this irreplaceable experience.

Table of Contents

List of Symbols

Symbol	Definition	Units (SI)
a, a_{geo}	Semi-major axis, geosynchronous orbit radius	m
G	Universal Gravitational Constant	N$\cdot$ m^2/kg^2
M	Mass of planet	kg
R	Equatorial radius of planet	m
r_p	Radius of perigee	m
r_a	Radius of apogee	m
e	Eccentricity of orbit	—
p	Semi-latus rectum	m
h	Specific angular momentum of orbit	m^2/sec
Ω	Right ascension of the ascending node	rad
ω	Argument of perigee	rad
i	Inclination angle	rad
θ	True anomaly, spherical coordinate azimuthal angle	rad
ω_i	Angular velocity about i-th axis	rad/sec
$M_{net,i}$	Net moment in the i-th direction	N$\cdot$m
W, w	Weights	—
b	Biases	—
a	Activations	—
I	Image or pixel intensity	nits
I_{ij}	Mass moment or product of inertia about i and j axes	kg$\cdot$m^2
r	Spherical coordinate radius, orbit radius	—
$\hat{p}_i$	Perifocal frame unit vector in i direction	—
$\hat{ec}_i$	Earth-centered Inertial frame unit vector in i direction	—
ϕ	Spherical coordinate polar angle, spacecraft roll perturbation	rad
x	x-coordinate of spacecraft location in reference frame	m
y	y-coordinate of spacecraft location in reference frame	m
z	z-coordinate of spacecraft location in reference frame	m
ψ	Spacecraft yaw perturbation	rad
β	Spacecraft pitch perturbation	rad

List of Tables

List of Figures

Chapter 1 – Introduction

1.1 Motivation

Artificial intelligence is a powerful, rapidly developing tool applicable to many fields of study and disciplines. Through machine learning, algorithms or models can predict results through regression, object classification, and image recognition all on unobserved datasets that the model has not been exposed to prior. The results produced by reliable machine learning algorithms can then be extended to a variety of different applications including but not limited to control systems, design analysis, empirical replication and prediction, and simulation. Machine vision is a subdivision of artificial intelligence that trains the model on image data to perform a desired outcome. Through processing massive data sets containing many images, the model learns to respond to any image input. Machine vision can be used in problems where the decision-making process is informed by the surrounding physical or visual environment. As technology and space exploration both advance, deep space exploration becomes more feasible than ever before. However, deep space exploration is currently reliant on the Deep Space Network (DSN) at NASA JPL. The purpose of the network is to communicate with spacecraft significantly distant from Earth in order to track and monitor the spacecraft. DSN is currently managing communication with "over 40 space missions from nearly 30 countries" and is projected to reach 50% excess demand by the 2030s [1]. Additionally, recent large-scale missions such as Artemis I and the James Webb Space Telescope have together used approximately 10% of the total hours supplied by the DSN in 2022 [1]. As deep space exploration becomes more accessible and such missions become more prevalent in industry, deep space missions must avoid active reliance on the DSN. The success of deep space exploration missions is heavily dependent on communications with Earth, indicating that a larger problem exists if the communications subsystem is inoperable during flight. Without communications with Earth, the spacecraft can no longer communicate its position and velocity data to a ground station, nor can it receive command signals; this is known as the 'Lost in Space' problem. Machine vision allows for an effective solution to the 'Lost in Space' problem that also circumvents reliance on the DSN. Machine vision can be used to train a spacecraft flight computer to respond to its surrounding environment using data received through image sensors. Depending on the mission objective, the flight computer can be trained using synthetic images based on real timestamps and locations that are projected along the mission trajectory. With a diverse training dataset, the machine vision model can make GNC decisions based on the observed environment. The machine vision model can also be trained to predict its own position and attitude data from the observed environment. Since the spacecraft can provide itself with GNC command signals, there is no reliance on communicating with ground stations on Earth and there is no chance that the spacecraft will be lost in space and not recoverable.

1.2 Literature Review

Neural networks (NNs) are artificial intelligence constructs designed to mimic the neural structure of a human brain [2]. NNs consist of neurons or nodes that are interconnected with each other. Each neuron consists of an activation value that is calculated by tuning parameters called the weights and biases that exist for each neuron. For layered NNs, the starting layer of neurons is called the input layer. The input layer is the 'intake' of the neural network at which the input data has gone through no transformations. As the input data passes through the layers of the neural networks, it can be reshaped, transformed, and or feature-extracted depending on the NN structure at each layer. Once the data has passed through every layer of the neural network, it reaches the final layer: the output layer. The output layer consists of a shape that is given by the desired output for the problem. For example, if the problem that the NN is designed to solve is to determine which images are images of animals and which are not, the output layer would be binary, returning a value of 0 for images that do not contain animals and a value of 1 for images that do. Additionally, the shape of the input layer for this problem is determined by the resolution of the image and each pixel (and its corresponding red, green, and blue values if applicable). The accuracy of the output layer changes by tuning the weights and biases associated with each neuron within the network. The foundational basis of neural networks is further elaborated upon in Chapter 2.

Convolutional neural networks (CNNs) are a modernized variant of neural networks that are specifically purposed for image inputs. The first CNN, referred to as Neocognitron, was established in 1980 to recognize handwritten Japanese characters [3]. Neocognitron paved the way for the foundational concepts the CNNs are comprised of such as convolutions, feature extraction, and pooling layers [3]. CNNs receive the input image and reduce the image size by extracting groups of pixels in the image called local receptive fields in the convolutional layer. This effectively divides the image input into subgroups and allows the CNN to further decompose the local receptive fields by identifying notable features [2]. The features of each local receptive field are then consolidated in a pooling layer to obtain information about the whole image. As with NNs, CNNs consist of weights and biases for each local receptive field. After passing through each convolutional and pooling layer, a fully connected layer is used to assign weights and biases to each feature then passed to the output layer [4]. An example CNN architecture is illustrated in Figure 1.1 [4]. The structure and functions of each layer is further elaborated upon in Chapter 2. AlexNet is a CNN model that is purposed for classifying images into 1000 different classes [5]. The structure of the AlexNet model is further discussed in Chapter 3.

Optical flow is a quantity obtained through computer vision that describes the movement of pixels in a video between successive frames of the video. It is commonly used for video stabilization, tracking objects in a video, and motion detection. Optical flow operates is governed by a partial differential equation (known as the optical flow equation) that relates derivatives of the intensities of pixels with respect to space and time within a sequence of images [6]. Thus, for optical flow to be applied, it is essential to assume that objects within view maintain the same intensity and that neighboring pixels share similar motion [6, 7]. Finally, optical flow values are approximated through the use of different algorithms. These algorithms or methodologies are sometimes classified as sparse or dense optical flow

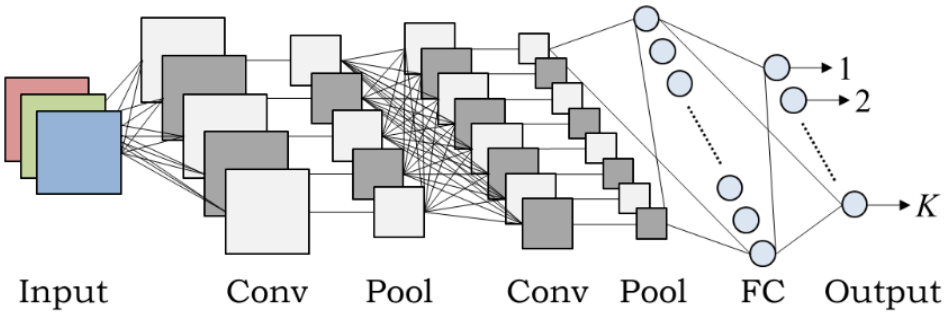

Figure 1.1: Convolutional neural network architecture example [4]

estimation. Sparse optical flow estimation involves tracking the motion of a few notable features while ignoring stationary pixels of objects in the image sequence. Dense optical flow involves finding the optical flow associated with every pixel in the image sequence, which can be more computationally expensive than sparse optical flow [8]. The function and use of optical flow is more thoroughly defined in Chapter 2. One of the most advanced, well-performing optical flow estimation algorithms is the recurrent all-pairs field transforms model (RAFT) established by Princeton University [7]. RAFT is briefly mentioned in Chapter 2 and its structure is further discussed in Chapter 3.

In recent years, considerable progress has been made in computer vision for aerospace and spacecraft applications. In particular, the advancement of deep learning allows for artificial intelligence structures such as CNNs to automate issues that may arise while utilizing classical computer vision. For instance, extracting features from an image to track their path or motion across a sequence of images is one of many applications in which deep learning has proven to be a useful and valuable concept. Deep learning has also been applied to existing computer vision methods such as optical flow to actively enhance the performance of the model iteratively. Additionally, machine learning or CNNs can also be applied to other focus areas of aerospace engineering such as fluid dynamics and control systems. As more advanced and improved deep learning models are established, the number of applications to aerospace engineering and science expand accordingly.

Feature extraction is an imaging processing task that identifies notable features in an image that can then be used for analysis in machine learning algorithms. Feature extraction has been used in tandem with an algorithm to determine the rotational state of small asteroids. The proposed algorithm involved over 800 test cases each with 250 synthetic images of two asteroids, Bennu and Itokawa, in a specified rotational state and lighting condition. In particular, the Speeded Up Robust Features (SURF) algorithm is used to identify features while the Kanade-Lucas-Tomasi (KLT) optical flow estimation algorithm is used to match features [9]. The features are then tracked and fitted to a matching ellipse that describes the rotation of the asteroid. Figure 1.2 [9] illustrates an example of the feature tracking and ellipse fitting processes employed to estimate the rotational state of the asteroid. By selecting several features from synthetic asteroid images, computer vision is used as a basis to determine the center of rotation and rotational axis of the asteroid to within 10 degrees for 80% of the considered test cases [9]. Thus, optical flow is a suitable technique for determining

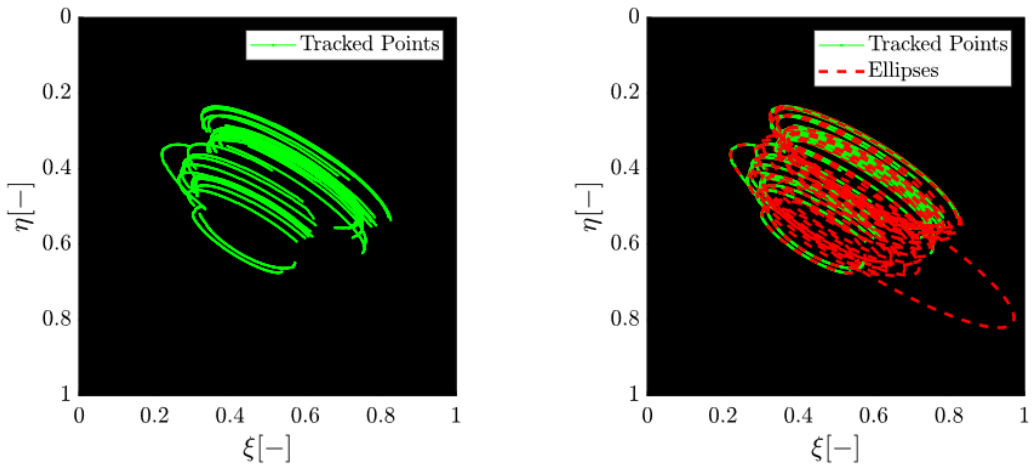

Figure 1.2: Example of tracked points and ellipse fitting [9]

the rotational state of objects.

Optical flow has also been applied to provide a position estimate of a UAV system based on ground movement. A discrete-time measurement optical flow measurement is modeled and transformed to directly augment position data rather than using velocity measurements to estimate position [10]. The algorithm is tested on a quadcopter UAV in outdoor environments and found to improve the accuracy of the position estimation of a magnetic, angular rate, and gravity (MARG) sensor [10].

CNNs extract features in an image automatically through convolution and subsequently develop the ability to differentiate among images. A siamese CNN has been developed to receive LiDAR or RGB-D data from synthetic images to identify and match features between images [11]. The purpose of this algorithm is to assist spacecraft in active debris removal by obtaining information about the motion of nearby objects through feature recognition. A shallow CNN architecture is used for feature extraction and matching because it "emphasizes features with a low level of abstraction such as corners and edges" and since "pooling layers lead to a loss of geometrical information" [11]. Figure 1.3 [11] illustrates the CNN architecture used to identify and match features within images of the satellite. Afterwards, the rotation is estimated through the tracked features using root-mean-square deviation and dense neural network approaches for comparison. The performance of the model on the dataset was found to be similar, if not improved, when compared to conventional methods. Overall, the proposed algorithm yields favorable results and also generalizes well to unseen geometries, suggesting that the algorithm is suitable for asteroid debris removal as well as other close-proximity purposes [11].

CNNs have also been used for supersonic CFD simulation in a problem regarding safe rocket stage separation. Shadowgraph images were used as a dataset for the neural network which then allowed the neural network to process data, providing valuable information such as jet mixing layer pulsations spectra, the spectra of acoustic waves, oscillations of the bow shock wave, and the bow shock position [12]. Additionally, the image processing techniques which were used allowed for the flow properties to be determined at any point. Computer vision tools were used to expedite the data processing for the data obtained from a phys-

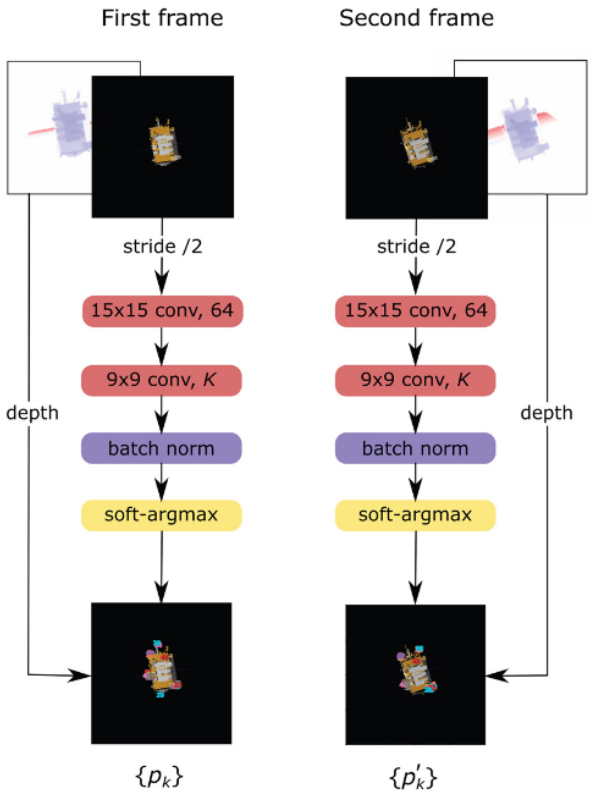

Figure 1.3: Landmark detection network architecture [11]

ical aerodynamic experiment and to obtain meaningful results for analyzing rocket stage separation in dense atmospheres [12].

A CNN-based architecture of a nonlinear control system has been developed for modelling the internal dynamics for fixed-wing aircraft [13]. The control system architecture, shown in Figure 1.4 adapts to the given aircraft, allowing for it to be trained to control different aircraft kinematic states. Furthermore, the inherent benefits of a CNN are used directly to automate the tuning process for the aircraft control system . The supervised learning method was performed offline and did not require aircraft flight data [13]. The results demonstrated that the control system performed well for 15 different aircraft, was able to control even inherently unstable systems, and adapted to both sudden and gradual changes in the states well [13].

Feature extraction and convolutional neural networks have been used to estimate motion and help expedite the processes involved in data processing and analysis. To train machine learning models and neural networks, data is essential as it provides a basis to train the model. Neural networks that use computer vision technology require training datasets in the form of images accompanied by the corresponding value or desired output for that image. By using a vast volume of image data along with computer vision technology, a vision-based model can then make decisions from the training dataset. When training a neural network or machine learning model purposed to be onboard a spacecraft, it is difficult to collect image data, especially when the mission trajectory has not been explored beforehand by

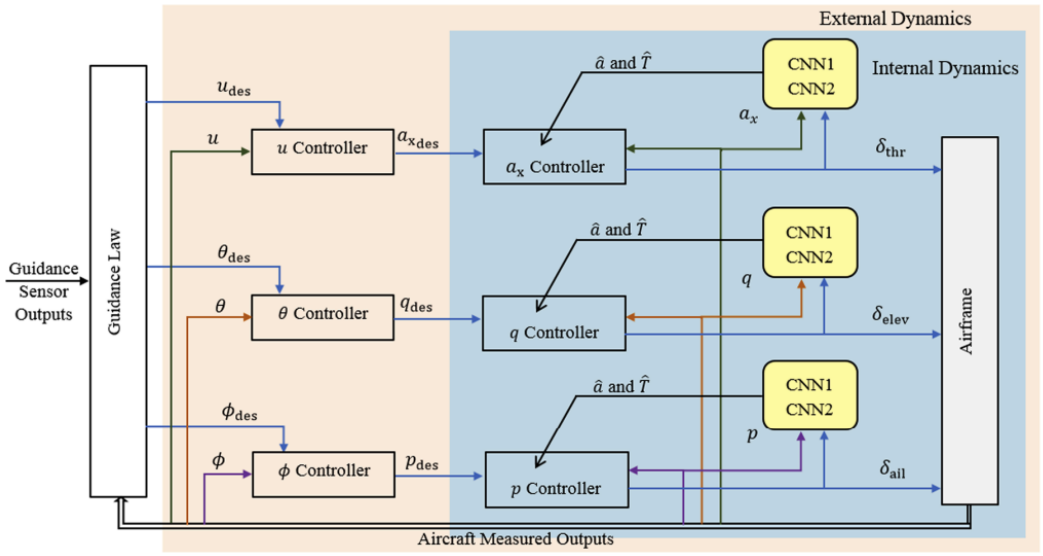

Figure 1.4: The overall structure of the control architecture for fixed-wing aircraft [13]

another spacecraft. Several neural networks and vision-based machine learning models have been trained using highly realistic synthetic images obtained through computer programs. In fact, synthetic images have been generated and used to train algorithms for identifying the rotational state of small asteroids and to estimate the pose (the position and rotational orientation) of uncooperative target spacecraft [9, 11, 14]. A convolutional neural network has been developed to estimate the pose of a non-cooperative target spacecraft and track the spacecraft [5]. The synthetic images were generated using simulated spacecraft orbits from the software Basilisk that were then used in Unreal Engine to generate realistic images that account for the effects from the Sun, Earth, and stars [14]. Synthetic images also provide a method of obtaining a large volume of data which can be further populated through methods such as data augmentation. Although synthetic images are particularly useful for applications in which the available data is scarce, they can potentially lead to problems with the model performance. For instance, a dataset containing only synthetic images that are highly realistic may not generalize well to real images where the depth of each pixel is not known [9]. Therefore, it is essential to introduce realistic factors such as noise and lighting effects to generate realistic synthetic images, especially if no real images that match the desired dataset are available.

To inform the spacecraft position and orientation, dynamics is essential as a basis of physical models. Analyses on the rotational dynamics of a spacecraft and determining the planetary altitude to inform its position require astrodynamics and rigid body dynamics as a basis of understanding. Specifically, the equatorial radius, mass, and rotational period of planets are of particular interest when obtaining synthetic images of each of the planets, which are provided in [15]. Furthermore, defining an initial state of the spacecraft requires knowledge of astrodynamics and rigid body dynamics with the added context of rigid body rotation under the effects of gravitational fields, which are all formulated in [16]. Thus, the necessary components of astrodynamics and rigid body dynamics applied to the 'lost in space' problem are fully stated in Chapter 2.

1.3 Project Proposal

This project's objective is to develop an algorithm for a spacecraft on a deep space mission. The algorithm uses the image data captured by a camera onboard the spacecraft to determine the position, velocity, and rotational motion of the spacecraft. The camera obtains pertinent information about the current environment and motion of the spacecraft such as nearby celestial objects and their relative motion through space and common features between consecutive images and their motion over time. The neural network and optical flow estimation model can use the data collected through images to then determine the spacecraft position and rotational velocities. Finally, the estimated position and attitude of the spacecraft can be fed to the GNC subsystem for the spacecraft to be able to continue its mission objective.

1.4 Methodology

A convolutional neural network will be used as the framework for determining the 6DOF attitude of the spacecraft based on the image data received from onboard cameras. The convolutional neural network will be trained on sequences of synthetic images obtained from computer software that visually simulates the environment for a proposed mission trajectory. The effects of lighting, noise, and surrounding celestial bodies will be considered when generating synthetic images. The synthetic images will be simulated at small time steps relative to one another to better inform the motion of the spacecraft. Data obtained through image processing can also be included in the training dataset, such as motion tracking of other bodies across each of the sequences of images. Astrodynamics can be used as a basis for verifying the propagated motion of celestial objects as well as informing the current position of the spacecraft in time. The image sequences will then be tied to the desired output of the convolutional neural network which include the position, attitude, and rotational motion of the spacecraft. Thus, the convolutional neural network develops an intuition for how to determine its position and attitude based on received sequences of images during its mission trajectory. Figures 1.5 and 1.6 below display an overview of the inputs and outputs for the position and attitude estimation systems for this project.

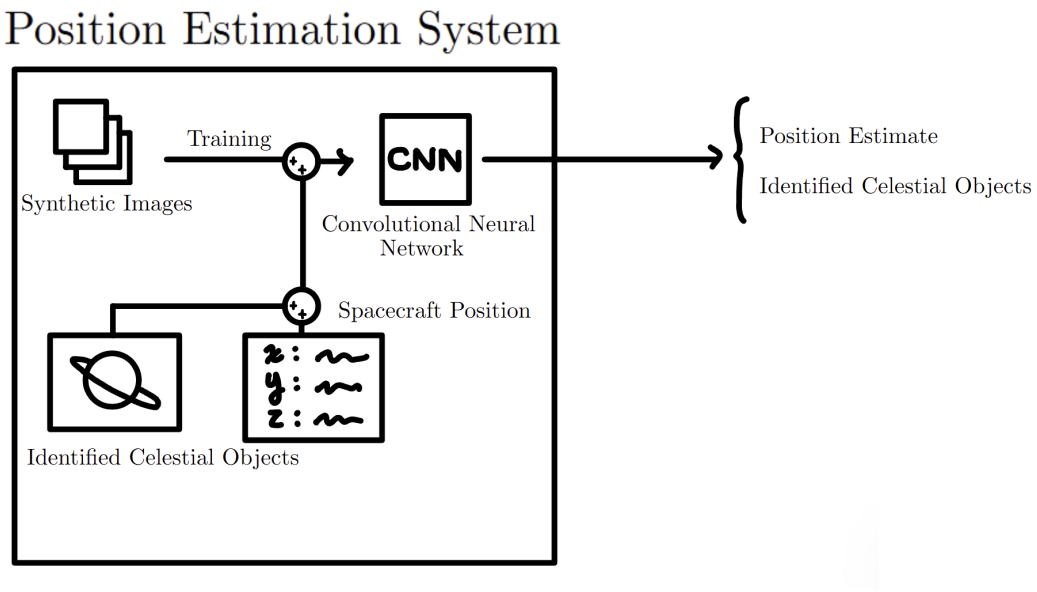

Figure 1.5: Overview of position estimation system

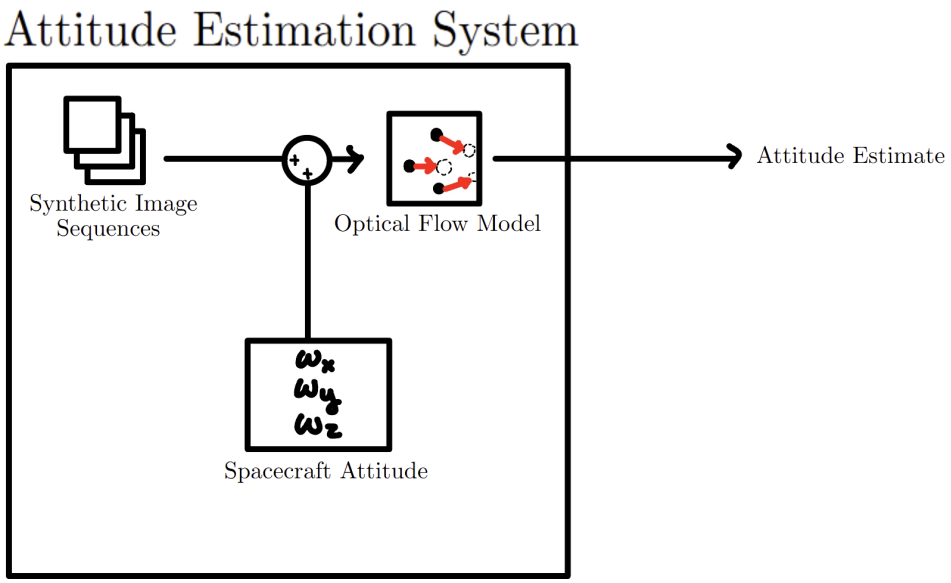

Figure 1.6: Overview of attitude estimation system

Chapter 2 – Background

2.1 Neural Networks

Neural networks are interconnected systems made up of perceptrons or neurons. In a layered neural network structure, neurons are grouped together by layers. The initial layer is referred to as the input layer and the last layer the output layer. Each neuron stores information in the form of an activation value ranging from 0 to 1. The activation value is essentially a confidence level or probability value that the neural network assigns for identifying a specific feature. For instance, the famous MNIST dataset example from the early stages of machine learning discussed by Nielsen [2] involves a neural network tasked with identifying handwritten digits from grayscale images. On the final layer or output layer of the neural network, the activation values are a measure of how certain the neural network is about the image being a specific digit. The activation of a neuron is calculated using all of the activation values from the neurons of the previous layer and tuned using weights and bias values. The weights determine the importance of each activation from the neurons of the previous layer and the bias is an additive constant. The formula for calculating the activation vector of a neuron is given in equation (2.1). Here, f is the function that normalizes the values

$$\vec{a}^{\,(1)} = f(W\vec{a}^{\,(0)} + \vec{b}) \tag{2.1}$$

obtained in the calculations, W is the matrix containing the weights of each neuron activation value, b is the bias vector, and a is the activation value vector. The superscript denotes the layer for the calculated activation values. The normalizing function differs depending on the context of the problem, but the rectified linear unit function (ReLU), which is defined as 0 for negative inputs and x for positive inputs, is commonly used.

The values for the weights and biases can be tuned using a gradient descent algorithm driven by the value of a cost function calculated from the output of the neural network. Nielsen defines the cost function in [2] as equation (2.2). Here, n is the total number of training inputs, w and b are individual values for the weights and biases, and $y(x)$ is the desired output value.

$$C(w,b) = \frac{1}{2n}\sum_{x} ||y(x) - a||^2 \tag{2.2}$$

2.2 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a more widely used and specified form of a neural network. In the context of image inputs, CNNs do not take in each individual pixel activation as an input. Instead, local receptive fields (or a square array of pixels in an image), are selected as individual inputs: a single local receptive field is illustrated in Figure 2.1 [2].

This allows for the local receptive fields to detect features as the CNN scans across the image to feed inputs into the next layer. Additionally, CNNs are inherently translation invariant due to their structure [2]. Every neuron in a local receptive field has a shared weight and bias value that have similar mathematical functions as they did in neural networks. The

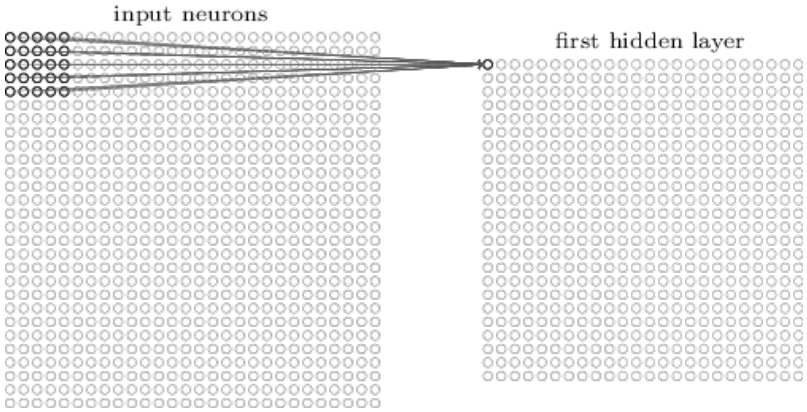

Figure 2.1: Local receptive fields [8]

calculation for the activation from the j, kth local receptive field is given in equation (2.3).

$$f\left(b + \sum_{l=0}^{N}\sum_{m=0}^{N} w_{l,m}a_{j+l,k+m}\right) \tag{2.3}$$

As before, f is the normalizing function, w are the weights, a are the activations, b are the biases, and N is the total dimension of local receptive fields that form a hidden layer (N x N local receptive fields). Each hidden layer whose activations are determined through the local receptive fields of the previous layer are referred to as a feature map. Thus, multiple feature maps are needed for detecting multiple features in an image. A group of several feature maps forms a convolutional layer in the CNN, named after the computations involved in determining the activations.

Another component of CNNs is the pooling layer. The primary purpose of the pooling layer is to reduce the dimension of the input layer while retaining the information about features within the region. Using the feature maps from the convolutional layer, pooling layers essentially construct a condensed feature map [2] containing all the information from each feature map in the convolutional layer.

The general structure of a CNN is depicted in Figure 1.1 [4]. Images with color are typically fed into the CNN as RGB data, measuring the red, green, and blue values for each pixel in the colored image. Within the CNN structure, the image is decomposed through the hidden layers which are shown in the diagram to be either convolutional layers or pooling layers. As discussed, convolutional layers use the mathematical operation of a convolution to generate several feature maps from the given input. These feature maps are then condensed dimensionally through the pooling layers which subsequently specifies the dimensions of the next convolutional layer. After data has been processed through the convolutional and pooling layers of the CNN, a fully connected (FC) layer is employed to establish values for the weights and biases for each feature which are tuned during backpropagation to improve the performance of the CNN. Finally, the output layer (for CNNs applied to classification problems) outputs the confidence level for each of the specified classes for the problem. For instance, in the MNIST dataset example, there are a total of 10 classes in the output layer for each digit from zero to nine.

2.3 Optical Flow

Optical flow is a computer vision technique that uses the changing intensities of pixels in a video or image sequence to track and quantify the relative motion of objects captured in the image sequence. This technique works under the assumptions that the pixels of an object do not change in intensity and that neighboring pixels share similar motion [6].

Consider the intensity of a pixel I in an image as a function of the pixel location, (x, y), and of time t as shown in Figure 2.2 [8]. As stated, the pixels of an object are assumed to

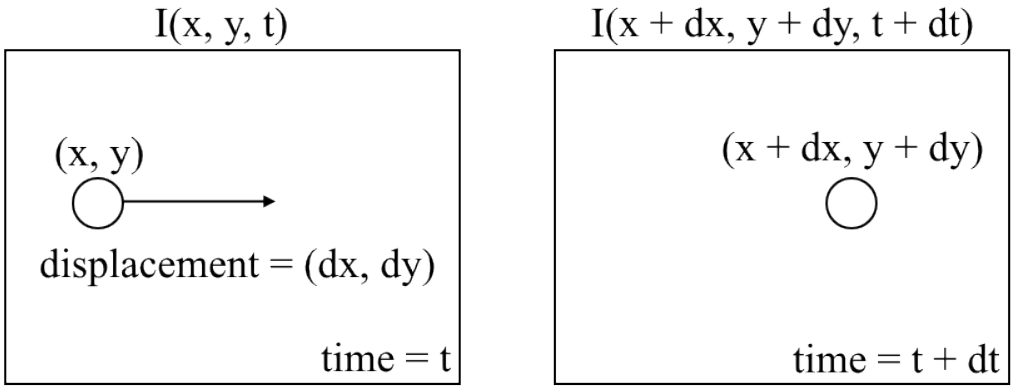

Figure 2.2: Definition of optical flow [8]

not change in intensity. Thus, it follows that

$$I(x,\, y,\, t) = I(x + dx,\, y + dy,\, t + dt). \tag{2.4}$$

Taking the Taylor series approximation of the right hand side of (2.4), the following equation is obtained.

$$I(x,\, y,\, t) = I(x + dx,\, y + dy,\, t + dt) = I(x,\, y,\, t) + \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt \tag{2.5}$$

$$\implies \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt = 0 \tag{2.6}$$

Dividing (2.6) by dt yields the optical flow equation (2.7), where u and v are the velocity vector components of a pixel dx/dt and dy/dt, respectively.

$$\frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t} = 0 \tag{2.7}$$

The optical flow equation is a partial differential equation with two unknowns. Thus, methods are required to determine u and v. These correspond to the velocity vector discussed earlier that is commonly referred to as a flow vector.

The two primary classifications of optical flow algorithms are dense and sparse optical flow. Dense optical flow seeks to find flow vectors for every pixel within an image while sparse optical flow only targets pixels that correspond to notable features within the image

[6]. Figure 2.3 [8] visualizes the output of sparse and dense optical flow estimation algorithms on overhead footage of a highway. The sparse optical flow algorithm yields the corners and lights of vehicles throughout the duration of the video (shown by the green lines) while the dense optical flow algorithm yields every pixel and object in motion .

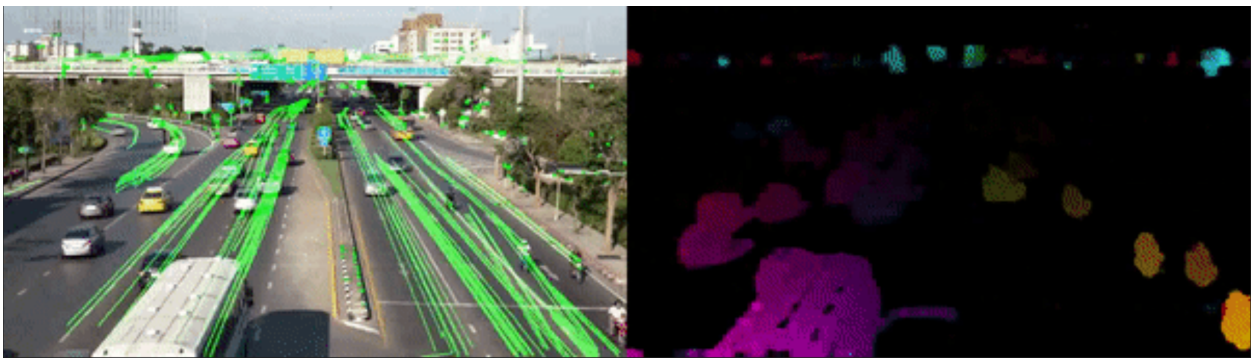

Figure 2.3: Comparison of sparse (left) and dense (right) optical flow [8]

The Lucas-Kanade method is a classical method for sparse optical flow that involves taking a 3x3 patch of pixels that are assumed to move together [8]. The basis of this method is formed from the assumption that neighboring pixels share similar motion. This indicates a system of nine equations in the form (2.7), but still two unknowns: an over determined system. The solution can then be obtained using a least-squares fitting through (2.8) [6].

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \sum_i \left(\frac{\partial I}{\partial x_i}\right)^2 & \sum_i \frac{\partial I}{\partial x_i}\frac{\partial I}{\partial y_i} \\ \sum_i \frac{\partial I}{\partial x_i}\frac{\partial I}{\partial y_i} & \left(\frac{\partial I}{\partial y_i}\right)^2 \end{bmatrix}^{-1} \begin{bmatrix} -\sum_i \frac{\partial I}{\partial x_i}\frac{\partial I}{\partial t_i} \\ -\sum_i \frac{\partial I}{\partial y_i}\frac{\partial I}{\partial t_i} \end{bmatrix} \tag{2.8}$$

The Recurrent All-pairs Field Transforms (RAFT) method is a modern optical flow estimation method from Princeton University that has achieved high performance in recent years [7]. From a high-level perspective, RAFT consists of a feature encoder, a correlation layer that computes visual similarity, and an iterative updater that recurrently updates optical flow from the correlation volumes [7]. As mentioned in discussing CNNs, feature extraction is inherent in deep learning structures.

2.4 Astrodynamics and Rigid Body Dynamics

One of the primary functions for the convolutional neural network design is to determine the position of the spacecraft. To do so, rigid body dynamics and astrodynamics are essential as a basis for both confirming the classification of any perceived celestial objects within the field of vision as well as providing a starting point to estimate the position.

2.4.1 Spacecraft Rigid Body Dynamics

In the absence of external forces and moments, outside the sphere of influence for any nearby bodies, the spacecraft equations of motion are Euler's equations of rotational motion: (2.9), (2.10), and (2.11).

$$M_{net,x} = I_{xx}\dot{\omega}_x + (I_{zz} - I_{yy})\omega_y\omega_z = 0 \tag{2.9}$$

$$M_{net,y} = I_{yy}\dot{\omega}_y + (I_{xx} - I_{zz})\omega_x\omega_z = 0 \tag{2.10}$$

$$M_{net,z} = I_{zz}\dot{\omega}_z + (I_{yy} - I_{xx})\omega_x\omega_y = 0 \tag{2.11}$$

These equations of motion assume that the moments of inertia are taken about the principal axes of the spacecraft and are constant with respect to time.

When the spacecraft is within the sphere of influence of celestial objects and the gravitational field does not impart a negligible effect, the same equations of motion can be used with the contribution of gravity to the external moments acting on the spacecraft. After simplifying and parameterizing the terms in the equations in the Euler angles of the spacecraft, the resulting equations (2.12), (2.13), and (2.14), are the equations of motion for a gravity gradient stabilized spacecraft [16].

$$I_{xx}\ddot{\phi} - (I_{xx} - I_{yy} + I_{zz})\dot{\theta}\dot{\psi} + 4\dot{\theta}^2(I_{yy} - I_{zz})\phi = 0 \tag{2.12}$$

$$I_{yy}\ddot{\beta} + 3\dot{\theta}^2(I_{xx} - I_{zz})\beta = 0 \tag{2.13}$$

$$I_{zz}\ddot{\psi} + (I_{xx} - I_{yy} + I_{zz})\dot{\theta}\dot{\phi} + \dot{\theta}^2(I_{yy} - I_{zz})\psi = 0 \tag{2.14}$$

2.4.2 Orbital Dynamics

To assist in classification and determining the position of the spacecraft, the orbital dynamics must be considered. Gibb's method can be used to both determine the projected orbit of celestial objects as well as determine the current orbit of the spacecraft about celestial objects. Gibb's method uses the position vectors of an object at three different instances in time to determine the orbit. The position vector and velocity vector of an object at a single instance in time can be used to determine the orbital parameters or propagate the motion of the orbit. The computations performed using Gibb's method are summarized in (2.15), (2.16), (2.17), and (2.18) [16].

$$h = \sqrt{GM\frac{|(\vec{r}_1 \times \vec{r}_2) + r_2(\vec{r}_2 \times \vec{r}_3) + r_3(\vec{r}_3 \times \vec{r}_1)|}{|(\vec{r}_1 \times \vec{r}_2) + (\vec{r}_2 \times \vec{r}_3) + (\vec{r}_3 \times \vec{r}_1)|}} \tag{2.15}$$

$$\hat{p}_z = \frac{(\vec{r}_1 \times \vec{r}_2) + (\vec{r}_2 \times \vec{r}_3) + (\vec{r}_3 \times \vec{r}_1)}{|(\vec{r}_1 \times \vec{r}_2) + (\vec{r}_2 \times \vec{r}_3) + (\vec{r}_3 \times \vec{r}_1)|} \tag{2.16}$$

$$e\hat{p}_y = \frac{\vec{r}_1(r_2 - r_3) + \vec{r}_2(r_3 - r_1) + \vec{r}_3(r_1 - r_2)}{|(\vec{r}_1 \times \vec{r}_2) + (\vec{r}_2 \times \vec{r}_3) + (\vec{r}_3 \times \vec{r}_1)|} \tag{2.17}$$

$${}^N\vec{v}^Q = \frac{GM}{h}\left(\frac{\hat{p}_z \times \vec{r}}{r} + e\hat{p}_y\right) \tag{2.18}$$

The orbit of an object can be fully defined and stated through the six Keplerian elements: the semi-major axis a, the eccentricity e, the right ascension of the ascending node Ω, the argument of perigee ω, the inclination angle i, and the true anomaly θ. These are given by (2.19)-(2.24).

$$a = \frac{1}{2}(r_p + r_a) \tag{2.19}$$

$$e = \frac{r_a - r_p}{r_a + r_p} \tag{2.20}$$

$$\Omega = \cos^{-1}\left(\frac{\hat{ec}_x \cdot \vec{n}}{|\hat{ec}_x||\vec{n}|}\right) \tag{2.21}$$

$$\omega = \cos^{-1}\left(\frac{\vec{e} \cdot \vec{n}}{|\vec{e}||\vec{n}|}\right) \tag{2.22}$$

$$i = \cos^{-1}\left(\frac{\hat{ec}_z \cdot \vec{h}}{|\hat{ec}_z||\vec{h}|}\right) \tag{2.23}$$

$$\theta = \cos^{-1}\left(\frac{\vec{r} \cdot \vec{e}}{|\vec{r}||\vec{e}|}\right) \tag{2.24}$$

The six Keplerian elements of an orbit are illustrated in Figure 2.4 [16].

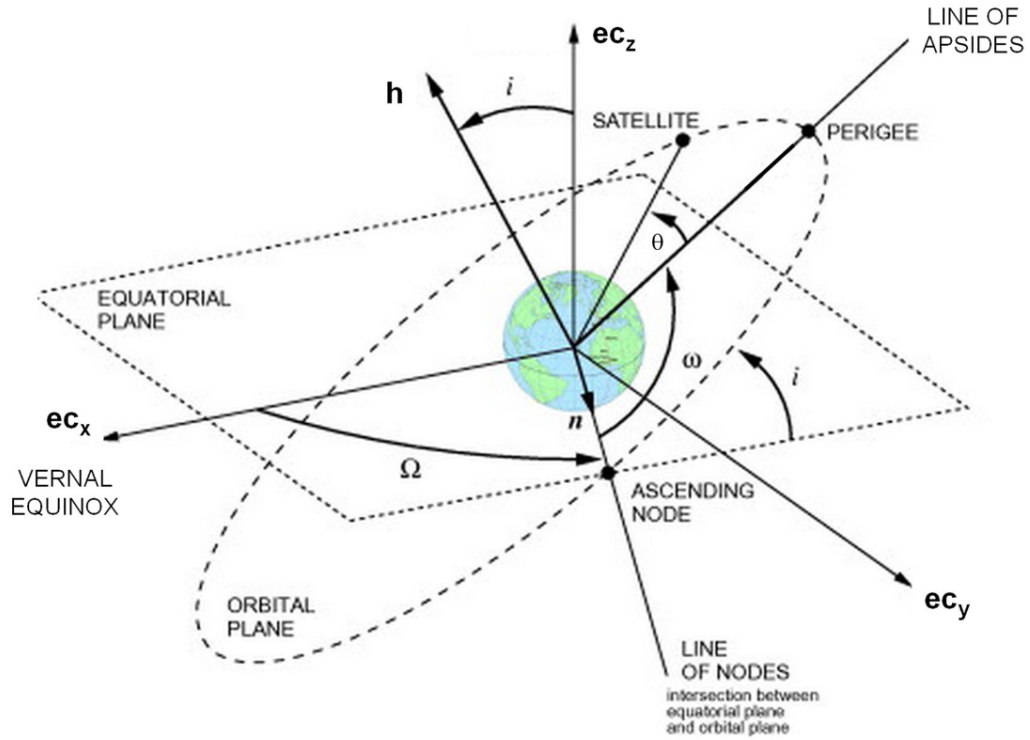

Figure 2.4: The six Keplerian elements [16]

The restricted two-body problem (R2BP) describes the motion of an object in orbit due to the gravitational attraction between the object and a body. This simplification yields a system of two coupled ordinary differential equations that can be solved numerically. The equations of motion for an orbit, (2.25) and (2.26), can be propagated to determine the projected motion of an object over time.

$$\ddot{r} - r\dot{\theta}^2 = -\frac{GM}{r^2} \tag{2.25}$$

$$r\ddot{\theta} + 2\dot{r}\dot{\theta} = 0 \tag{2.26}$$

Alternatively, if several Keplerian elements are known, the solution to the restricted two-body problem, (2.27), can be used to determine the position of a spacecraft in its orbit about a celestial body.

$$r(\theta) = \frac{p}{1 + e\cos\theta} = \frac{h^2/GM}{1 + e\cos\theta} \tag{2.27}$$

The period of an object or spacecraft about a celestial body is related through Kepler's third law of planetary motion.

$$T^2 = \frac{4\pi^2}{GM}a^3 \tag{2.28}$$

Chapter 3 – System Layout

3.1 System Components

3.1.1 Position Estimation System

The position estimation system is one of the two primary systems to achieve the mission objective. To estimate the position of the spacecraft during a deep space mission, the surrounding environment of the spacecraft must be observed to gather as much information as possible. Convolutional neural networks can then be used to identify objects in the field of vision of the spacecraft, classifying celestial objects by type and or name. If a celestial body is identified by name and the spacecraft is within the sphere of influence of said body, existing data on the mass and radius of the body can then be used to calculate the orbit of the spacecraft about the object. This offers two methods by which the position estimation algorithm can validate the position of the spacecraft: using the CNN to identify the celestial body and cross verifying the orbital movement of the spacecraft with the known mass of the celestial body using Gibb's method or orbital equations of motion. If the nearby celestial body is within a specified tolerance value, the position of the spacecraft relative to the body is known through the orbit equation.

The position estimation system described depends on a CNN that is trained to identify celestial objects and an algorithm that can identify and perform the necessary computations to determine the position of the spacecraft. By identifying the planet and the altitude from that planet, the heliocentric position of the spacecraft can also be estimated through vector addition if essential to the mission objective. However, this induces more error in position estimation as the orbits of the planets are reduced to circular, coplanar orbits in the estimation.

3.1.2 Attitude Estimation System

An attitude estimation system is the other primary system essential to the mission objective and provides useful information to the GNC subsystem. An optical flow algorithm can be trained and specified to identify notable features in the surrounding environment, tracking their movement through time. Assuming the surrounding objects are moving negligibly relative to the spacecraft, the rotational motion of the spacecraft can be determined through the frame-by-frame data. The appropriate equations of motion for a rotating object, both in the absence and presence of a gravitational field, can be used to determine the rotational velocities over time. The attitude estimation system described depends mainly on the optical flow algorithm measuring relative velocity through the motion of pixels to determine the attitude of the spacecraft. The attitude data can be fed directly to the GNC subsystem if rotation equilibrium or pointing is necessary for the mission objective.

3.2 Model Structures

The position and attitude estimation systems utilize machine learning models to estimate the position and rotational velocities of the spacecraft from image data collected through an

onboard camera. In particular, the position estimation system uses a CNN to identify the planet and the altitude of the spacecraft relative to that planet while the attitude estimation system uses an optical flow algorithm to determine the rotational velocity of the spacecraft. Thus, suitable CNN and optical flow algorithms are needed for these systems.

3.2.1 AlexNet

AlexNet is a deep convolutional neural network initially purposed to classify images in the ImageNet dataset into 1000 different classes for the ILSVRC (2010 and 2012) competitions [5]. AlexNet contains eight layers in total: five convolutional layers and three fully connected layers. The input is a colored image with dimension size 224 × 224 × 3. However, the source images are colored images with resolutions of 256 × 256 pixels. The input images are obtained through the data augmentation process in which five 224 × 224 patches (the corners and center patches, along with their horizontal reflections) of the 256 × 256 size image are obtained [5]; The third dimension of the input image corresponds to the RGB (red, green, and blue) color channels. Figure 3.1 [5] shows the architecture of the AlexNet CNN process split across two different GPUs.

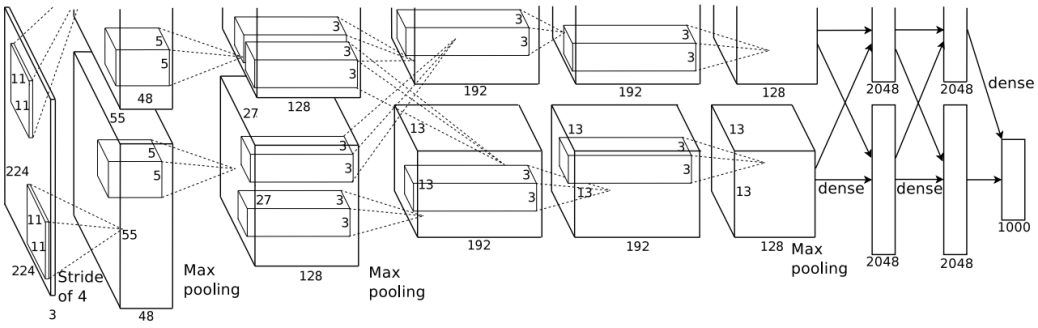

Figure 3.1: Overall architecture of the AlexNet CNN [5]

The structure of AlexNet is altered, specifically at the output layer, to identify the planet and estimate the altitude of the spacecraft from the planet. Instead of an output layer that specifies a normalized activation value for each of the 1000 classes that AlexNet is originally trained to identify, the output layer consists of the normalized activation value for the planets within the solar system and the estimated value for the altitude. This means that the output of the CNN yields 9 numerical values (8 normalized) in total.

3.2.2 RAFT

The structure of the RAFT algorithm is shown in Figure 3.2. The feature encoder extracts per-pixel features of two consecutive image frames of a video while a context encoder extracts features from only the first frame [7]. A 4-D correlation volume then computes the visual similarity by taking the inner product of all features in the first and second frames [7]. The process is then updated iteratively to improve the accuracy of the optical flow results. The authors of RAFT state that RAFT offers state-of-the-art accuracy, strong generalization, and high performance [7].

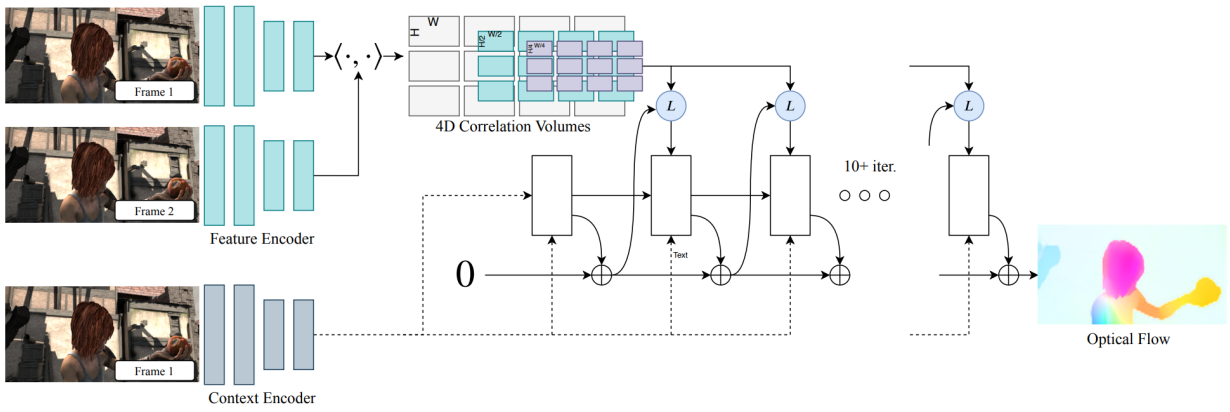

Figure 3.2: Overall architecture of the RAFT optical flow estimation algorithm [7]

Chapter 4 – Simulation

4.1 Data Collection

The estimation systems in this project involve machine learning models that require training data to estimate the desired outputs and testing data to evaluate the performance of the estimation systems. At a high level, the entire dataset must contain input data as well as the corresponding output data for the models to learn how to estimate the output accurately. The model processes the training data (including the associated output labels) to develop an intuition of the relationship between the inputs and outputs. After the model is trained, the performance of the model in estimating the desired output is evaluated through testing data. Testing data is stored with the corresponding output values, but the outputs are kept independent from the model to differentiate from training data and test the model in a proper testing environment.

To train the position and attitude estimation systems, a realistic, simulated space environment is essential for obtaining training images and videos. The open source software OpenSpace is used for this purpose. OpenSpace provides real time orbits of the planets within the solar system as well as the orbits of satellites in a 3-D environment. The software allows the user to navigate freely and provides the user with the altitude of the camera's perspective. The distant star field, stellar objects, and interstellar objects are also modeled in OpenSpace. The OpenSpace software allows for a Lua script to be run using the OpenSpace scripting API, directly affecting the simulation parameters and simulation state of the software. Additionally, existing mission data can also be imported into the software to generate simulated footage of spacecraft missions and orbits.

4.1.1 Data for Position Estimation System

To develop the position estimation system, training and testing data must consist of images of planets captured from the point of view of a spacecraft along with the planet captured in the image and the associated altitude at which the image was taken. The data is captured from software that both simulates and visualizes a realistic space environment for the solar system and provides altitude data. To preprocess the image data for training, the image data must be standardized to a set resolution.

To gather the necessary data, a Lua script that randomizes the time, orientation, and altitude is written and utilized. All parameters are randomized using the pseudorandom number generator Lua function 'math.random'. The randomized orientation is obtained by mapping the current reference frame of the camera to a spherical coordinate system (4.1). The definition of the spherical coordinate system is shown in Figure 4.1 obtained from [17].

$$(r, \theta, \phi) \rightarrow \begin{bmatrix} r\cos\theta\sin\phi \\ r\sin\theta\sin\phi \\ r\cos\phi \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \text{ where } r \in [0, \infty), \theta \in [0, 2\pi), \text{ and } \phi \in [0, \pi]. \tag{4.1}$$

The randomized orbital parameters are then inputted (with the correct units and reference frame) into a navigation state, defining the position of the camera. The camera is set to be

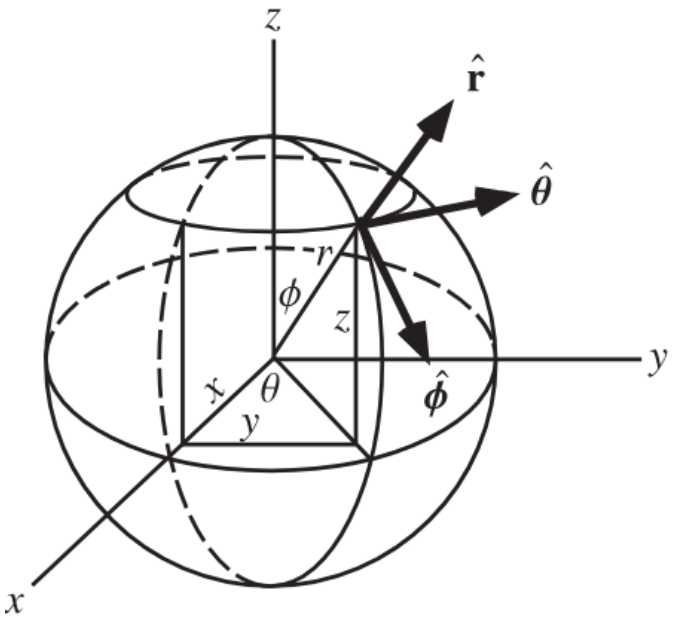

Figure 4.1: Spherical coordinate system geometry [17]

anchored about the planet of interest, meaning the planet is always located at the center of the field of view. After randomizing these orbital parameters, a time delay interrupts completion of the script to ensure rendering of the planet is fully complete before image capture. After the time delay, a screenshot of the planet view is taken and stored along with the corresponding altitude label and planet. The general form of the Lua scripts used to capture images of each planet is provided in Appendix C. The custom keybinding asset file used to execute the Lua script with a keyboard input is provided in Appendix D.

While the azimuthal angle θ and polar angle ϕ are within the ranges specified in (4.1), the radius is randomized within the range $r \in [c_1 a,\ c_2 a]$, where a is the geosynchronous orbit semi-major axis or radius for that planet, and c_1 and c_2 are constants depending on the planet. The constants c_1, c_2 are introduced as some planets have equatorial radii that are significant proportions of the geosynchronous orbit radius. The range of r values is selected relative to the geosynchronous orbit radius as such orbits are desirable for collecting information of the planet surface. The semi-major axis for a geosynchronous orbit is determined through Kepler's third law of planetary motion (2.28).

The resulting calculations are tabulated in Table 4.1 with essential values of each planet obtained from [15]. Namely the planetary mass, rotational period T (with negative value corresponding to rotations opposite of Earth's rotation sense), geosynchronous orbit radius a, equatorial radius R as well as the determined constants are tabulated. The values of c_1 for each planet were determined using the ratio between the equatorial radius of the planet R to the geosynchronous orbit radius a. Specifically, it was found that $c_1 = 1/3$ offered a satisfactory range of Earth orbit radii and that the ratio between Earth's radius to one-third of its geosynchronous orbit radius had a value of about 0.45. This value was then used to roughly determine the constant of c_1 for the other planets based on their respective R/a ratio. The constant c_2 was determined through similar means. The equations used to obtain

c_1 and c_2 given the radius to geosynchronous radius ratio are shown in (4.3) and (4.4). The necessary calculations are performed using a MATLAB script that is provided in Appendix B.

Table 4.1: Orbit Parameter Calculations

Planet	Mass [kg]	T [hrs]	a [10^5 km]	R [10^4 km]	R/a	c_1	c_2
Mercury	3.3010×10^{23}	1407.5	2.4283	0.24397	0.01	0.0222	0.0997
Venus	4.8673×10^{24}	-5832.4	15.362	0.60518	0.0039	0.0087	0.0391
Earth	5.9722×10^{24}	23.934	0.42155	0.63710	0.1511	0.3333	1.5000
Mars	6.4169×10^{23}	24.623	0.20423	0.33895	0.1660	0.3660	1.6472
Jupiter	1.8981×10^{27}	9.92496	1.5997	6.9911	0.4370	0.9639	4.3374
Saturn	5.6832×10^{26}	10.656	1.1221	5.8232	0.5189	1.1445	5.1504
Uranus	8.6810×10^{25}	-17.23992	0.82668	2.5362	0.3068	0.6766	3.0449
Neptune	1.0241×10^{26}	16.11000	0.83490	2.4622	0.2949	0.6504	2.9270

$$c_1 = 2.2055 \cdot \frac{R}{a} \tag{4.2}$$

$$c_2 = 9.925 \cdot \frac{R}{a} \tag{4.3}$$

Sample position estimation system dataset images (before any preprocessing) are shown for each of the planets in Figures 4.2 to 4.9. Though the resolution of the sample images differs slightly from the set standard resolution of 1920×1080 pixels, the raw data was obtained at this resolution. The sample images are taken at the bounds of the range of possible orbit radius values to demonstrate the appropriateness of the selected c_1 and c_2 values for each planet. The calculated minimum and maximum radius values for the orbits about each of the planets within the solar system are tabulated in Table 4.2.

Table 4.2: Minimum and Maximum Orbit Radius Values

Planet	Minimum Radius [10^5 km]	Maximum Radius [10^5 km]	Range [10^5 km]
Mercury	0.0538	0.2421	0.1883
Venus	0.1335	0.6006	0.4672
Earth	0.1405	0.6323	0.4918
Mars	0.0748	0.3364	0.2616
Jupiter	1.5419	6.9386	5.3967
Saturn	1.2843	5.7795	4.4952
Uranus	0.5594	2.5172	1.9578
Neptune	0.5430	2.4437	1.9007

Figure 4.2: Sample position estimation system dataset images of Mercury (minimum radius pictured left, maximum radius pictured right)

Figure 4.3: Venus sample position estimation system dataset images

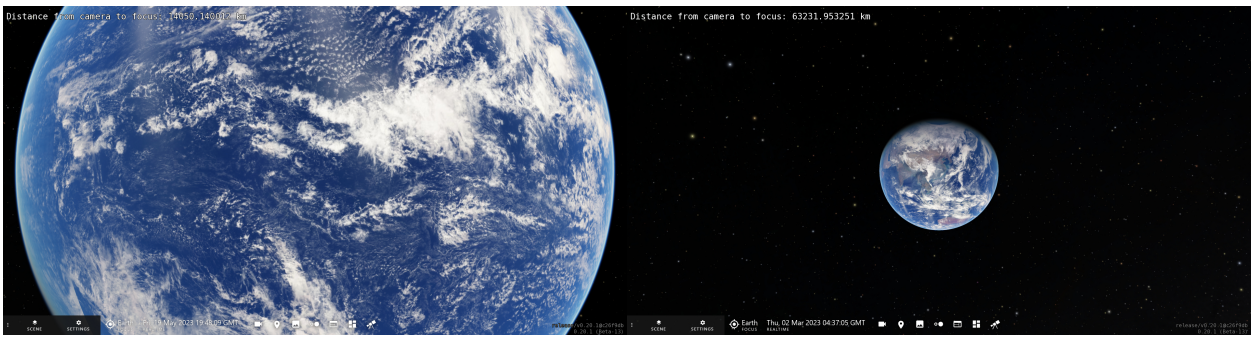

Figure 4.4: Earth sample position estimation system dataset images

Figure 4.5: Mars sample position estimation system dataset images

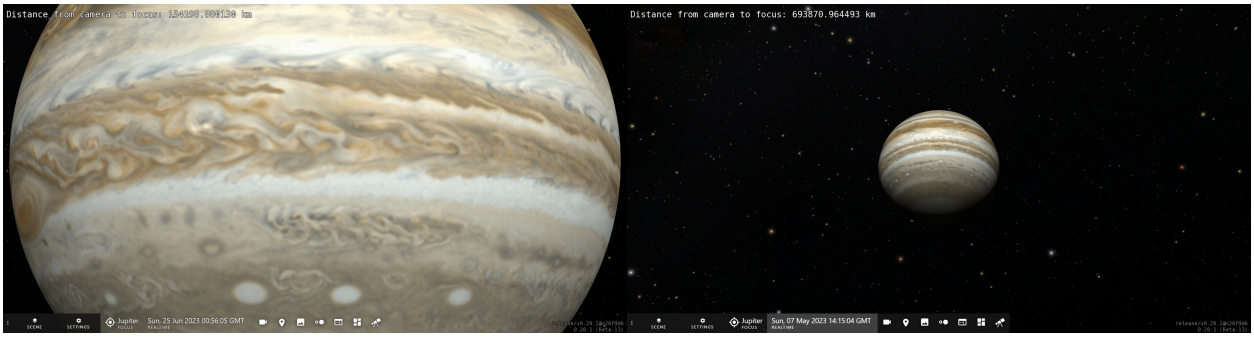

Figure 4.6: Jupiter sample position estimation system dataset images

Figure 4.7: Saturn sample position estimation system dataset images

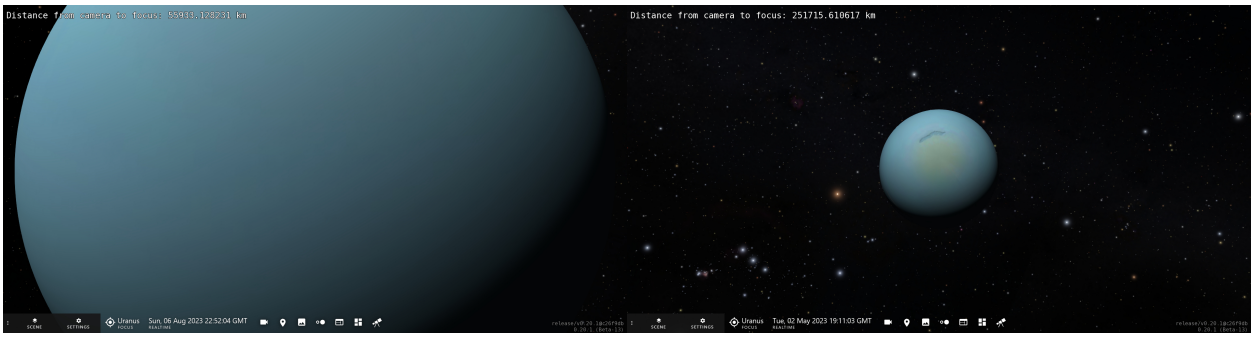

Figure 4.8: Uranus sample position estimation system dataset images

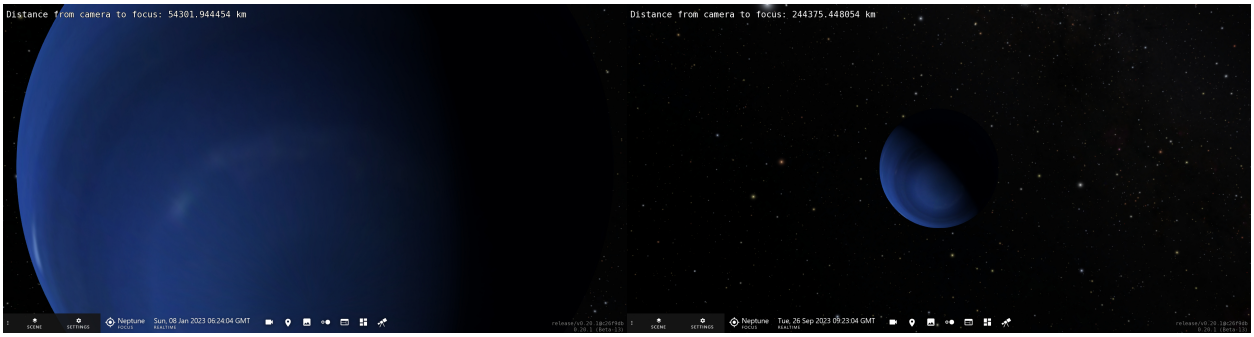

Figure 4.9: Neptune sample position estimation system dataset images

Aside from the orientation and altitude relative to the focused planet, the time and date is randomized in the year 2023. The reason why the current year is not utilized is due to the default textures used by OpenSpace to render the Earth's surface. The texture is rendered using real-time satellite imagery, meaning that the current date does not have the necessary image data to render the texture of the Earth. Thus, the image data is collected from a complete calendar year so incomplete rendering of planet surfaces does not occur. Additionally, there is an issue with using random numbers to generate a randomized date within a year. Specifically, randomizing the date while the months are also randomized causes errors since all months do not contain the same number of days. Thus, the days are limited to the lowest number of days within a month, which is 28. Besides these few caveats, the date and time is randomized within the year 2023 down to the specified minute.

Planet rendering is an inconsistent issue that requires a short delay to be included between successive executions of a script. This poses an issue with collecting data images from screenshots on a large scale autonomously as the time needed to render a planet surface can roughly take anywhere between 0.5 and 10 seconds, depending on the current resource usage of the computer while OpenSpace is running and the resolution of the texture of a planet surface. In some cases, the texture does not ever fully render due to lack of available satellite imagery. Texture rendering is an issue particularly for Earth at the North and South poles as the texture does not cover the entirety of the Earth model and appears to have a texture wrapping error which is shown in Figure 4.11. Additionally, the successive delays significantly contributes to the cumulative amount of time for data collection. As a compromise between

time and data quality, screenshots are manually observed to determine whether the planet is near fully rendered or not. If the planet is missing a significant portion of its texture in a given image (as shown in Figure 4.10), the image and corresponding altitude data entry is omitted from the dataset. However, the images of Earth with the Earth texture not fully wrapping at the North and South poles remained in the dataset.

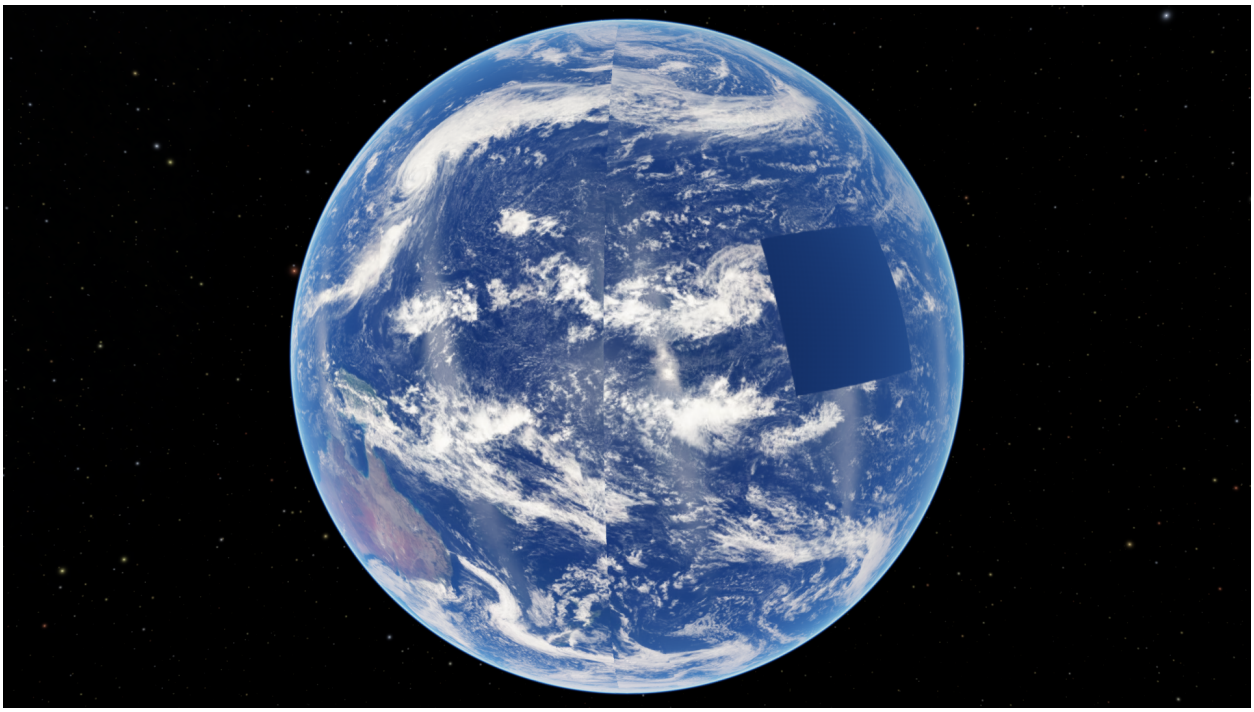

Figure 4.10: Example of image data omitted from dataset due to incomplete rendering of Earth texture

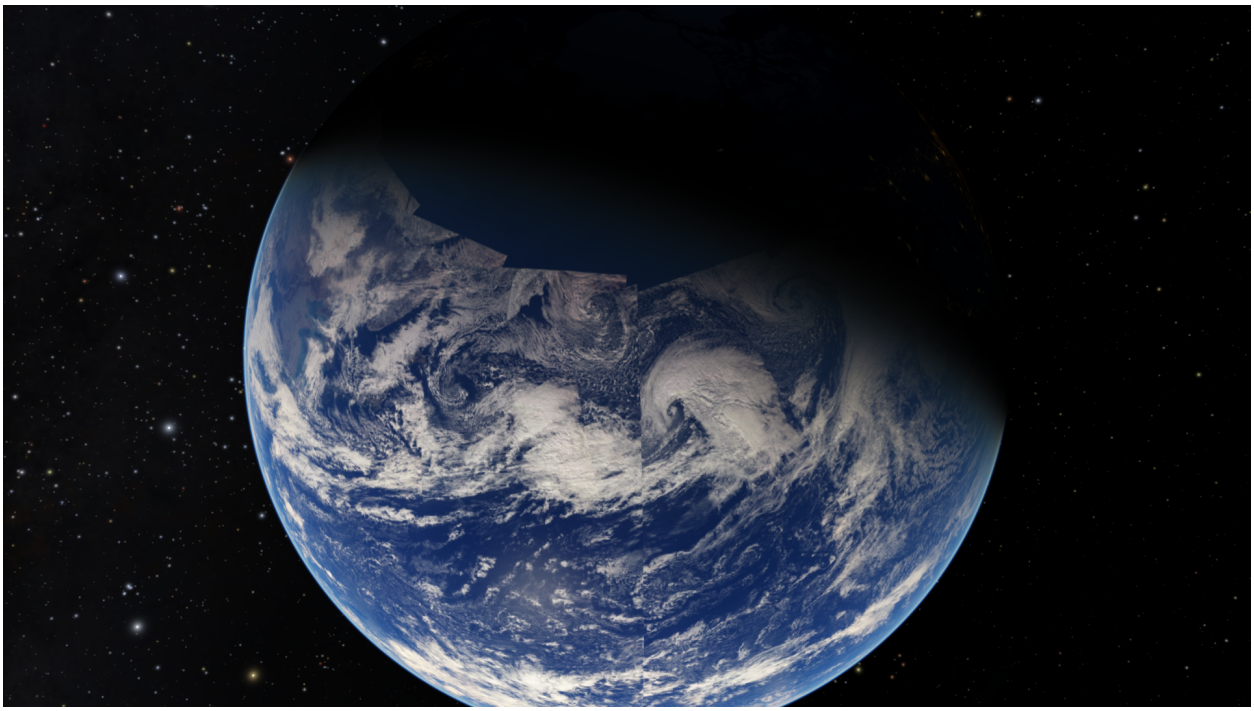

Figure 4.11: Example of incomplete texture wrapping at one of Earth's poles

4.1.2 Data for Attitude Estimation System

To develop the attitude estimation system, training and testing data must consist of image sequences of the surrounding environment captured from the point of view of a spacecraft along with the corresponding rotational rates at which the image sequence was taken. The optical flow algorithm trains on either image sequence data or video data, improving the tracking performance as it is trained. Since the only desired output for the attitude estimation system are the rotational velocities about the spacecraft body axes, there is no need to identify the spacecraft orientation based on the field of view. As with the data collected for the position estimation system, the image sequence or video data should also be standardized to a set resolution and frame rate.

To generate the dataset for training the attitude estimation system, the General Mission Analysis Tool (GMAT) can be used to generate orbital data including the rotational data of the spacecraft. The rotational data is exported from GMAT to an ephemeris file compatible with visualization software that can be used to generate video data of the simulated orbit from the spacecraft camera perspective and the corresponding rotational velocities of the spacecraft at discrete time steps. With the video data, optical flow algorithms can be applied and processed to predict rotational velocity values of the spacecraft.

4.2 Preprocessing

The data collected must be further preprocessed before it can be directly used for training the algorithms. For instance, noise can be introduced to the image dataset to test whether the algorithms perform well with realistic factors such as camera noise. The images can

also be reduced to monochromatic images for simplifying the image data, adding a realistic factor, and testing whether or not color images are essential in determining the planet and altitude.

The images in the dataset were preprocessed primarily by resizing the images. Namely, each image in the position estimation system was taken at an original resolution of 1920×1080 pixels with the FOV of the camera centered and focused about the planet in the image. The raw image from the screenshot is then cropped to the square resolution 1080×1080 pixels then the resolution is scaled down to 224×224 pixels to be suitable for training AlexNet. The geometric image transformations are performed using the OpenCV library in Python, and the code used is given in Appendix E. Due to various reasons, such as the need to omit data, the default file names of the images obtained using OpenSpace were unsuitable for training. Thus, Python scripts, that are provided in Appendices F and G, were needed to rename the images within the dataset as well as to format the altitude labels corresponding to each image.

4.3 Assumptions

There are several key assumptions made about the overall system and the observable space environment. For instance, the images obtained from the camera are assumed to be taken by a camera with a focused lens while the videos are assumed to have negligible motion blur. This is likely different from a space mission where these issues must be overcome with optical effects such as lens and motion blur being fully considered. The dataset for the position estimation system operates under the assumption that the camera is anchored about the planet: the planet of interest is centered in the field of view of the camera. In reality, such camera behavior would require a GNC system coupled with the camera FOV as well as adaptive lensing. Additionally, it is assumed that there is no effect of solar or cosmic radiation on the quality of the images or videos obtained by the camera. The distant star field is assumed to be static from the perspective of the spacecraft. Effects of solar illumination on the surfaces of planets is simulated with shaders in OpenSpace, but not necessarily found to be realistic. Furthermore, the resolution of the surface textures of some planets may be limited. Specifically, weather events such as storms on planets may not be accurately captured through the use of the OpenSpace software. The computing hardware required to store image or video data and process it through the CNN or optical flow algorithm is assumed to operate fully without any degradation due to solar radiation exposure or other harmful effects.

Chapter 5 – Results

5.1 Dataset Description

The dataset for training and testing the position estimation system consists of two primary components: an image containing 1000 or more images of each of the eight planets and a comma-separated values (CSV) file listing the image file name, the planet in the image, and the altitude value relative to the planet in kilometers. The dataset undergoes transformations to convert the image data into a tensor format and the RGB values of the image are normalized to align with the mean and standard deviation used during AlexNet's training. The altitude values corresponding to each image in the CSV files are also normalized by the maximum and minimum values that appear in the data to ensure that both tasks, classification and regression, are handled effectively. Additionally, the normalized altitude values allow for a more intuitive result when computing the root mean squared error during training, validating, and testing.

5.2 Training and Evaluation

After preprocessing the data to formats necessary for training, the AlexNet model is loaded. The last layer of the AlexNet classifier is modified to be followed by a dense, fully-connected layer that goes through a ReLU activation function, a dropout layer with a dropout rate of 40% that introduces regularization, and the final output layer with 9 neurons—8 for classifying the planet and 1 for estimating the spacecraft altitude. This adapts the AlexNet model for the position estimation task and to preserve feature extraction capabilities of the pre-trained weights and biases of previous layers.

Designing the position estimation algorithm involves three main processes: training, validation, and testing. Before any three processes are executed, the data set must be effectively distributed for each of these purposes; 70% of the data is allocated for training while 15% of the data are allocated to training and validation each. Additionally, the training process is divided into epochs in which a fraction of the total training and validation data is passed to the model. Training involves exposing the model to the image data and the corresponding planet label and altitude value for each image. Dropout ensures that during the training process, 40% of neurons are deactivated, encouraging the model to perform classification and regression formed from many different neurons rather than a few specific neurons. After each training epoch, the performance of the model is measured through the validation process. Validation is performed to test how well the model is currently able to determine the planet through classification and estimate its altitude through regression after the training process in an epoch. Thus, the model is given images from the dataset and asked to predict the planet and altitude labels or outputs. The validation process allows the monitoring of how the model's learning process is progressing and is indicative whether or not the model is overfitting on the training dataset. The loss value in each epoch quantifies the difference between the desired output and the output produced by the model. The loss in the position estimation system is defined as the sum of classification loss and the regression loss. In the position estimation system, the loss function for classification uses cross-entropy loss and mean-squared error loss for the regression. The emphasis on a given task can be implemented by introducing weights to the loss values for both tasks. The

loss values obtained through training and validation drive the backpropagation algorithm in which gradient descent to adjust the weights and bias values for each neuron to produce the most accurate output. After all training and validation has completed—that is, all epochs are complete— the model is evaluated on the unseen test dataset to perform an unbiased evaluation of its performance by computing the root mean-squared error for the regression task and accuracy for classification. Test performance also measures how well the model is to perform on new data, indicative of real-world performance. All three processes outlined are performed in a Python script using primarily the torchvision and sci-kit learn packages in tandem with torch cuda to enable graphics processing unit (GPU) computation and minimize the computational time needed. The position estimation model was trained on an NVIDIA RTX 4070Ti GPU.

Although the loss function is the quantity used for tuning the weight and bias values during backpropagation, model performance is best described through other metrics. To quantify the model performance for classification tasks, accuracy scores are calculated to determine the model performance on a given dataset. Accuracy is calculated by taking the total number of correctly-predicted planet labels over the total number of labels in the dataset. Additionally, the precision and recall performance metrics provide a more thorough summary of model performance. For each planet label class, there are four possible cases of the model's prediction: true positive, true negative, false positive, and false negative. For example, given an image of Earth, the model can output correctly that the planet is Earth (true positive), incorrectly that the planet is not Earth (false negative), incorrectly that the planet is a planet besides Earth (false positive), and correctly that the planet is not a planet besides Earth (true negative). Accuracy, precision, and recall are all calculated using the total number of these outcomes in a given dataset. Denoting the total number of cases with true positives as TP, true negatives as TN, false positives as FP, and false negatives as FN, these three performance metrics are defined as the following (5.1-3). Intuitively, accuracy measures the number of correctly-predicted values (true positive and true negatives) over all cases.

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.2}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.3}$$

Thus, precision measures how many true positive predictions occurred among all positive predictions while recall measures how many true positive predictions occurred among all actual positives. Since these four cases are only applicable to a binary classification problem (classifying as either 1 or 0), accuracy, precision, and recall can be calculated for each class in multi-class classification. In the context of the position estimation system, this means that there is an associated precision, recall, and accuracy for each planet label. To concisely describe these results, precision and recall can be stored into a single metric called the F-score or f_1 score defined in (5.4). The f_1 score is the harmonic mean of the precision and recall values.

$$f_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2 \cdot TP}{2 * TP + FP + FN} \tag{5.4}$$

The model performance for the classification task is also visualized using a plot referred to as a confusion matrix. The confusion matrix for a multi-class classification model such as the position estimation system has rows for each planet label and columns for each predicted planet label. The elements of the matrix ideally form a diagonal matrix, with the planet label and predicted planet label aligning for every image within the dataset.

The position estimation system regression task performance is quantified using the root-mean-squared error (RMSE) between the altitude label and the model's predicted altitude. Since the altitude is normalized before the training process, the RMSE is reported in both its normalized value form and un-normalized value form. The un-normalized RMSE quantifies the average error of the model's predicted altitude of the spacecraft in kilometers.

5.3 Model Results

5.3.1 Position Estimation System Performance

The Python code that performs the data preprocessing, training (with 10 epochs), validation, and testing procedures has a run time of approximately 3 minutes. Thus, experimentation with the appropriate hyperparameters as well as the weights for the loss values of each task was modified to yield better performance. Specifically, it was observed that favorable results for the planet classification task is far more achievable than favorable results for the spacecraft altitude regression task. Thus, the regression task loss was weighted so that it is two-thousand times more important than the classification task. Though this causes slower convergence for the accuracy of the classification task, the resulting test and final validation accuracies are within roughly 3% of their final value if the two tasks are weighted equally. This compromise reduces the resulting un-normalized RMSE for the model from roughly 9300 to 2800 kilometers. Figure 5.5 shows the validation loss for the position estimation system versus the epoch number when the losses for regression and classification tasks are weighted equally. This weighting of the loss values is deemed favorable for the overall model performance and is best exemplified through comparing Figures 5.4 and 5.5.

The results are visualized and tabulated as follows: the confusion matrices of the position estimation system at the start and end of the validation processes are shown in Figure 5.1 and 5.2, respectively. The confusion matrix of the model on the test dataset is provided in 5.3. All confusion matrices contain the number of cases within the box as well as a color scale to demonstrate the relative magnitude of entries. The plot of validation loss versus epoch number for the position estimation system is provided in Figure 5.4. Figure 5.5 is not representative of the final model, but presented for comparison. Tables 5.1 and 5.2 provide the performance metric values of the model for the classification and regression tasks, respectively. Figures 5.1 and 5.2 demonstrate that the model is progressively learning throughout the training process as the initial confusion matrix exhibits poor classification performance, but the model's classification performance on the validation dataset for the last epoch is significantly better.

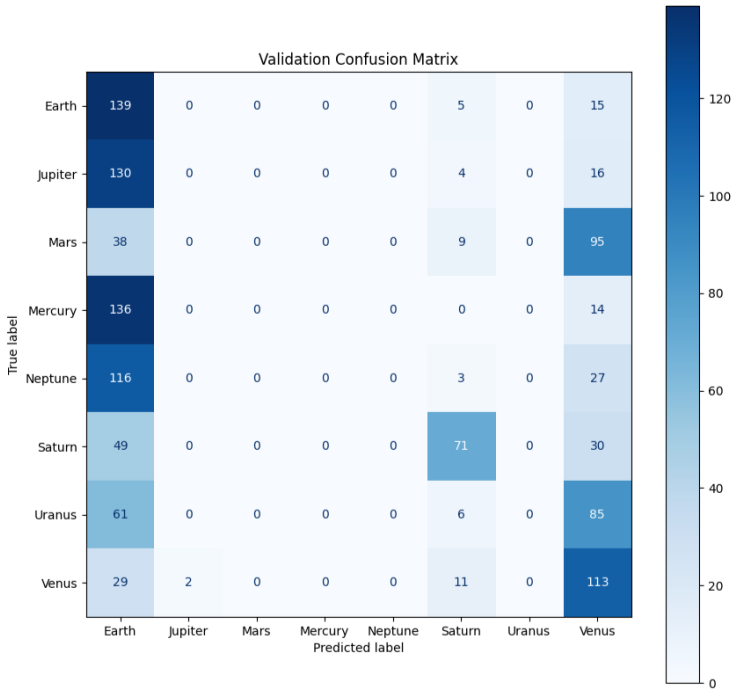

Figure 5.1: Position estimation system validation confusion matrix: first epoch

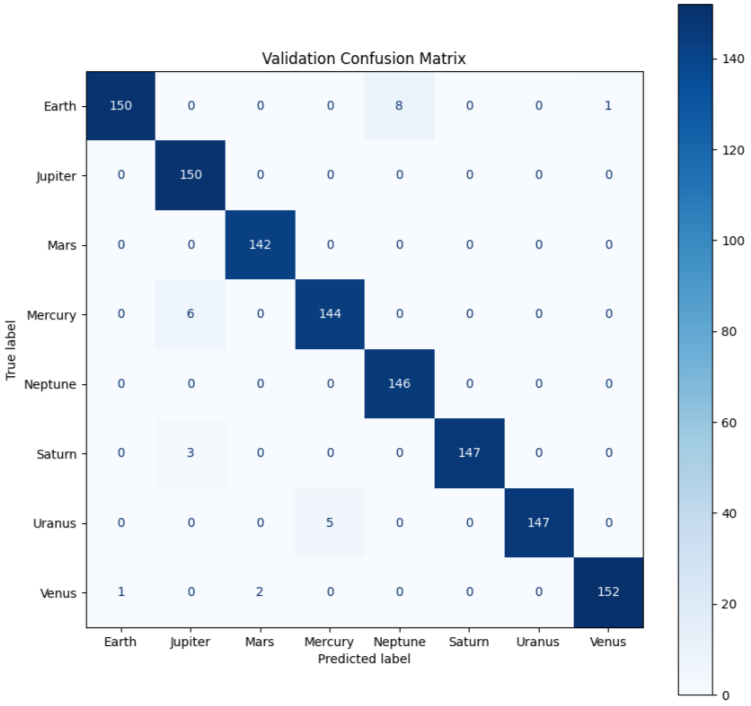

Figure 5.2: Position estimation system validation confusion matrix: last epoch

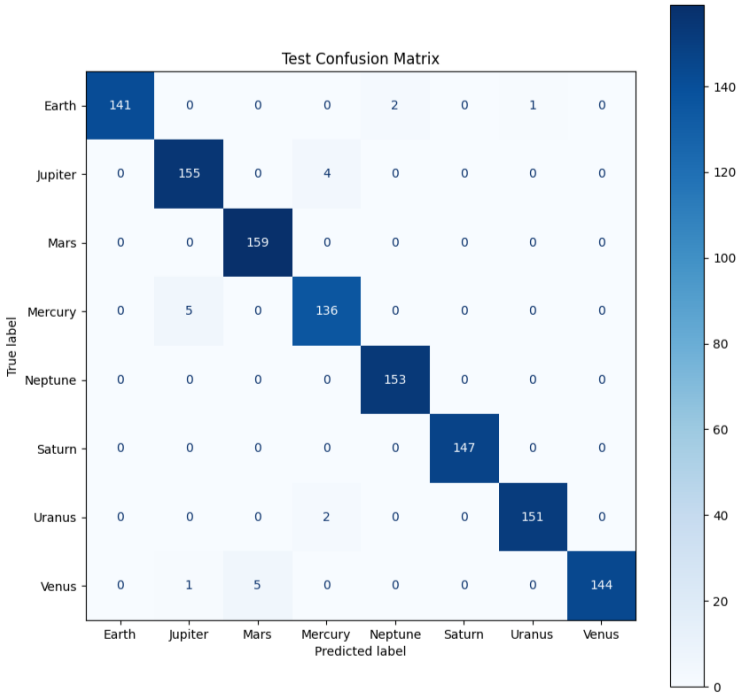

Figure 5.3: Position estimation system test confusion matrix

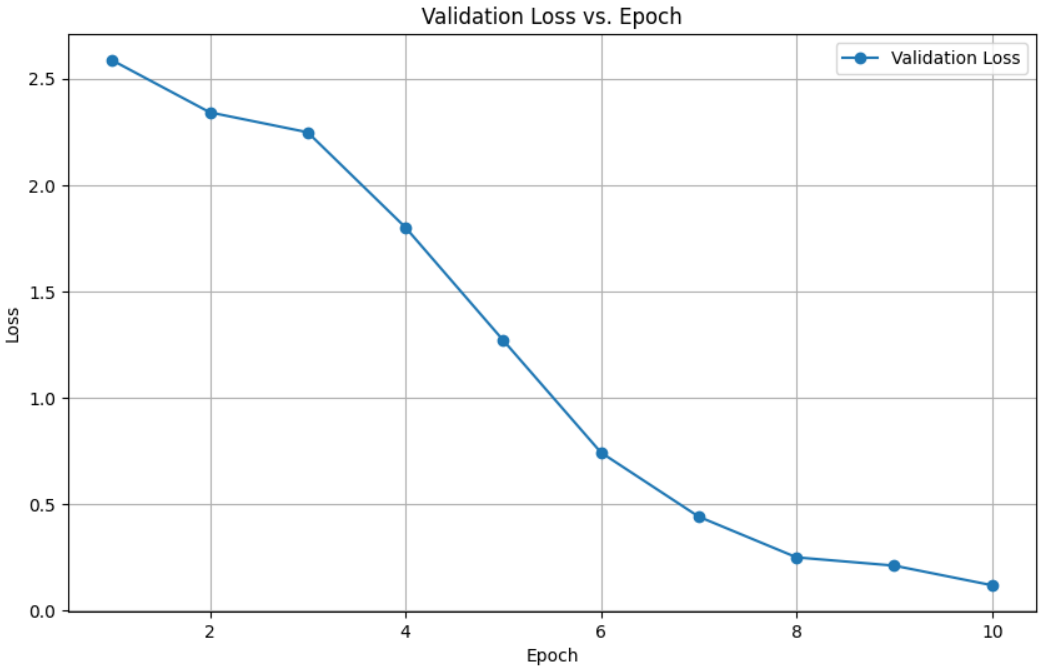

Figure 5.4: Position estimation system validation loss value vs. epoch number (weighted losses)

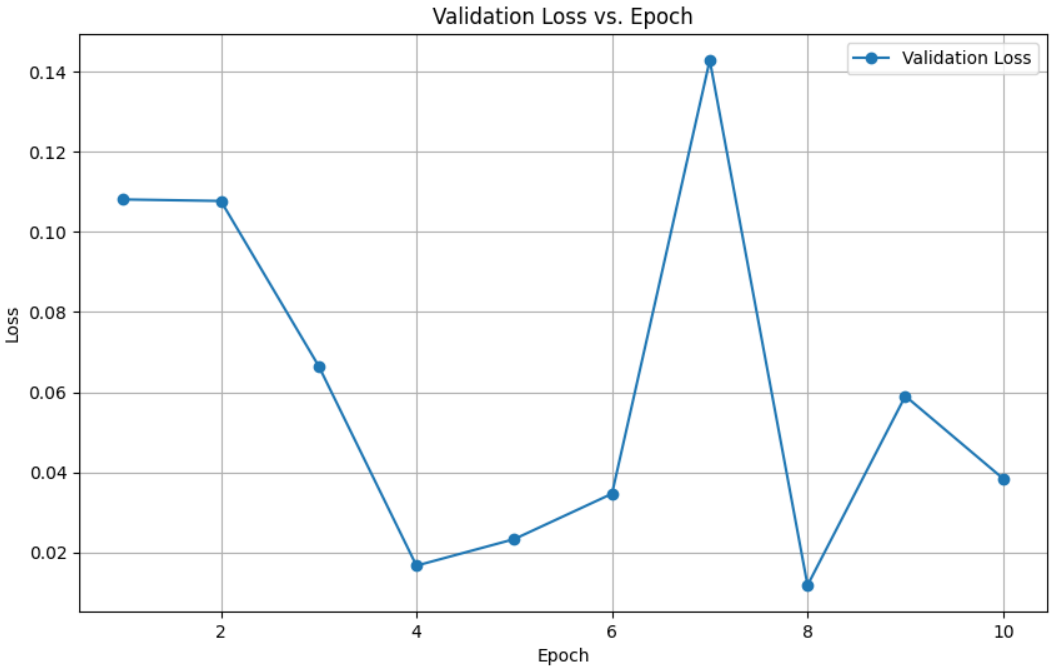

Figure 5.5: Position estimation system validation loss value vs. epoch number (equal losses)

Table 5.1: Position estimation system: classification task test performance

Label	Precision	Recall	f_1 Score	Accuracy
Mercury	0.9577	0.9645	0.9611	0.9645
Venus	1.0000	0.9600	0.9796	0.9600
Earth	1.0000	0.9792	0.9895	0.9792
Mars	0.9695	1.0000	0.9845	1.0000
Jupiter	0.9627	0.9748	0.9688	0.9748
Saturn	1.0000	1.0000	1.0000	1.0000
Uranus	0.9934	0.9869	0.9902	0.9869
Neptune	0.9871	1.0000	0.9935	1.0000
Overall	0.9838	0.9832	0.9834	0.9834

Table 5.2: Position estimation system: regression task performance

Dataset	Normalized Altitude RMSE	Un-normalized Altitude RMSE [km]
Validation (first epoch)	0.0161	9957.84
Validation (last epoch)	0.0048	2984.58
Test	0.0045	2803.01

An additional analysis on the performance of the position estimation system was conducted to inform the types of images that the model struggled to classify the planets for as well as the RMSE performance per planet. This analysis was performed on the entire dataset after the model was fully trained. File explorer previews for 9 misclassified images among the entire dataset are shown in Figure 5.6. This demonstrates that the misclassified images are all images in which the illumination of the planet surface from the camera perspective causes poor visibility of the planet. Moreover, the number of incorrectly classified images among the entire dataset imply an accuracy of 98.32%, which is approximately equal to the test dataset accuracy. The un-normalized, overall RMSE for the altitude regression task is provided for each planet in Figure 5.7. It is concluded that the model performs altitude estimation most accurately for Neptune and least accurately for Venus. It is also noted that despite the drastically different size and masses of each of the eight planets, the overall RMSE scores for the planets are reasonably similar and all are of comparable magnitude. The Python script used to conduct these analyses is provided in Appendix H.

Figure 5.6: Position estimation system: sample of incorrectly classified images from entire dataset

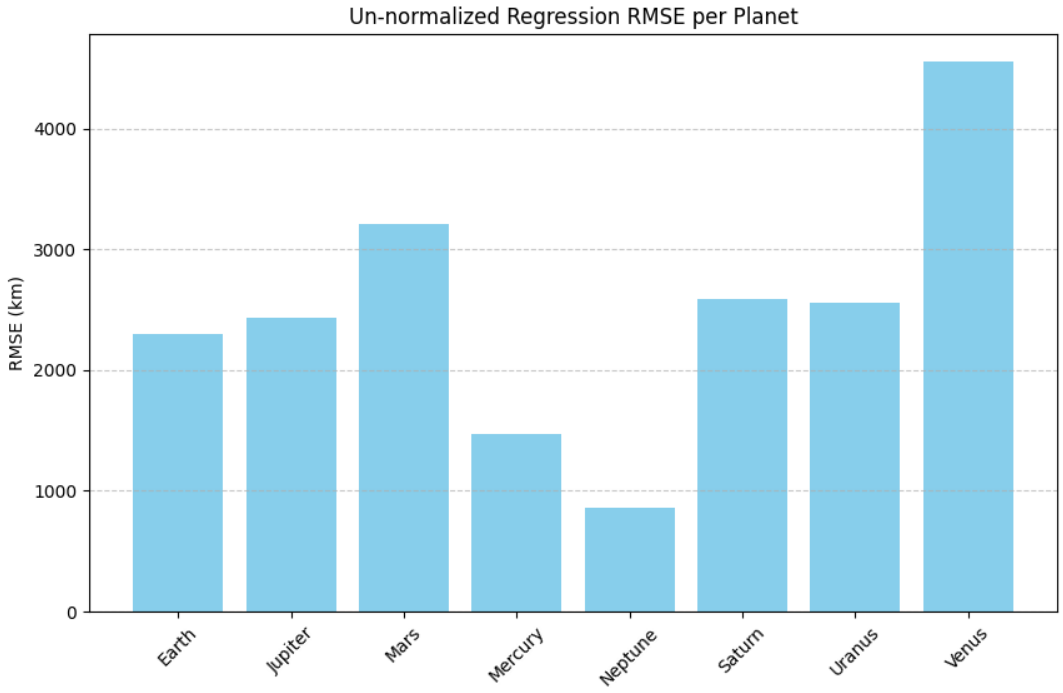

Figure 5.7: Position estimation system: un-normalized altitude regression RMSE per planet on entire dataset

5.3.2 Proposed Attitude Estimation System Methodology

A novel approach to the attitude determination system is proposed. The proposed methodology is comprised of three main phases: data generation, optical flow calculation, and mapping optical flow to angular velocities.

As mentioned before, orbital data containing the rotational velocities of the spacecraft at discrete timesteps can be generated using the software GMAT. The outputted orbital data can be exported to an ephemeris file compatible with visualization software such as Ansys STK. The visualization software is then used to generate and record video footage of the spacecraft orbit from the camera perspective of the spacecraft. Depending on the required volume of data, the data generation process can be a time-consuming process. However, for deep space missions with elaborate funding and planning, orbital simulations of the perspective of the spacecraft during different mission phases is feasible. This concludes the data collection phase of developing the attitude determination system.

During the optical flow calculation phase, a Python library with RAFT implementation, such as PyTorch, can be used to write a Python script that computes the optical flow between consecutive frames of the video. While other optical flow estimation algorithms can be used as an alternative, RAFT is currently the state-of-the-art optical flow algorithm and consists of deep learning structures to improve its accuracy. Since the proposed methodology passes through two machine learning models, accuracy in the optical flow values is of great importance. Sparse optical flow is particularly well-suited for this task as the distant star field is visually comprised of mainly empty, dark space. Since the optical flow simply yields the velocities of pixels in the image plane of the camera, the optical flow vectors must be

mapped to the rotational rates of the spacecraft.

In the final phase, mapping of optical flow to angular velocities, the velocities of pixels in the image frame undergo a mapping to obtain the angular velocities corresponding to the pixel velocities. This can be done using classical computer vision techniques such as visual odometry with the assumption that the pixels forming the distant star field are all treated as points at infinity. However, to establish a more autonomous attitude estimation system, artificial intelligence is leveraged to obtain the angular velocities of the spacecraft from the optical flow captured from its perspective. Using artificial intelligence models allows the mapping to establish complex, nonlinear relationships between the optical flow and the rotational velocities. For instance, a support-vector machine regressor, a random forest regressor, or a neural network can be used to map the optical flow values to rotational velocities. These predicted rotational velocities can be compared to those of the ephemeris file, which are treated as the ground truth, to train the model and evaluate its performance. Therefore, to ensure that the rotational velocity prediction model is performing as accurately as it can, a wide range of orbital video data is necessary for robustness.

Chapter 6 – Conclusion

The position estimation system performed remarkably well at the classification task and reasonably well for the regression task. The final validation and test dataset accuracies aligning closely demonstrates that the model is not overfitting on the training data. The AlexNet-based position estimation model is able to correctly identify the planet within the spacecraft camera view to an excellent test accuracy of 98.34%. The model also provides altitude estimates relative to the planet with an approximate average error margin of 3000 kilometers. Though this is not a trivial error margin by any means when considering close-proximity orbits of the planets, it demonstrates that despite the vastly different radii and masses of each of the planets, the model is still able to estimate the altitude of the spacecraft. This is considered to be mainly attributed to normalizing the views of each planet using the ratio between the equatorial radius of the planet and the corresponding geosynchronous orbit radius for that planet. Additionally, close-proximity orbit radii may not be as useful for spacecraft purposed for deep space missions unless planetary gravity assist maneuvers or planetary orbits are intended. The position estimation system demonstrates that existing artificial intelligence and machine learning architecture can be leveraged and purposed for spacecraft GNC systems.

Improvements and future work can be made to the position estimation system. Despite the excellent results in planetary classification, the dataset used to train and test the model is fairly limited and assumes many constraints on the spacecraft and its camera. Due to the limitations of the dataset used to train the position estimation system, there is no accurate measure of computational time within this project that is representative of a real spacecraft processing images in real time. However, it is noted that GPUs are best suited for this task as they reduce the computational time needed to train the model. For a more robust and well-trained position estimation system, a larger dataset with a variety of different image sources, both real and synthetic, should be used. Furthermore, different views of the planet from greater distances and from a larger variety of spacecraft orientations would introduce more rigor and realism to the dataset, challenging the model to learn on less ideal images. As space exploration further develops as an industry and discipline, images from real mission data can be used to train the position estimation system on a variety of stellar and interstellar object, broadening its identification capabilities. Though this project focuses on classifying the eight planets within the solar system, AlexNet is capable of categorizing images into 1000 categories, suggesting that the total number of classes (or in this case, of celestial bodies) should be expanded. While the altitude regression task performance may currently be unsatisfactory by some mission requirements, modifications to the model architecture can be made to improve the performance. For instance, decoupling the classification and regression tasks into separate models may be beneficial for the altitude regression task. Finally, as artificial intelligence and deep learning structures such as CNNs advance and improve, the position estimation system can be re-established, leveraging the available technologies of today.

The possibility of an attitude estimation system is explored and the methodology of a novel approach is outlined. Due to time constraints and issues with interfacing different soft-

ware, implementation of the proposed attitude estimation system was not feasible. While the capabilities of artificial intelligence and machine learning algorithms are vast and actively expanding, they are heavily dependent on data required for training. As such, the proposed attitude estimation system should be explored if the means to obtain the necessary video data is readily available to use in an agreeable format. This data can be obtained through spacecraft flight footage from the onboard camera perspective and accompanied by attitude data recorded by high-precision instrumentation. Additionally, the attitude estimation system should be applied to specific mission objectives that require knowledge of the rotational state of the spacecraft. This establishes an allowable margin of error in the attitude estimation through the mission requirements. With the mission requirement determined, the feasibility of using different optical flow estimation algorithms and angular velocity prediction models can also be determined and inform the volume and diversity of data required to train these algorithms to perform to designated mission standards.

References

[1] Means, H., "The Deep Space Network: Overburdened and underfunded," *Physics Today*, 1 December 2023. Retrieved 13 Nov 2024 from `https://pubs.aip.org/physicstoday/article/76/12/22/2923590/The-Deep-Space-Network-Overburdened-and.`

[2] Nielsen, M., *Neural networks and deep learning*, Determination Press, 2015.

[3] Kumar, B., "Convolutional Neural Networks: A Brief History of their Evolution," *Medium*, August 31, 2021. Retrieved 13 Nov 2024 from `https://medium.com/appyhigh-technology-blog/convolutional-neural-networks-a-brief-history-of-their-evolution-ee3405568597.`

[4] Hidaka, A., and Kurita, T., "Consecutive Dimensionality Reduction by Canonical Correlation Analysis for Visualization of Convolutional Neural Networks," ISCIE International Symposium on Stochastic Systems Theory and its Applications, 2017. https://doi.org/10.5687/sss.2017.160.

[5] Krizhevsky, A., Sutskever, I., and Hinton, G. E., "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems*, Vol. 25, edited by F. Pereira, C. Burges, L. Bottou, and K. Weinberger, Curran Associates, Inc., 2012. Retrieved 13 Nov 2024 from https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[6] "Optical Flow," , OpenCV, Retrieved 13 Nov 2024 from https://docs.opencv.org/4.x/d4/dee/tutorial_optical_flow.html.

[7] Cochard, D., "RAFT: A Machine Learning Model for Estimating Optical Flow," *Medium*, January 3, 2022. Retrieved 13 Nov 2024 from https://medium.com/axinc-ai/raft-a-machine-learning-model-for-estimating-optical-flow-6ab6d077e178.

[8] Lin, C., "A Comprehensive Guide to Motion Estimation with Optical Flow," *Nanonets*, April 24, 2019. Retrieved 13 Nov 2024 from https://nanonets.com/blog/optical-flow/.

[9] Panicucci, P., Lebreton, J., Brochard, R., Zenou, E., and Delpech, M., "Vision-based estimation of small body rotational state," *Acta Astronautica*, Vol. 213, December 2023, pp. 177–196. https://doi.org/10.1016/j.actaastro.2023.08.046.

[10] Li, X., Xu, Q., Tang, Y., Hu, C., Niu, J., and Xu, C., "Unmanned Aerial Vehicle Position Estimation Augmentation Using Optical Flow Sensor," *IEEE Sensors Journal*, Vol. 23(13), 2023, pp. 14773–14780. https://doi.org/10.1109/JSEN.2023.3277614.

[11] Guthrie, B., Kim, M., Urrutxua, H., and Hare, J., "Image-based attitude determination of co-orbiting satellites using deep learning technologies," *Aerospace Science and Technology*, January 2022. https://doi.org/10.1016/j.ast.2021.107232.

[12] Doroshchenko, I., Znamenskaya, I., Sysoev, N., and Lutskii, A., "Simulation of supersonic jet flow past a blunt body in a laboratory experiment using computer vision," *Acta Astronautica*, Vol. 215, February 2024, pp. 69–78. https://doi.org/10.1016/j.actaastro.2023.11.021.

[13] Seifouripour, Y., and Nobahari, H., "A control architecture for fixed-wing aircraft based on the convolutional neural networks," *Journal of the Franklin Institute*, Vol. 361, 2024. https://doi.org/10.1016/j.jfranklin.2024.106664.

[14] Becktor, J., Seto, W., Deole, A., Badyopadhyay, S., Rahimi, N., Talebi, S., Mesbahi, M., and Rahmani, A., "Robust Vision-based Multi-spacecraft Guidance Navigation and Control using CNN-based Pose Estimation," 2022 IEEE Aerospace Conference (AERO), Big Sky, MT, USA, 2022, pp. 1–10. https://doi.org/10.1109/AERO53065.2022.9843396.

[15] Barnett, A., "Planet Compare," 2024. Retrieved 13 Nov 2024 from https://solarsystem.nasa.gov/planet-compare/.

[16] Hunter, J., *Astrodynamics Course Reader*, Maple Press, California, 2021.

[17] Weisstein, E. W., "Spherical Coordinates," *Wolfram MathWorld*, 2024. Retrieved 13 Nov 2024 from https://mathworld.wolfram.com/SphericalCoordinates.html.

Appendix A: Equations of Motion for a Rotating Rigid Body

Define the Newtonian reference frame N formed by the orthogonal basis vectors $\hat{n}_x$, $\hat{n}_y$, and $\hat{n}_z$ and the body reference frame B formed by the orthogonal basis vectors $\hat{b}_x$, $\hat{b}_y$, and $\hat{b}_z$. Given a rigid body with principal mass moments of inertia about an orthogonal basis: I_{xx}, I_{yy}, and I_{zz}, and angular velocity ${}^N\vec{\omega}^B = \omega_x\hat{b}_x + \omega_y\hat{b}_y + \omega_z\hat{b}_z$. Euler's equations are then formulated for the rotating rigid body. The sum of external moments acting on the body about its center of mass is equal to the Newtonian time derivative of the angular momentum of the object about its center of mass.

$$\sum \vec{M}^{B/B_{cm}} = \frac{{}^N d}{dt}\left({}^N\vec{H}^{B/B_{cm}}\right),\ {}^N\vec{H}^{B/B_{cm}} = \vec{\vec{I}}^{B/B_{cm}} \cdot {}^N\vec{\omega}^B$$

$${}^N\vec{H}^{B/B_{cm}} = \left(I_{xx}\hat{b}_x\hat{b}_x + I_{yy}\hat{b}_y\hat{b}_y + I_{zz}\hat{b}_z\hat{b}_z\right)\cdot\left(\omega_x\hat{b}_x + \omega_y\hat{b}_y + \omega_z\hat{b}_z\right)$$

$${}^N\vec{H}^{B/B_{cm}} = I_{xx}\omega_x\hat{b}_x + I_{yy}\omega_y\hat{b}_y + I_{zz}\omega_z\hat{b}_z$$

$$\frac{{}^N d}{dt}\left({}^N\vec{H}^{B/B_{cm}}\right) = I_{xx}\dot{\omega}_x\hat{b}_x + I_{yy}\dot{\omega}_y\hat{b}_y + I_{zz}\dot{\omega}_z\hat{b}_z + {}^N\vec{\omega}^B \times {}^N\vec{H}^{B/B_{cm}}$$

$${}^N\vec{\omega}^B \times {}^N\vec{H}^{B/B_{cm}} = \left(\omega_x\hat{b}_x + \omega_y\hat{b}_y + \omega_z\hat{b}_z\right)\times\left(I_{xx}\omega_x\hat{b}_x + I_{yy}\omega_y\hat{b}_y + I_{zz}\omega_z\hat{b}_z\right)$$

$$= \begin{vmatrix} \hat{b}_x & \hat{b}_y & \hat{b}_z \\ \omega_x & \omega_y & \omega_z \\ I_{xx}\omega_x & I_{yy}\omega_y & I_{zz}\omega_z \end{vmatrix}$$

$$= \left(I_{zz}\omega_y\omega_z - I_{yy}\omega_y\omega_z\right)\hat{b}_x - \left(I_{zz}\omega_x\omega_z - I_{xx}\omega_x\omega_z\right)\hat{b}_y + \left(I_{yy}\omega_x\omega_y - I_{xx}\omega_x\omega_y\right)\hat{b}_z$$

$$= \left(I_{zz}\omega_y\omega_z - I_{yy}\omega_y\omega_z\right)\hat{b}_x + \left(I_{xx}\omega_x\omega_z - I_{zz}\omega_x\omega_z\right)\hat{b}_y + \left(I_{yy}\omega_x\omega_y - I_{xx}\omega_x\omega_y\right)\hat{b}_z$$

$$\frac{{}^N d}{dt}\left({}^N\vec{H}^{B/B_{cm}}\right) = \left(I_{xx}\dot{\omega}_x + I_{zz}\omega_y\omega_z - I_{yy}\omega_y\omega_z\right)\hat{b}_x + \left(I_{yy}\dot{\omega}_y + I_{xx}\omega_x\omega_z - I_{zz}\omega_x\omega_z\right)\hat{b}_y + \left(I_{zz}\dot{\omega}_z + I_{yy}\omega_x\omega_y - I_{xx}\omega_x\omega_y\right)\hat{b}_z$$

$$\sum \vec{M}^{B/B_{cm}} = M_{net,x}\hat{b}_x + M_{net,y}\hat{b}_y + M_{net,z}\hat{b}_z$$

Taking the dot product of the vector form of the Euler equations with the B frame basis vectors yields the following equations of motion.

$$M_{net,x} = I_{xx}\dot{\omega}_x + (I_{zz} - I_{yy})\omega_y\omega_z \tag{A.1}$$

$$M_{net,y} = I_{yy}\dot{\omega}_y + (I_{xx} - I_{zz})\omega_x\omega_z \tag{A.2}$$

$$M_{net,z} = I_{zz}\dot{\omega}_z + (I_{yy} - I_{xx})\omega_x\omega_y \tag{A.3}$$

Appendix B: MATLAB Code for Calculating Planet Parameters: 'planetsparameters.m'

```
clc; clear; close all;
% Masses [kg] and Rotational Periods [hrs] of Solar System
   Planets
m_mercury = 3.3010e23; T_mercury = 1407.5;
m_venus = 4.8673e24;   T_venus = 5832.4;
m_earth = 5.9722e24;   T_earth = 23.934;
m_mars = 6.4169e23;    T_mars = 24.623;
m_jupiter = 1.8981e27; T_jupiter = 9.92496;
m_saturn = 5.6832e26;  T_saturn = 10.656;
m_uranus = 8.6810e25;  T_uranus = 17.23992;
m_neptune = 1.0241e26; T_neptune = 16.11000;
mass = [m_mercury,m_venus,m_earth,m_mars,m_jupiter,m_saturn,
   m_uranus,m_neptune];
period = [T_mercury,T_venus,T_earth,T_mars,T_jupiter,T_saturn,
   T_uranus,T_neptune];
G = 6.67*10^(-20); % Gravitational Constant [N*km^2/kg^2]
geostat_rad = zeros(1,length(mass)); % Pre-allocate
for i = 1:length(mass)
    T_seconds = period(i)*60*60; % Convert hours to seconds
    geostat_rad(i) = ( G*mass(i)*T_seconds^2/(4*pi^2) )^(1/3);
       % Solve for 'a' through Kepler's Third Law
end
% Equatorial Radii of Solar System Planets [km]
rad = [0.24397, 0.60518, 0.63710, 0.33895, 6.9911 5.8232 2.5362
    2.4622]*10^4;
% Define constants A1 and A2 based off of Earth views
A1 = 1/(rad(3)/geostat_rad(3)*3);
A2 = 1/(rad(3)/geostat_rad(3)*2/3);
c1 = zeros(1,length(geostat_rad)); c2 = zeros(1,length(
   geostat_rad)); % Pre-allocate
for i = 1:length(geostat_rad)
    c1(i) = A1*(rad(i)/geostat_rad(i)); % Solve for c1
    c2(i) = A2*(rad(i)/geostat_rad(i)); % Solve for c2
end
```

Appendix C: Lua Code for Randomizing and Capturing Images of Earth: 'earth_image.lua'

```
original_state = openspace.navigation.getNavigationState()
anchor = original_state.Anchor

a_geo = 2.428311947929777e+08 --Geosynchronous Orbit Radius for
    Mercury
--Mercury Constants
c1 = 0.0222
c2 = 0.0997
r_planet = 0.24397e+07 --Equatorial Radius

function getRandomTime()
    local year = 2023 -- Set the base year
    local month = math.random(1, 12)
    local day = math.random(1, 28)
    local hour = math.random(0, 23)
    local minute = math.random(0, 59)
    return string.format("%d-%02d-%02dT%02d:%02d:00Z", year,
       month, day, hour, minute)
end

-- Function to set the navigation state
function setNavigationState(state)
    if type(state) ~= "table" then
        error("Invalid state: must be a table containing a
           valid navigation state.")
    end

    -- Setting the navigation state using the OpenSpace API
    openspace.navigation.setNavigationState(state,true)
end

function delay(seconds)
    local start = os.clock()
    while os.clock() - start < seconds do end
end

-- Spherical coordinates to constrain altitude, latitude,
   longitude range
local r = math.random(math.floor(c1*a_geo),math.floor(c2*a_geo)
   )
```

```
local phi = math.random(0,180)*math.pi/180
local theta = math.random(0,360)*math.pi/180
-- Randomize time to random day and time in the year 2023
local time = getRandomTime();
-- Define randomized state
local navigationState = {
    Anchor = "Mercury",
    ReferenceFrame = anchor,
    Position = { r*math.cos(theta)*math.sin(phi),
    r*math.sin(theta)*math.sin(phi),
    r*math.cos(phi) },
    Up = { 0.0, 0.0, 1.0 },
    Timestamp = time
}
-- Set the navigation state to the randomized value
setNavigationState(navigationState)
local dist = openspace.navigation.distanceToFocus()
local altitude = (dist - r_planet)*0.001
local file = io.open("altitude_data.txt", "a")
file:write("Altitude: " .. altitude .. " km",", Anchor: " ..
   anchor .. "\n")
file:close()
openspace.takeScreenshot()
```

Appendix D: Custom Keybinding Asset File for OpenSpace 'run_script_keybinding.asset'

```
local earthimage = {
  Identifier = "os.earthimage",
  Name = "Run Simple Test 2 Script",
  Command = [[
    dofile("C:/OpenSpace/scripts/earth_image.lua")
  ]],
  Documentation = "Runs the script earth_image.lua",
  GuiPath = "/System/Scripts",
  IsLocal = true
}

asset.onInitialize(function()
  openspace.action.registerAction(earthimage)
  openspace.bindKey("H", earthimage.Identifier)
end)

asset.onDeinitialize(function()
  openspace.clearKey("H")
  openspace.action.removeAction(earthimage)
end)
```

Appendix E: Python Script for Transforming Images: 'image_resize_batch.py'

```
import cv2
import os

def process_image(image_path, output_folder):
    img = cv2.imread(image_path)
    # Image Transformations
    img = img[0:1080, 420:1500]
    img = cv2.resize(img,(224,224))
    output_path = os.path.join(output_folder, os.path.basename(image_path)
                                                )
    cv2.imwrite(output_path, img)

input_folder = r"C:\Users\Jesse\Desktop\Master's Project - Python Code\
                                        planet images"
output_folder = r"C:\Users\Jesse\Desktop\Master's Project - Python Code\
                                        output"

if not os.path.exists(output_folder):
    os.makedirs(output_folder)

for filename in os.listdir(input_folder):
    if filename.lower().endswith((".jpg", ".jpeg", ".png")):
        image_path = os.path.join(input_folder, filename)
        process_image(image_path, output_folder)
```

Appendix F: Python Script for Renaming Image Files: 'image_rename_batch.py'

```
import os

input_folder = r"C:\Users\Jesse\Desktop\Master's Project - Python Code\
                                        earth images"
prefix = "earth"
counter = 1

for filename in os.listdir(input_folder):
    if filename.lower().endswith((".jpg", ".jpeg", ".png", ".bmp", ".tif",
                                                    ".tiff")):
        old_file_path = os.path.join(input_folder, filename)
        file_extension = os.path.splitext(filename)[1]
        new_filename = f"{prefix}_{counter}{file_extension}"
        new_file_path = os.path.join(input_folder, new_filename)
        os.rename(old_file_path, new_file_path)
        counter += 1

print(f"Renamed {counter - 1} images successfully.")
```

Appendix G: Python Script for Formatting Altitude Labels: 'labels.py'

```
import os
import pandas as pd
import re

def parse_altitude_line(line):
    match = re.search(r"Altitude: ([\d.]+)", line)
    if match:
        return float(match.group(1))
    else:
        raise ValueError(f"Invalid altitude line format: {line}")

def generate_csv(image_dir, altitude_files, output_csv):
    data = []
    planet_classes = {planet: idx for idx, planet in enumerate(
                                        altitude_files.keys())}

    for planet, altitude_file in altitude_files.items():
        with open(altitude_file, "r") as f:
            altitudes = [parse_altitude_line(line.strip()) for line in f.
                                                    readlines()]
        for idx, altitude in enumerate(altitudes):
            image_name = f"{planet}_{idx + 1}.png"
            if os.path.exists(os.path.join(image_dir, image_name)):
                data.append([image_name, altitude, planet_classes[planet]]
                                                        )
            else:
                print(f"Warning: Image {image_name} not found in {
                                                        image_dir}")
    df = pd.DataFrame(data, columns=["image_name", "altitude", "
                                        planet_class"])
    df.to_csv(output_csv, index=False)
    print(f"CSV file saved to {output_csv}")

image_directory = r"C:\Users\Jesse\Desktop\output"
altitude_files = {
    "earth": r"C:\OpenSpace\user\screenshots\EARTH DATA\earth_altitudes.
                                        txt",
    "jupiter": r"C:\OpenSpace\user\screenshots\JUPITER DATA\jupiter
                                        altitudes.txt",
    "mars": r"C:\OpenSpace\user\screenshots\MARS DATA\mars altitudes.txt",
    "mercury": r"C:\OpenSpace\user\screenshots\MERCURY DATA\mercury
                                        altitudes.txt",
    "neptune": r"C:\OpenSpace\user\screenshots\NEPTUNE DATA\neptune
                                        altitudes.txt",
    "saturn": r"C:\OpenSpace\user\screenshots\SATURN DATA\saturn_altitudes
                                        .txt",
```

```
    "uranus": r"C:\OpenSpace\user\screenshots\URANUS DATA\uranus altitudes
                                            .txt",
    "venus": r"C:\OpenSpace\user\screenshots\VENUS DATA\venus altitudes.
                                            txt"
}
output_csv_file = r"C:\Users\Jesse\Desktop\Master's Project - Python Code\
                                        labels"

generate_csv(image_directory, altitude_files, output_csv_file)
```

Appendix H: Python Script for Training, Validating, and Testing Position Estimation System: 'PositionEstimationSystemTraining.py'

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, models, transforms
from torch.utils.data import DataLoader, random_split
import pandas as pd
import numpy as np
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay,
                                    root_mean_squared_error,
                                    precision_score, recall_score,
                                    f1_score
import matplotlib.pyplot as plt
import random

# Set random seed for reproducibility
torch.manual_seed(42)
random.seed(42)
np.random.seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)

# Set device to GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Data preprocessing
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

# Dataset file paths
data_dir = r"C:\Users\Jesse\Desktop\Master's Project - Python Code\image
                                    dataset"
altitude_csv = r"C:\Users\Jesse\Desktop\Master's Project - Python Code\
                                    labels"

dataset = datasets.ImageFolder(data_dir, transform=transform)
altitudes = pd.read_csv(altitude_csv)

# Normalize altitude values
altitudes_min = altitudes['altitude'].min()
altitudes_max = altitudes['altitude'].max()
altitudes['normalized_altitude'] = (altitudes['altitude'] - altitudes_min)
                                    / (altitudes_max - altitudes_min)
```

```
# Allocate the dataset into training, validation, and test sets
data_size = len(dataset)
train_size = int(0.7 * data_size)
val_size = int(0.15 * data_size)
test_size = data_size - train_size - val_size

dataset_train, dataset_val, dataset_test = random_split(dataset, [
                                        train_size, val_size, test_size])

train_loader = DataLoader(dataset_train, batch_size=32, shuffle=True,
                                        pin_memory=True)
val_loader = DataLoader(dataset_val, batch_size=32, shuffle=False,
                                        pin_memory=True)
test_loader = DataLoader(dataset_test, batch_size=32, shuffle=False,
                                        pin_memory=True)

# Load pre-trained AlexNet model
from torchvision.models import AlexNet_Weights
model = models.alexnet(weights=AlexNet_Weights.DEFAULT)

# Modify model architecture
model.classifier[6] = nn.Sequential(
    nn.Linear(model.classifier[6].in_features, 512),
    nn.ReLU(),
    nn.Dropout(0.4),
    nn.Linear(512, 9)
)

model = model.to(device)

# Define loss functions and optimizer
classification_criterion = nn.CrossEntropyLoss()
regression_criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001) # lr=0.0001

# Function to plot confusion matrix
def plot_confusion_matrix(labels, predictions, class_names, title="
                                        Confusion Matrix"):
    cm = confusion_matrix(labels, predictions, labels=range(len(
                                            class_names)))
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=
                                            class_names)
    fig, ax = plt.subplots(figsize=(10, 10))
    disp.plot(cmap=plt.cm.Blues, ax=ax)
    plt.title(title)
    plt.show()

# Set regression weight
regression_weight = 2000

# Training loop
def train_model(model, train_loader, val_loader, epochs=10):
    val_losses = []
```

```
    for epoch in range(epochs):
        model.train()

        running_loss = 0.0
        correct = 0
        total = 0

        for inputs, labels in train_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            altitudes_batch = torch.tensor([altitudes.iloc[label.item()].
                                            normalized_altitude for
                                            label in labels], dtype=
                                            torch.float).to(device)

            optimizer.zero_grad()

            outputs = model(inputs)
            class_outputs = outputs[:, :8]
            regression_outputs = outputs[:, 8]

            classification_loss = classification_criterion(class_outputs,
                                            labels)
            regression_loss = regression_criterion(regression_outputs,
                                            altitudes_batch)

            loss = classification_loss + regression_weight *
                                            regression_loss
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            _, predicted = torch.max(class_outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        epoch_loss = running_loss / len(train_loader)
        epoch_accuracy = 100 * correct / total

        print(f"Epoch {epoch + 1}/{epochs}, Loss: {epoch_loss:.4f},
                                            Accuracy: {epoch_accuracy:.2f
                                            }%")

        val_loss, val_accuracy = validate_model(model, val_loader)
        val_losses.append(val_loss)
        print(f"Validation Loss: {val_loss:.4f}, Validation Accuracy: {
                                            val_accuracy:.2f}%")

    # Plot validation loss vs epoch
    plt.figure(figsize=(10, 6))
    plt.plot(range(1, epochs + 1), val_losses, marker='o', label="
                                        Validation Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Validation Loss vs. Epoch")
```

```
    plt.legend()
    plt.grid()
    plt.show()

    print("Training complete!")

def validate_model(model, val_loader):
    model.eval()
    val_loss = 0.0
    correct = 0
    total = 0

    all_labels = []
    all_predictions = []
    all_regression_outputs = []
    all_altitudes = []

    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            altitudes_batch = torch.tensor([altitudes.iloc[label.item()].
                                            normalized_altitude for
                                            label in labels], dtype=
                                            torch.float).to(device)

            outputs = model(inputs)
            class_outputs = outputs[:, :8]
            regression_outputs = outputs[:, 8]

            classification_loss = classification_criterion(class_outputs,
                                            labels)
            regression_loss = regression_criterion(regression_outputs,
                                            altitudes_batch)

            loss = classification_loss + regression_weight *
                                            regression_loss
            val_loss += loss.item()

            _, predicted = torch.max(class_outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            all_labels.extend(labels.cpu().numpy())
            all_predictions.extend(predicted.cpu().numpy())
            all_regression_outputs.extend(regression_outputs.cpu().numpy()
                                            )
            all_altitudes.extend(altitudes_batch.cpu().numpy())

    val_loss /= len(val_loader)
    val_accuracy = 100 * correct / total

    regression_rmse = root_mean_squared_error(all_altitudes,
                                        all_regression_outputs)
```

```
    unnormalized_rmse = regression_rmse * (altitudes_max - altitudes_min)

    print(f"Validation Classification Accuracy: {val_accuracy:.2f}%,
                                        Regression RMSE (normalized): {
                                        regression_rmse:.4f}, RMSE (
                                        unnormalized): {unnormalized_rmse
                                        :.2f} km")

    return val_loss, val_accuracy

def test_model(model, test_loader):
    model.eval()
    test_loss = 0.0
    correct = 0
    total = 0

    all_labels = []
    all_predictions = []
    all_regression_outputs = []
    all_altitudes = []

    with torch.no_grad():
        for inputs, labels in test_loader:
            inputs, labels = inputs.to(device), labels.to(device)
            altitudes_batch = torch.tensor([altitudes.iloc[label.item()].
                                            normalized_altitude for
                                            label in labels], dtype=
                                            torch.float).to(device)

            outputs = model(inputs)
            class_outputs = outputs[:, :8]
            regression_outputs = outputs[:, 8]

            classification_loss = classification_criterion(class_outputs,
                                            labels)
            regression_loss = regression_criterion(regression_outputs,
                                            altitudes_batch)

            loss = classification_loss + regression_weight *
                                            regression_loss
            test_loss += loss.item()

            _, predicted = torch.max(class_outputs, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            all_labels.extend(labels.cpu().numpy())
            all_predictions.extend(predicted.cpu().numpy())
            all_regression_outputs.extend(regression_outputs.cpu().numpy()
                                            )
            all_altitudes.extend(altitudes_batch.cpu().numpy())

    test_loss /= len(test_loader)
    test_accuracy = 100 * correct / total
```

```
regression_rmse = root_mean_squared_error(all_altitudes,
                                          all_regression_outputs)

unnormalized_rmse = regression_rmse * (altitudes_max - altitudes_min)

precision_per_class = precision_score(all_labels, all_predictions,
                                      average=None, labels=range(len(
                                      dataset.classes)))
recall_per_class = recall_score(all_labels, all_predictions, average=
                                      None, labels=range(len(dataset.
                                      classes)))
f1_score_per_class = f1_score(all_labels, all_predictions, average=
                                      None, labels=range(len(dataset.
                                      classes)))

overall_precision = precision_score(all_labels, all_predictions,
                                      average='macro')
overall_recall = recall_score(all_labels, all_predictions, average='
                                      macro')
overall_f1_score = f1_score(all_labels, all_predictions, average='
                                      macro')

class_accuracies = []
for i in range(len(dataset.classes)):
    tp = np.sum((np.array(all_predictions) == i) & (np.array(
                                          all_labels) == i))
    total_class_samples = np.sum(np.array(all_labels) == i)
    accuracy = tp / total_class_samples if total_class_samples > 0
                                          else 0
    class_accuracies.append(accuracy)

print(f"Test Classification Accuracy: {test_accuracy:.2f}%, Regression
                                       RMSE (normalized): {
                                      regression_rmse:.4f}, RMSE (
                                      unnormalized): {unnormalized_rmse
                                      :.2f} km")
print(f"Overall Precision: {overall_precision:.4f}, Overall Recall: {
                                      overall_recall:.4f}, Overall f1
                                      Score: {overall_f1_score:.4f}")
print("Test Precision per class:")
for i, class_name in enumerate(dataset.classes):
    print(f"  {class_name}: {precision_per_class[i]:.4f}")

print("Test Recall per class:")
for i, class_name in enumerate(dataset.classes):
    print(f"  {class_name}: {recall_per_class[i]:.4f}")

print("Test f1 Score per class:")
for i, class_name in enumerate(dataset.classes):
    print(f"  {class_name}: {f1_score_per_class[i]:.4f}")

print("Test Accuracy per class:")
for i, class_name in enumerate(dataset.classes):
```

```
        print(f"  {class_name}: {class_accuracies[i]:.4f}")

    plot_confusion_matrix(
        all_labels,
        all_predictions,
        class_names=dataset.classes,
        title="Test Confusion Matrix"
    )

# Train the model
train_model(model, train_loader, val_loader, epochs=10)

# Test the model
test_model(model, test_loader)

# Save the trained model
torch.save(model, "alexnet_planet_model.pth")
print("Model saved!")
```

Appendix I: Python Script for Analyzing Performance of Position Estimation System: 'PlanetRMSE_and_misclassified.py'

```
import shutil
import torch
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import root_mean_squared_error
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import pandas as pd

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model_path = "alexnet_planet_model.pth"
model = torch.load(model_path,weights_only=False)
model = model.to(device)

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
])

data_dir = r"C:\Users\Jesse\Desktop\Master's Project - Python Code\image
                                    dataset"
altitude_csv = r"C:\Users\Jesse\Desktop\Master's Project - Python Code\
                                    labels"

dataset = datasets.ImageFolder(data_dir, transform=transform)
altitudes = pd.read_csv(altitude_csv)

altitudes_min = altitudes['altitude'].min()
altitudes_max = altitudes['altitude'].max()
altitudes['normalized_altitude'] = altitudes['altitude']/ (altitudes_max -
                                     altitudes_min)

data_loader = DataLoader(dataset, batch_size=32, shuffle=False, pin_memory
                                    =True)

misclassified images
def evaluate_regression_and_misclassified(model, data_loader,
                                    planet_classes, altitudes,
                                    altitudes_min, altitudes_max):
    model.eval()

    all_labels = []
    all_regression_outputs = []
    all_altitudes = []
    misclassified_images = []
```

```
    image_paths = dataset.imgs  # Retrieve image file paths

    with torch.no_grad():
        for i, (inputs, labels) in enumerate(data_loader):
            inputs, labels = inputs.to(device), labels.to(device)
            altitudes_batch = torch.tensor(
                [altitudes.iloc[label.item()].normalized_altitude for
                                                    label in labels],
                                                    dtype=torch.float
            ).to(device)

            outputs = model(inputs)
            class_outputs = outputs[:, :8]
            regression_outputs = outputs[:, 8]  # Last output for
                                                  regression

            _, predicted_classes = torch.max(class_outputs, 1)

            for j in range(inputs.size(0)):
                if predicted_classes[j].item() != labels[j].item():
                    misclassified_images.append(image_paths[i *
                                                        data_loader.
                                                        batch_size + j][0
                                                        ])

            all_labels.extend(labels.cpu().numpy())
            all_regression_outputs.extend(regression_outputs.cpu().numpy()
                                                  )
            all_altitudes.extend(altitudes_batch.cpu().numpy())

    all_labels = np.array(all_labels)
    all_regression_outputs = np.array(all_regression_outputs)
    all_altitudes = np.array(all_altitudes)

    all_regression_outputs_unnormalized = all_regression_outputs * (
                                        altitudes_max - altitudes_min) +
                                        altitudes_min
    all_altitudes_unnormalized = all_altitudes * (altitudes_max -
                                        altitudes_min) + altitudes_min

    planet_regression_results = {}

    for planet_index, planet_name in enumerate(planet_classes):
        planet_mask = all_labels == planet_index

        planet_true_altitudes = all_altitudes_unnormalized[planet_mask]
        planet_predicted_altitudes = all_regression_outputs_unnormalized[
                                            planet_mask]

        planet_rmse = root_mean_squared_error(planet_true_altitudes,
                                            planet_predicted_altitudes)
        planet_regression_results[planet_name] = planet_rmse
    return planet_regression_results, misclassified_images
```

```
planet_classes = dataset.classes
planet_regression_results, misclassified_images =
                                        evaluate_regression_and_misclassified
                                        (
    model, data_loader, planet_classes, altitudes, altitudes_min,
                                            altitudes_max
)

print("Regression RMSE (in km) for each planet:")
planets = []
rmses = []

for planet, rmse in planet_regression_results.items():
    if rmse is not None:
        print(f"  {planet}: {rmse:.2f} km")
        planets.append(planet)
        rmses.append(rmse)
    else:
        print(f"  {planet}: No samples available")

plt.figure(figsize=(10, 6))
plt.bar(planets, rmses, color='skyblue')
plt.xlabel("Planets")
plt.ylabel("RMSE (km)")
plt.title("Un-normalized Regression RMSE per Planet")
plt.xticks(rotation=45)
plt.grid(axis="y", linestyle="--", alpha=0.8)
plt.show()

print("Misclassified Images:")
for image_path in misclassified_images:
    print(image_path)

output_dir = r"C:\Users\Jesse\Desktop\Master's Project - Python Code\
                                        Misclassified_Images"

for image_path in misclassified_images:
    shutil.copy(image_path, output_dir)

print(f"Misclassified images copied to: {output_dir}")
```