Deep Learning Imitation of Particle Filter for Autonomous Vertical Optical Lunar Lander

Howard A. Beck [*]
Massachusetts Institute of Technology, Cambridge, MA 02139

Roberto Furfaro [†]
University of Arizona, Tucson, AZ 85721

Brian Gaudet [‡]
University of Arizona, Tucson, AZ 85721

A convolutional recurrent neural network was developed for optimal state estimation of altitude, velocity, and mass in a simulated vertical lunar landing using only images taken from an on-board camera. The ground truth optimal estimates for training were calculated using a Rao-Blackwellized particle filter, which used particles for position estimation. As of now, the RBPF used a simplified model of the lunar terrain to cut computation time, and generated initial particles close to the real altitude to allow fewer particles to die out after the first image measurement. This allowed the RBPF to more accurately approximate the optimal state estimate, with a larger effective number of particles from the start. Currently, the neural network is able to produce outputs in the correct range, and the simulated lander could stay above the surface for the entire simulated time, only requiring a slight modification to the control law. Neural networks can be quickly executed using graphics processing units (GPUs) which may be available on next-generation flight computers, meaning they could help overcome the curse of dimensionality and computation time required for accurate particle-based state estimation as heavy computations are performed offline.

I. Introduction

As the need for more precise landings in space operations increases, so will the precision needed in navigation systems. Accurate navigation systems are required for future missions on the Moon [1] and even around the solar system that will require precise landings, which are of interest for several reasons. They help solve the hazard avoidance problem, as a landing area can be predetermined and targeted that doesn't present any hazards to the lander. In the case of robotic exploration, they can also relax design criteria as rovers will not have to travel large distances to sites of interest. Shorter distances also allow rovers to have reduced risks of malfunction, while potentially lowering costs and leaving space for more equipment. Precise landings can also allow for exploration of rough, inaccessible terrain on bodies such as the Moon [2] and Mars [3], or accurate landings on small bodies [4].

A large amount of recent research has gone into developing navigation algorithms for the purposes of such precise landings. For example, Gaudet and Furfaro [5] used a RBPF to develop a navigation system for a Mars landing with radar altimetry, and note that using a GPU could speed up their algorithm even further. Of particular interest to this study however are image-based navigation systems for lunar landings. Roumeliotis et al. [6] use image-based motion estimation (IBME) from images taken while landing, combined with data from an inertial measurement unit (IMU), to develop a navigation system using an Indirect Kalman filter. Bilodeau et al. [7] developed a Terrain-Relative Absolute Navigation (TRAN) system for lunar landings that relies on identifying craters for position estimation. A review for terrain relative navigation approaches using passive imaging and active range sensing such as LIDAR can be found in Johnson and Montgomery [1].

Due to the great expected computational power of next-generation flight computers with graphics processing units

[*]Incoming freshman, E-mail:hbeck@mit.edu
[†]Professor, Department of Systems and Industrial Engineering, Department of Aerospace and Mechanical Engineering
[‡]Engineer

(GPUs), artificial intelligence has been increasingly studied as a means for solving space-related problems. Of particular interest to this study is AI applied to developing guidance, navigation, and control (GNC) systems, though other applications can be found in surveys by Izzo et al. [8] and Kothari et al. [9]. To address the problem of guidance and control on Mars, Gaudet et al. [3] make use of deep reinforcement learning to create an integrated guidance and control (IGC) system that approximates a fuel-optimal control law for a 6-DOF powered descent. In another work, Furfaro et al. [10] use deep convolutional and recurrent neural networks to translate sequences of images taken from a lunar lander, as generated by POV-Ray using a Digital Terrain Map (DTM) of the Moon, into 1-D and 2-D fuel-optimal trajectories as calculated by MATLAB's Gauss Pseudospectral OPtimization Software (GPOPS). However, this method doesn't estimate the lander's state explicitly. Though this approach is sufficient for controls, having a state estimator is still important for telemetry and in understanding how accurately the state can be controlled given the available data. Separating estimation and controls also allows for the control system to be more easily changed, and in the case of a deep learning control system, allows it to more easily be retrained as a state vector has a much lower dimension than an image. There has been progress towards using deep learning for state estimation, such as in the study by Campbell et al. [11] where they trained a deep neural network to estimate an optical lunar lander's position along one dimension, with images generated in much the same way as the previous study. Outside of moon landings, Proença and Gao [12] have also used deep learning to estimate the pose of a spacecraft given images of it.

However, there is still a gap in combining deep learning navigation systems with previously mentioned state estimation algorithms. Many of these DL approaches do not provide estimates of hidden states such as translational and rotational velocities. This paper makes progress in studying DL for state estimation by training a neural network to learn the output of a particle filter, specifically, a Rao-Blackwellized particle filter, with image data used as measurements. Such an approach would allow the navigation system to produce better informed estimates about its state, as it uses a Bayesian algorithm based on past measurements. It could also in theory not suffer from many of the RBPF's issues, as it can be trained on a number of particles that is not feasible on-board. This would allow it to be more accurate than a RBPF that can be used in practice. Should more dimensions or measurements be added, a similarly sized network could be trained, which could help avoid the curse of dimensionality at the expense of more time spent generating training data and training the network.

Approximating the optimal state estimate as opposed to the real state may also have the potential to reduce over-fitting in the neural network. This is because the optimal state estimate is in theory a deterministic function of all previous measurements and control inputs, while that data is not sufficient to perfectly predict the real state. While this distinction is small, it may prove useful in systems with a large uncertainty in state estimates. In such examples, the real state and optimal state estimate may be distant in the state space, leading to significant noise in training data. Of course, the Monte Carlo nature of particle filters restrict the extent to which this distinction could be important, but this can be improved by adding more particles.

II. Method

To determine if a deep learning approach could be used to imitate a particle filter for vertical optical lunar landers, a training set was generated for the deep neural network to use to learn from sample simulated trajectories. In each trajectory, a simulated lunar lander had a controlled landing, which was cut off at the point where optical navigation wasn't feasible due to a lack of detail in the surface. The landing site was the Apollo 16 landing site, which was chosen due to the availability of terrain data around that area as well as use in similar research testing AI capabilities for lunar landings, such as Campbell et al. [11]. At each time step, a 64x64 pixel grayscale image was generated, representing what a camera at the lander's position and pointing down would observe. After each image was generated, a particle filter was used to estimate three state variables of the lander: altitude, vertical velocity, and mass. These estimates were then used with the Zero-Effort-Miss/Zero-Effort-Velocity (ZEM/ZEV) control law to command a new thrust to the lander, after which the physics of the lander were simulated to update its real state for the next time step. These state estimates from the RBPF were used as ground truth optimal state estimates, which the deep learning algorithm learned to estimate.

Specifically, a Rao-Blackwellized particle filter was used as opposed to a regular particle filter, since they tend to increase accuracy while simultaneously reducing the number of particles needed. Because RBPFs only use particles to estimate part of the state vector, they require fewer particles than a regular particle filter would need to estimate all the states. This means less computation time was needed due to fewer particles, which also had the advantage of allowing more data points to be calculated to for training.

A. Equations of Motion, Constraints, and Simulation

Due to the motion of the lander being constrained to the vertical direction, the model of this lander only considered one thruster pointed up being used. The equations of motion of a vertical lunar lander can be derived using Newton's laws and the rocket equation. Specifically, the total force on the lander in the vertical direction is given by:

$$F = T - mg$$

where T is the upwards pointing thrust force generated by the engines and g is the local gravitational field at the lander's position. The acceleration of the lander can then be found by dividing the net force by the lander's mass, which can be written as:

$$\frac{dv}{dt} = \frac{T}{m} - g \tag{1}$$

where dv/dt denotes the rate of change of velocity v, equivalent to the vertical acceleration. Of course, the rate of change of position over time is given by the velocity:

$$\frac{dy}{dt} = v$$

As propellant gets expelled by the engines, the lander holds less mass. This rate of change in mass can written in terms of the engine's specific impulse, namely:

$$\frac{dm}{dt} = \frac{T}{I_{sp}g_0}$$

Here, I_{sp} is the specific impulse of the engine, and g_0 is the gravitational field on Earth that the specific impulse is measured in.

These derivations assume there are no imperfections in the physics model, which is not true in real hardware. To model some of this uncertainty, the thrust T is considered to be a random variable, which takes on a different value at any given time. This can be seen as being caused by random deviations in the amount of propellant being exhausted due to hardware imperfections. The real value of thrust T was chosen to be normally distributed with a mean of T_{cmd}, the commanded thrust, and a standard deviation of 1% of T_{cmd}. Alternatively, this can be written as:

$$T = T_{cmd}(1 + 0.01w)$$

where w is a normally distributed variable with mean 0 and a standard deviation of 1.

Now, the equations of motion can be modified using the theory of stochastic differential equations to account for this random noise. The position, velocity, and mass states are combined into a state vector x:

$$x = \begin{bmatrix} y \\ v \\ m \end{bmatrix}$$

The differential equation describing how all states in z change over time is given by:

$$dx = d\begin{bmatrix} y \\ v \\ m \end{bmatrix} = \begin{bmatrix} v \\ \frac{T_{cmd}}{m} - g \\ \frac{T_{cmd}}{I_{sp}g_0} \end{bmatrix} dt + 0.01T_{cmd}\begin{bmatrix} 0 \\ \frac{1}{m} \\ \frac{1}{I_{sp}g_0} \end{bmatrix} dW \tag{2}$$

where W is a Wiener process, which can be interpreted as a random variable with a rate of change with a mean of 0 and standard deviation of 1. Each row of this expression determines how the respective state variable changes given a change in time and a change in the Wiener process.

To update the state vector, equation (2) was numerically solved at every time step using a stochastic version of the Improved Euler method, described by Roberts [13], with 10 subintervals. This algorithm can efficiently approximate such stochastic differential equations.

The specific values of constants for the simulation are given below. Many of the following values for constants were taken from Furfaro et al. [10] due to the similarities with this research in order to provide reasonable values. Namely, the authors also develop a neural network for a vertical lunar landing relative terrain navigation system, using a digital terrain model of the Apollo 16 landing site. In this case, the value $g_0 = 9.80665\text{m/s}^2$ was used, which is the standard value of the gravitational field on the surface of the Earth. Additionally, for simulating the motion of the lander on the moon, the gravitational field was taken to be constant and equal to be 1.622m/s^2. The specific impulse of the thruster was set to 200s. As for the initial conditions of the lander, the starting state was chosen from a Gaussian distribution, with mean

$$\bar{x} = [1250\text{m}, -8\text{m/s}, 1300\text{kg}]^T$$

and a standard deviation of $83.\bar{3}$ meters in position, $0.\bar{6}$ meters per second in velocity, and 5 grams in mass. These values create a 3-sigma ellipse contained inside the initial conditions used in Furfaro et al. [10].

B. State Estimation and Rao-Blackwellized Particle Filter

The ground truth mean of the lunar lander's state were calculated using a Rao-Blackwellized Particle Filter. In this implementation, the position was estimated using particles, while the marginalized states were velocity and mass. These marginalized states were estimated using an Unscented Kalman Filter. Though formally marginalized states for a RBPF should be estimated using an optimal filter, no such filter exists for this specific system due to the nonlinearity introduced by dividing thrust by mass. However, due to the large mass, it is approximately linear enough that an UKF can provide a near-optimal estimate, and can thus be used.

It is standard to initialize particles with the same distribution as the state variable - in this case, with a mean of 1.25km and standard deviation of $83.\bar{3}$m. However, images taken at that height from the surface had enough information to estimate the altitude to within a few meters, and most particles ended up dying as a result. Images taken at those points were too different from the measurement image, and thus the particle weights quickly went to 0 as the probability of the real position being the particle's position was low. This was accounted for by instead generating particles with a mean at the real position and variance of 2 square meters. After the first measurement, the variance in the particle filter's estimate was almost always less than 2 square meters, so this ensured that the particles could accurately represent the real distribution instead of being too close together. This caused particles to be closer to the real position and thus this problem didn't arise as often, since images captured at each particle were more similar to the measurement image. It's worth noting that due to the initial distribution of particles being different from the initial distribution of the state, the weights had to be calculated as:

$$w_0^i \propto \frac{\text{N}(y_{real}, \sqrt{2})(y^i)}{\text{N}(1250, 83.\bar{3})(y^i)}$$

where $\text{N}(x, \sigma)$ represents a Gaussian probability distribution with average x and standard deviation σ, w^i is the weight of the i-th particle, and y^i is the altitude of the i-th particle.

Despite this, there were still many particles which had 0 weight after the first measurement. While there are techniques such as resampling particles that get rid of these "dead" particles, it was found that resampling at the start reduced the quality of future estimates greatly. This can be explained by the fact that resampling removes variance from the set of particles - less examples of states are being represented with the same number of particles, and when there were already few useful particles to begin with, this caused there to be even less variance. Instead, more particles were added for the first few seconds, and then they were resampled into a lower number of particles for the rest of the trajectory.

Due to time constraints of this project so far, the RBPF used 1000 particles for the first 5 seconds of the landing, and 100 for the rest of the landing. This had a good balance of accuracy and computation time, though there are plans to increase the number of particles in both segments to reduce the amount of noise.

The estimates from each state were calculated as a weighted average across all particles of the particle's position, and the mean velocity and mass calculated from its Kalman filter, weighted by the particle's weight. Due to the relatively low number of particles, some sample trajectories did have an unreasonable amount of error. To determine which trajectories could not be used, the RBPF also calculated the standard deviations of each state variable. If at any point in the landing the difference between any real and estimated states exceeded 4 standard deviations, then the entire trajectory was considered invalid. Normally, data is almost never above 3 standard deviations from the average. However, the particle filter produces results that have a higher amount of randomness due to its nature as a Monte Carlo algorithm, and the amount of particles used was not enough to smooth out that random noise enough. Ultimately, about one fourth of trajectories were invalid due to this reason, but to compensate more were generated in order to get enough training data.

At each time step, the RBPF represented a distribution of states as a set of particles representing positions and an a corresponding average velocity, mass, and their covariances for each particle. When the lander's real state was updated, a sample velocity and mass were generated for each particle, based on the mean and covariance of that particle's corresponding UKF. This strategy meant the position was updated in time in the same way as particles were updated, meaning that recalculating weights was equivalent to multiplying them by $\rho(y^i|z)$, meaning the probability density that the altitude y^i of the i-th particle corresponds to the output image z. The output image z can be found using the equation:

$$z = h(y) + v$$

where $h(y)$ is a mathematical function that represents the image render at position y, and v is additive noise representing errors in the camera's measurement. In this case, z is a 4096-dimensional vector representing grayscale values for each pixel in the 64x64 images. The noise that was added as v had an average of 0 and standard deviation of $5/255$ in grayscale, which provided a noticeable amount of noise when images were captured. Determining $\rho(y^i|z)$ is then reduced to determining the probability density of $z - h(y^i)$, meaning that an image must be rendered at each particle. Calculating this density reduces to taking the product across all pixels of the probability distribution that the pixel in the measurement image was taken from the same position as the corresponding pixel in the particle image. Using product notation, this is

$$\rho(y^i|z) = \prod_{j=1}^{4096} N(0, 5/255)(z_j - h(y^i)_j)$$

where a subscript of j represents considering the j-th pixel only.

After the particle filter calculated new estimates at each time step, it also calculated the "effective number of particles," given by the inverse of the sum of all inverse square weights. Using summation notation, this gives:

$$N_{eff} = \frac{1}{\sum_i \frac{1}{w_i^2}}$$

N_{eff} can be loosely interpreted as how many particles are actually contributing to the calculation instead of being dead. When N_{eff} was too low, particles needed to be resampled in order to make the best use of all the computational time spent on all particles. When N_{eff} was less than half of the total number of particles, which was 100 or 1000 depending on the time, the particles were resampled using the systematic resampling algorithm. For a description and comparison of common resampling algorithms, see Douc and Cappé [14]. Resampling tends to replace particles with low probabilities with copies of particles with high probabilities, in order to more accurately represent a probability distribution without considering zero weights. The same resampling algorithm was also used to reduce the number of particles from 1000 to 100 at time $t = 5$.

Once the particles were updated with new positions y_t^i , the Unscented Kalman Filter corresponding to the particle was updated to provide a posterior estimate of the velocity and mass of the preceding time step: $v_{t-1}^{i,+}$ and $m_{t-1}^{i,+}$. This can be done as the position at time t is a function of the Kalman filter state - velocity and mass - and position, all at time $t-1$. By updating this estimate, the uncertainty in the estimate of velocity and mass at time $t-1$ can be reduced, which then improves estimates of the future velocity and mass at time t. Thus, the next step was to predict the velocity and mass and their covariance at time t using a Kalman filter predict step. One complication was that when using y_t^i as an output that the Kalman filter uses to improve its estimate, there is cross-correlation between the process and measurement noise. This is because the only source of uncertainty comes from the noise in the real thrust being applied, which affects both how velocity and mass change over time, and also as a consequence, how the position changes over time. The fix for this is detailed in Schon et al. [15] for a regular Kalman filter used for Rao-Blackwellized particle filtering, but a similar algorithm for which was used for Unscented Kalman Filters can be found in Chang [16].

C. Control Law

Once the RBPF estimated the states of the lander, the ZEM/ZEV control law was used to move it towards the target position and velocity. ZEM/ZEV calculates the Zero-Effort-Miss and Zero-Effort-Velocity, which are the position and velocity at the end of the landing if no more thrust were to be applied, and uses these to calculate a new thrust to apply. Specifically, if y_T and v_T are the target position and velocity and t_{go} is the time until the lander should arrive at its target, then:

$$ZEM = y_T - \left(y + vt_{go} + \frac{1}{2}gt_{go}^2\right)$$

$$ZEV = v_T - \left(v + gt_{go}\right)$$

where y, v, and m are the position, velocity, and mass of the lander and g is the gravity of the Moon. The thrust the lander applies is then given by:

$$T = m\left(\frac{6}{t_{go}^2}ZEM - \frac{2}{t_{go}}ZEV\right)$$

The targeted position and velocity was 50m and −2m/s, which put the lander above the surface to a point where images were indistinguishable. At that point, a real lander would need to use another sensor such as an altimeter for landing. The ZEM/ZEV control law was chosen due to being both easy to implement and effective at controlling the landing. For a more thorough description of ZEM/ZEV along with other applications, see Hawkins et al. [17].

Uniform random noise was added to the thrust, of 20-200 Newtons, with the specific value decreasing linearly over time. This was done to ensure that the neural network would not memorize the thrust and would instead need to be able to generalize to other ways of calculating a thrust to apply, as ZEM/ZEV is not the only control law. This specific amount of noise was found to provide a good amount of noise in the thrust while still guaranteeing a landing.

D. Image Generation

For the simulator, images were generated using Blender, a 3D modeling software. This software was picked due to its general ease of use, abundant documentation, and ability to seamlessly integrate Python scripts into its rendering. It's worth noting that the purpose of the image generation was not to be hyperrealistic or accurate, but rather to be able to quickly produce images for the particle filter that matched reality enough that they could show promise for future application given this project's time constraints.

A Digital Terrain Model (DTM) of the 6x6 kilometer patch of the moon's surface around the Apollo 16 landing site with 2 meter resolution was used to model the landing area. The DTM was taken from publicly available data from the Lunar Reconnaissance Orbiter. This resolution was found to be sufficient, given the images were relatively low quality, being only 64x64, and thus not too much fine detail was needed in the surface.

It was found that a typical approach to generating images from a DTM was too slow for the time constraints of this project so far, given how many images had to be generated for each trajectory for the particle filter to work. Therefore, 3 separate Blender files were created, each one using the last in some way, and being faster to render than the last. The first file was such a typical approach mentioned above, and the others used some optimization to speed up rendering. In the end, the particle filter used the third file for generating rendered images.

In the first file, the DTM was loaded into a displacement shader for a 6-by-6 square meters (with one meter in Blender corresponding to 1 kilometer in real life) 2D plane in Blender in order to add accurate height variation. The shader had a scale of 0.001 to convert kilometers in the DTM to Blender meters. The plane was also given a principled

BSDF material, and its displacement setting was set to “Displacement and Bump" in order to generate accurate rendering given the height displacement. The plane was configured to be adaptively subdivided, meaning every time the image was rendered, Blender automatically divided the plane into enough smaller segments to provide sufficient accuracy for the image. Without any sort of subdividing, the plane would just remain flat but angled as the corners had their heights displaced. Sunlight was represented as coming from a Blender Sun object, with an X rotation of -65.3 degrees in order for light to come in at an angle and cast a shadow on the surface, as would be expected during a real landing. To produce a rendering that was reasonably faithful to real life, the object’s base color to the shade of gray specified by hexadecimal color #717171. The plane’s material’s specularity was set to 0 to reflect the moon’s surface’s low amount of light reflection, and the roughness to 1 due to the moon’s terrain’s roughness. Again, these values were not chosen to product hyperrealistic simulated images, but rather to give a ballpark estimate of what a captured image might look like. All other settings in the BSDF were not changed from their default values. The Cycles render engine was used in order to accurately calculate how sunlight would bounce around the surface, with an experimental feature set to enable adaptive subdivisions. Cycles was configured to use the GPU Compute device, in order to speed up rendering when applicable. Using this method, images took about 3 seconds to render.

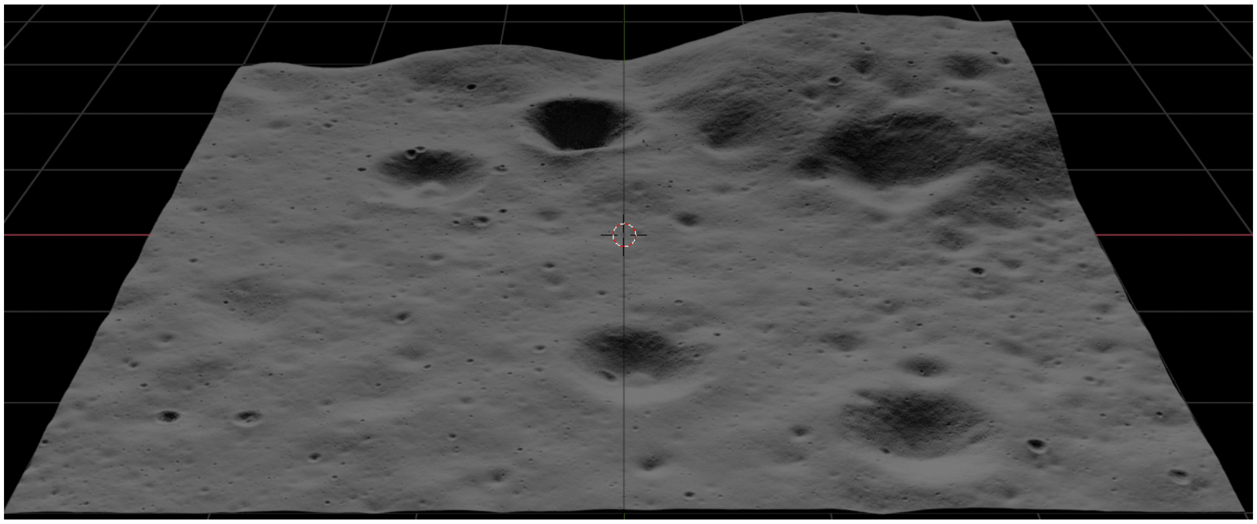
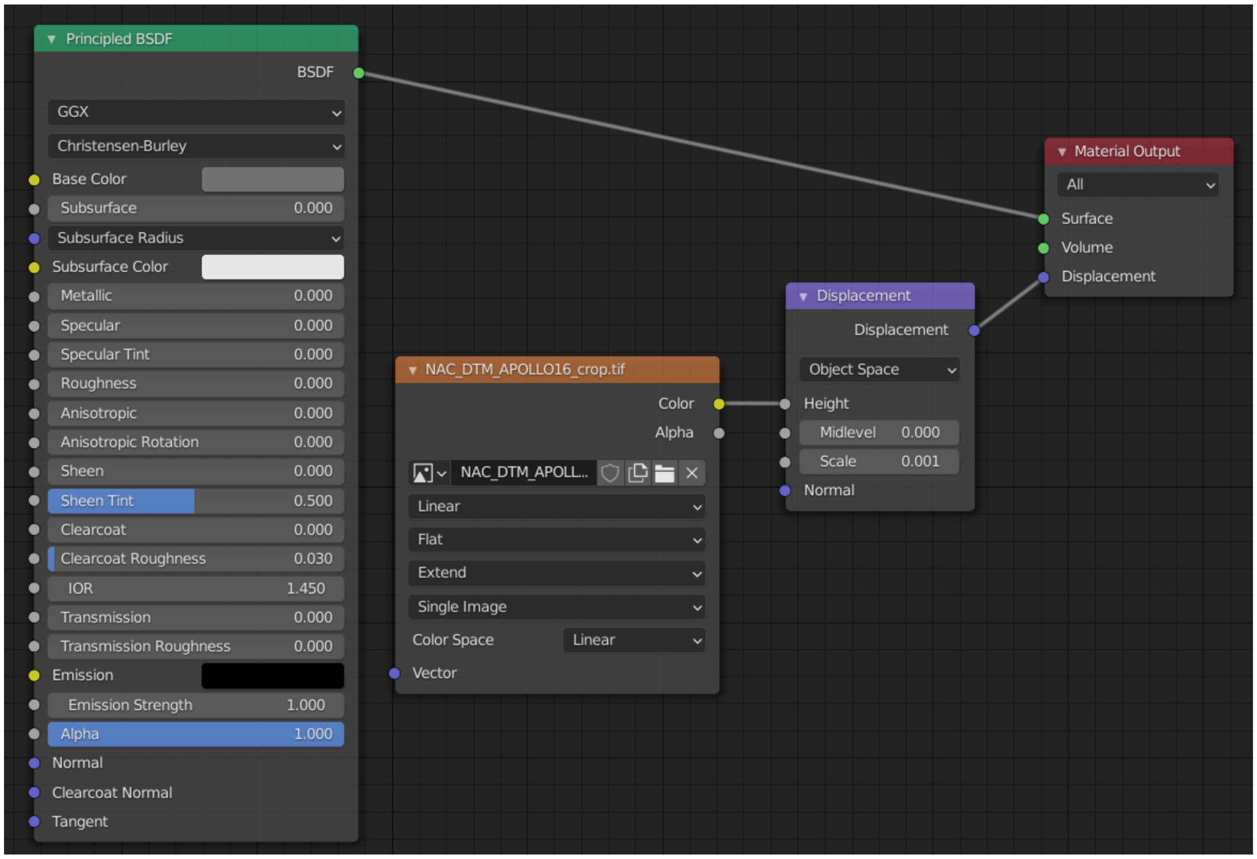


Fig. 1 Screenshot from Blender of the first file



Principled BSDF
BSDF
GGX
Christensen-Burley
Base Color
Subsurface 0.000
Subsurface Radius
Subsurface Color
Metallic 0.000
Specular 0.000
Specular Tint 0.000
Roughness 0.000
Anisotropic 0.000
Anisotropic Rotation 0.000
Sheen 0.000
Sheen Tint 0.500
Clearcoat 0.000
Clearcoat Roughness 0.030
IOR 1.450
Transmission 0.000
Transmission Roughness 0.000
Emission
Emission Strength 1.000
Alpha 1.000
Normal
Clearcoat Normal
Tangent

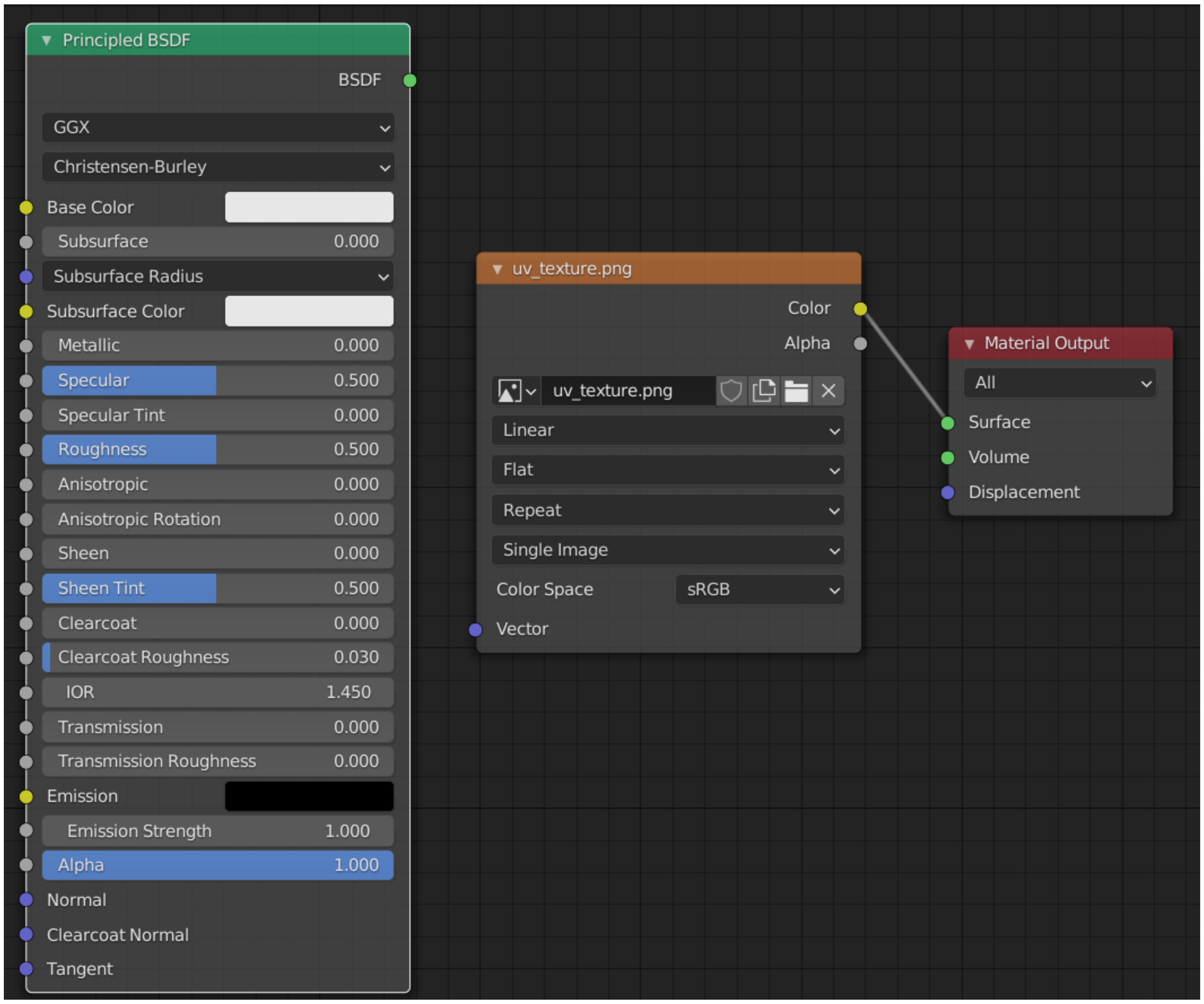
NAC_DTM_APOLLO16_crop.tif
Color
Alpha
NAC_DTM_APOLL...
Linear
Flat
Extend
Single Image
Color Space Linear
Vector

Displacement
Displacement
Object Space
Height
Midlevel 0.000
Scale 0.001
Normal

Material Output
All
Surface
Volume
Displacement

Fig. 2 Screenshot of the first file's Shader Editor screen for the plane

In order to speed up the rendering in the second file, another 6-by-6 meter plane was created, but instead a displace modifier was used to incorporate DTM height data. Like before, the strength of the modifer was set to 0.001 to properly scale the height. This way, the displacements would not have to be recalculated for every render as they would be for a displacement shader used in the first file, but rather only once at the start, which sped up rendering. The emission of the first Blender file was also “baked" as rendered from 1.25km from the surface, meaning the colors at different points along the surface were saved into a file. The bake was loaded into the Surface input in the Material Output shader in the plane's Shader Editor in Blender, so that the stored colors from the first file would be shown. To have enough detail in the surface, the plane was manually subdivided 200 times using the “Edge > Subdivide" button in its edit mode. Adaptive subdividing added more computation time as Blender needed to decide which areas to subdivide more in each render, and thus subdividing the surface manually was faster, albeit missing details when close to the surface. However, this wasn't a problem as there was not much detail in the landing site anyways, and any detail that was there would get lose in random noise added to the camera's output.


Principled BSDF

BSDF

GGX

Christensen-Burley

Base Color

Subsurface 0.000

Subsurface Radius

Subsurface Color

Metallic 0.000

Specular 0.500

Specular Tint 0.000

Roughness 0.500

Anisotropic 0.000

Anisotropic Rotation 0.000

Sheen 0.000

Sheen Tint 0.500

Clearcoat 0.000

Clearcoat Roughness 0.030

IOR 1.450

Transmission 0.000

Transmission Roughness 0.000

Emission

Emission Strength 1.000

Alpha 1.000

Normal

Clearcoat Normal

Tangent

uv_texture.png

Color

Alpha

uv_texture.png

Linear

Flat

Repeat

Single Image

Color Space sRGB

Vector

Material Output

All

Surface

Volume

Displacement

Fig. 3 Screenshot of the second file's Shader Editor screen for the plane

To create the third and final Blender file, the second file's plane was simply exported as a Wavefront file, and imported into another Blender file. This way, Blender would not even need to calculate the height displacements as they were stored directly in the file. The second Blender file, on the other hand, stored them by storing information on how to retrieve the height data. This did not provide a significant speed-up, as most of the computation time was already cut by baking the first file's plane's emission. This third file also used the Cycles renderer. Cycles renders images by simulating how light would bounce around the surface from the light source. However, in this case there was no need for a light source as the colors were already stored in the plane object. To take advantage of this, all the light path settings were set to their minimum values to make sure Cycles wasn't doing unnecessary calculations, such as simulating light bounces that didn't contribute to the lighting.

Computations for image generation and particle filtering were done using the University of Arizona's High Performance Computing systems, specifically, their Puma supercomputer. It was found that there was no significant difference between using the GPU or CPU Compute in the Device setting for Cycles, as previous optimizations were enough that using a GPU couldn't speed up rendering much further. As such, no GPU nodes were requested on Puma for training set generation, which allowed it to more easily allocate a computing node to run the code. In the end, it took around 0.07 seconds per render on Puma. Despite this speed, each of the 750 valid trajectories generated required 1000 images for the first 5 seconds and 100 for the next 65. This totaled to about 167 hours of computation time spent solely generating images for the particle filter.

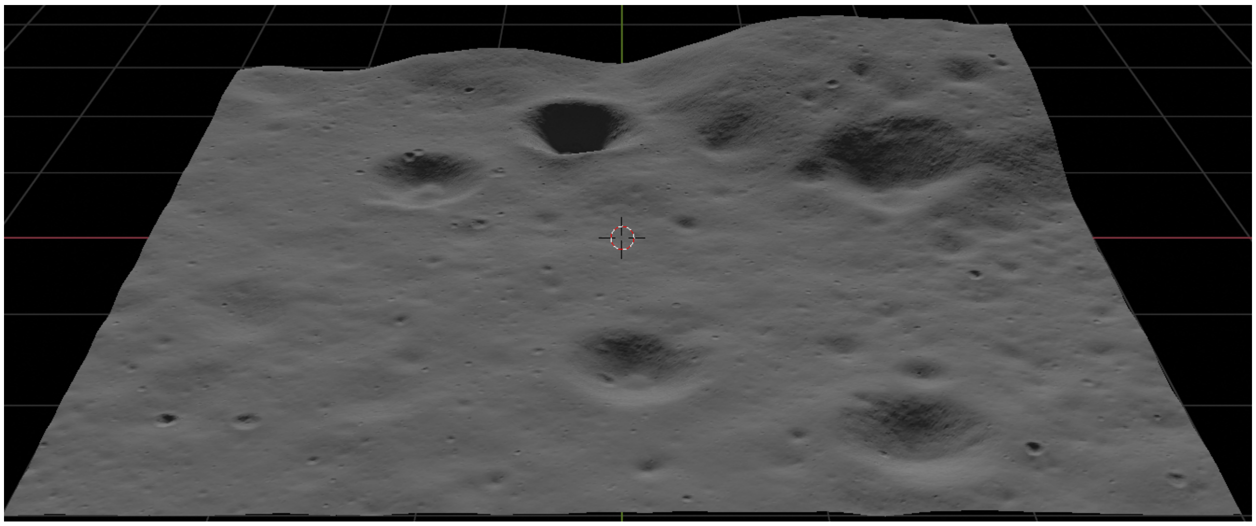


Fig. 4 Screenshot from Blender of the third file

E. Simplifications for RBPF

In developing this algorithm, multiple simplifications were made to reduce the complexity of the particle filter and to ensure data collection would not take too long. This is acceptable since the aim of this research is not to develop an algorithm that works in a high fidelity environment, but rather to show neural networks should be explored more for real-time Bayesian state estimation.

Starting with the system dynamics, the model used here failed to account for random noise in the specific impulse of the engine, and only accounted for random noise in the thrust. Additionally, in practice, any a lunar landing happens in three dimensions and also requires attitude control. Thus, a navigation system would have to estimate the full 3-dimensional position and velocity vectors, along with the lander's attitude and its velocity as it enters at an angle, instead of just vertical position and velocity used here. It may also be useful to account for deviations in the Moon's gravitational field which make it not uniform.

In addition, a navigation system used in practice would likely have access to more sensors to improve position and velocity estimate accuracy, such as an accelerometer and/or an altimeter. Real systems would also require a sensor such as a gyroscope for attitude estimation. The only sensor used in this research was a camera, and while estimates could be improved using more sensors, the control system was still able to land well without any more. In fact, adding in an accelerometer would have increased the complexity and computation time of the particle filter. The equation for acceleration, given in (1), depends on mass, so each UKF would need to have a second update step conditioned on the measurement. Additionally, there would be cross-correlation between the acceleration measurement and system dynamics noises to account for, as both depend on the random variable thrust.

F. Deep Learning Algorithm

Deep learning is able to approximate highly complex mathematical functions using layers of mathematical “neurons," each one producing a mathematical output based on the output of all the neurons in the previous layer, starting at the inputs. The simplest kind of layer is a fully-connected (FC) layer, sometimes also called a Dense layer, where each neuron's output is some pre-specified “activation function" of a weighted sum of all the previous layer's outputs, with the condition that the function must not be linear. Over time, the neural network is able to change the weights in this weighted sum to minimize a specified “loss" function, which is a measure of how badly the network performs.

The neural network developed in this study learned from a training set comprised of 700 trajectories generated by the RBPF and ZEM/ZEV. 50 other trajectories were used as “validation data," to make sure that the network's performance on the training data extended to data it had never seen before.

The deep learning approach used took advantage of research in Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). CNNs have “convolutional" layers which apply a mathematical convolution to smaller areas of an input image, of specified size and separated by a specified amount of pixels between these areas. The convolution has a “convolutional kernel," a matrix of the same size as the smaller areas. The output of the convolution is defined by multiply corresponding values in the smaller areas of the picture and convolutional kernel, and adding these products together. This is able to efficiently detect difference between pixels near each other, which helps with image processing. The convolutional kernel is one of the weights that the network learns over time. Some CNNs also have some sort of “pooling" layer which reduces the amount of data they produce, but given the small size of images, pooling layers were not added to avoid potentially losing important information. For a more thorough description of convolutional neural networks, see Wu [18]

RNNs provide a framework for a neural network to learn to process sequences of data, separated in time. In an RNN, some “hidden state" is stored, and is used in conjunction with the sequence of inputs to produce an output. Each input is combined with the hidden state in a mathematical function to create a new hidden state for the next input. At the end of this chain, the final output is usually taken to be the final hidden state.

In this case, the data was a list of images and thrusts, each one being 1 simulated second apart. A Long-Short-Term-Memory cell was used, which stores two different hidden state vectors: one for short-term data, and another for long-term data. The input to the layer gets combined with the short-term and long-term states through a series of mathematical functions to create the output. In the process, the long-term state goes through a “forget gate," which multiplies terms in the long-term state by numbers generated by an FC layer of the input data. This lets it stop storing information that is no longer relevant as time progresses and it analyzes data later on in the sequence of inputs. New information, calculated as the product of two FC layers of the input, is added after the forget gate to produce the long-term data used for the next input in time. The next time step’s short-term state is calculated by multiplying the output of another FC layer of the input by the long-term state, after being passed through the mathematical “hyperbolic tangent" function. A more complete description of LSTMs, with an application in space landings can be found in Furfaro et al. [10].

Various architectures, numbers of neurons, loss functions, and activation functions in each layer were tested, and the network described below tended to have the best performance. It first took an an input that was a sequence of thrusts applied to the lander and images taken from the lander in the next time step. The sequence consisted of images and thrusts from the 1 through n-th simulated seconds of the landing, where n is the amount of seconds into the landing that the neural network is called. For example, to estimate the state of the lander at 35 seconds into the landing, images from 1 second to 35 seconds and thrusts from 0 seconds (the start) to 34 seconds into the landing were incorporated into the input sequence. The neural network used processed the sequence of images with 3 convolutional layers, and then passed the output of the convolution into a FC layer. The output of the FC layer for a given image was then concatenated to the corresponding thrust applied before the image was taken. This new concatenated output was passed into an LSTM layer that worked with the sequence of thrusts and processed images. The output of the LSTM, was fed into 4 consecutive FC layers, and the output of the last one was taken to be the estimated position, velocity, and mass.

The first two convolutional layers had 32 different kernels, a kernel size of 3x3, and applied kernels separated by 2 pixels in the image. The third convolutional layer had 64 kernels, with a size of 2x2, and applied the kernels separated by 4 pixels in the image. The FC layer following the convolutions used 128 neurons, and so did the 3 FC layers immediately following the LSTM cell. The LSTM cell used hidden states composed of 256 different values. At the end, a FC layer with 3 neurons was used to give the final outputs.

The most successful activation function tested was the Leaky Rectified Linear Unit (LeakyReLU). This function takes an input x and returns x when it is positive, and a parameter α multiplied by x when it is negative. In this case, $\alpha = 0.3$ was used. LeakyReLU is a very simple nonlinear function, which makes it easy to use to train a neural network, and variants of ReLU have seen a lot of success in deep neural networks.

One other thing done to improve accuracy was to add dropout of 0.5 to the LSTM cell. This meant that every time the LSTM cell processed input data to train on, about half of its neurons in the FC layers were removed. Dropout has been shown to be successful in increasing accuracy of networks, as it forces neurons to learn more useful weights.

The training input and output data were also rescaled so that the minimum and maximum values were 0 and 1 respectively, for each pixel in all images, each thrust, and each value of position, velocity, and mass. This is standard practice so that the network has to work less to learn the proper scaling of data.

The loss function used to determine how badly the network was approximating the position, velocity, and mass of the lander is given below:

$$L = \sqrt{\frac{1}{70}\sum_{t_{go}=0}^{70-1}\left(\frac{6}{t_{go}^2}\cdot y_T - \frac{2}{t_{go}}\cdot v_T - g\right)^2 e_m^2 + \left(\frac{36}{t_{go}^4}e_y^2 + \frac{16}{t_{go}^2}e_v^2\right)e_m^2 + \left(\frac{6}{t_{go}^2}\bar{y} - \frac{2}{t_{go}}\bar{v}\right)^2 e_m^2 + \bar{m}^2\left(\frac{36}{t_{go}^4}e_y^2 + \frac{16}{t_{go}^2}e_v^2\right)}$$

where $y_T = 50m$ is the target position, $v_T = -2m/s$ is the target velocity, $\bar{y}$, $\bar{v}$, and $\bar{m}$ are the real position, velocity, and mass as predicted by the particle filter, and e_m, e_v, and e_m is the error between the particle filter and neural network prediction. This particular loss function was inspired by considering how much variance would be in the commanded thrust from the ZEM/ZEV control law after a filter approximated the state variables. Namely, that variance would be given by:

$$\begin{aligned}\text{Var}(T) = \frac{1}{70}\sum_{t_{go}=0}^{70-1}&\left(\frac{6}{t_{go}^2}\cdot y_T - \frac{2}{t_{go}}\cdot v_T - g\right)^2 \text{Var}(m)+\\ &\left(\frac{36}{t_{go}^4}\text{Var}(y) + \frac{16}{t_{go}^2}\text{Var}(v)\right)\text{Var}(m)+\\ &\left(\frac{6}{t_{go}^2}\bar{y} - \frac{2}{t_{go}}\bar{v}\right)^2 \text{Var}(m)+\\ &\bar{m}^2\left(\frac{36}{t_{go}^4}\text{Var}(y) + \frac{16}{t_{go}^2}\text{Var}(v)\right)\end{aligned}$$

where $\text{Var}(x)$ represents the variance of x. Notice that the variance is averaged across all times in the landing, which is done because constants of proportionality in ZEM/ZEV vary over time. In principle, it might be desirable to minimize this variance, to ensure the thrust profile closely follows what it would if all states were known with exact certainty.

To minimize the cost function, the Adam optimizer was used. A common optimizer, Stochastic Gradient Descent (SGD), changes all the weights of the network proportional to the "gradient" of the loss. That is, proportional to how much the loss would change if each weight was changed by a small amount. Over time, SGD tends to approach a minimum value in the loss. Adam is a computationally efficient improvement of SGD described in Kingma and Ba [19]. Adam uses a few manually tunable parameters, but the recommended parameters $\eta = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 0.0000001$ worked well for this application.

In the training process, it was noticed that data near the start and end of the trajectory tended to have abnormally large loss values. This was compensated for by weighing how much each data point in each trajectory contributed to the loss depending on how far away it was from the middle time $t = 35$. Specifically, the first and last simulated times weighed 5 times more than the ones at $t = 34$ and $t = 35$, and in between the weight scaled linearly.

G. Computations

The Rao-Blackwellized Particle Filter and image generation were run in the same Python script, which was loaded alongside the third Blender file. Blender was called from the command line on Puma, and there it was specified to load the Python script and to run Blender in the background, without its usual graphical interface. There was no way to easily display graphics on Puma outside of the command line, and running Blender in the background also tends to make it render faster as it only has to generate images without displaying them. Blender could not be run as-is in Puma and instead had to be wrapped inside of a Singularity container. Singularity provides a way to package together any needed programs and code for running jobs on HPC systems. Blender also comes with its own version of Python, so the

SciPy and FilterPy Python libraries had to be installed for statistical functions and Kalman filtering respectively into Blender's Python.

III. Preliminary Results

A representative sample trajectory in the training data is shown below.

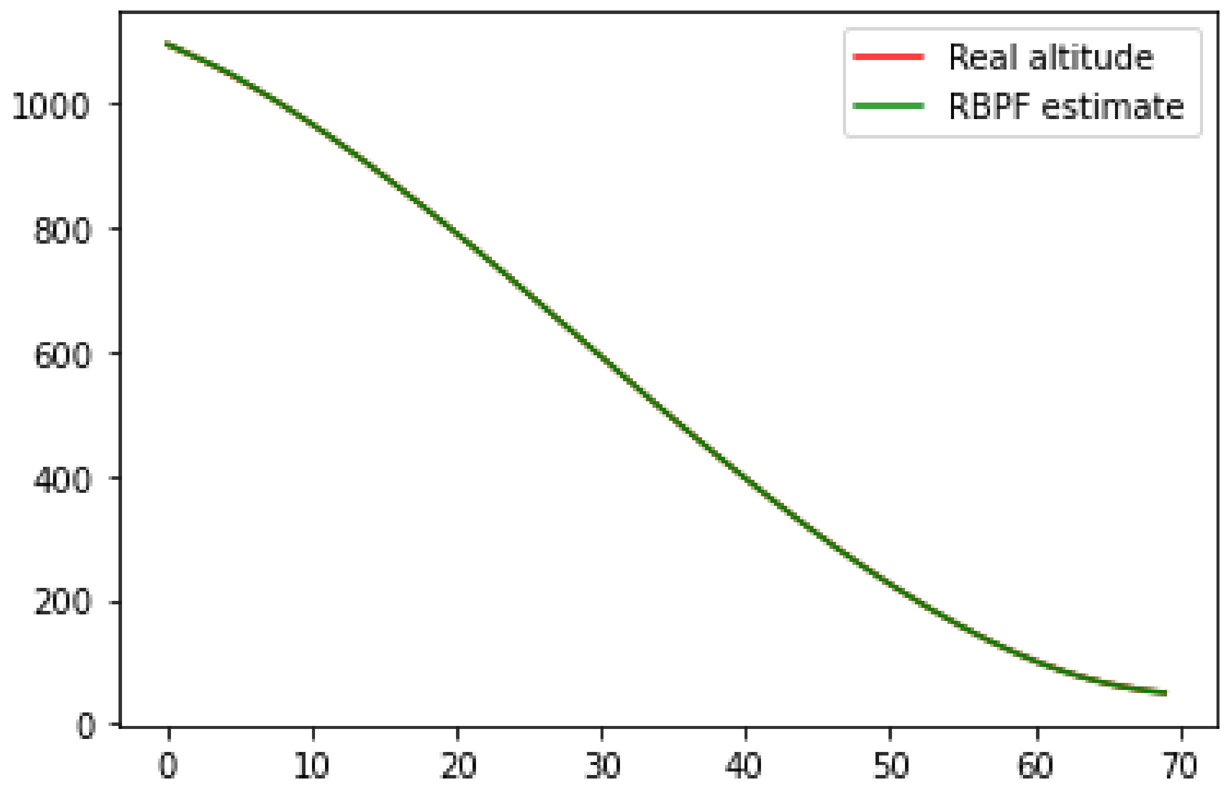


Fig. 5 Altitude over time for a RBPF trajectory

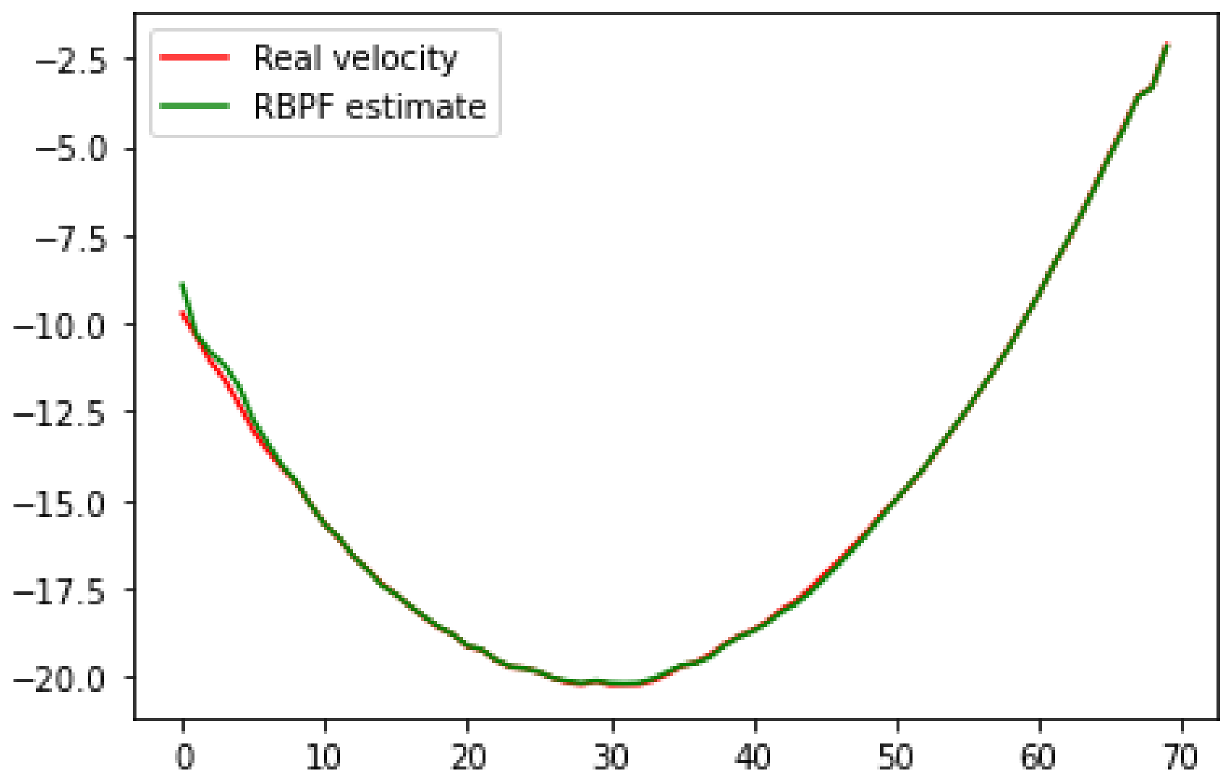


Fig. 6 Velocity over time for a RBPF trajectory

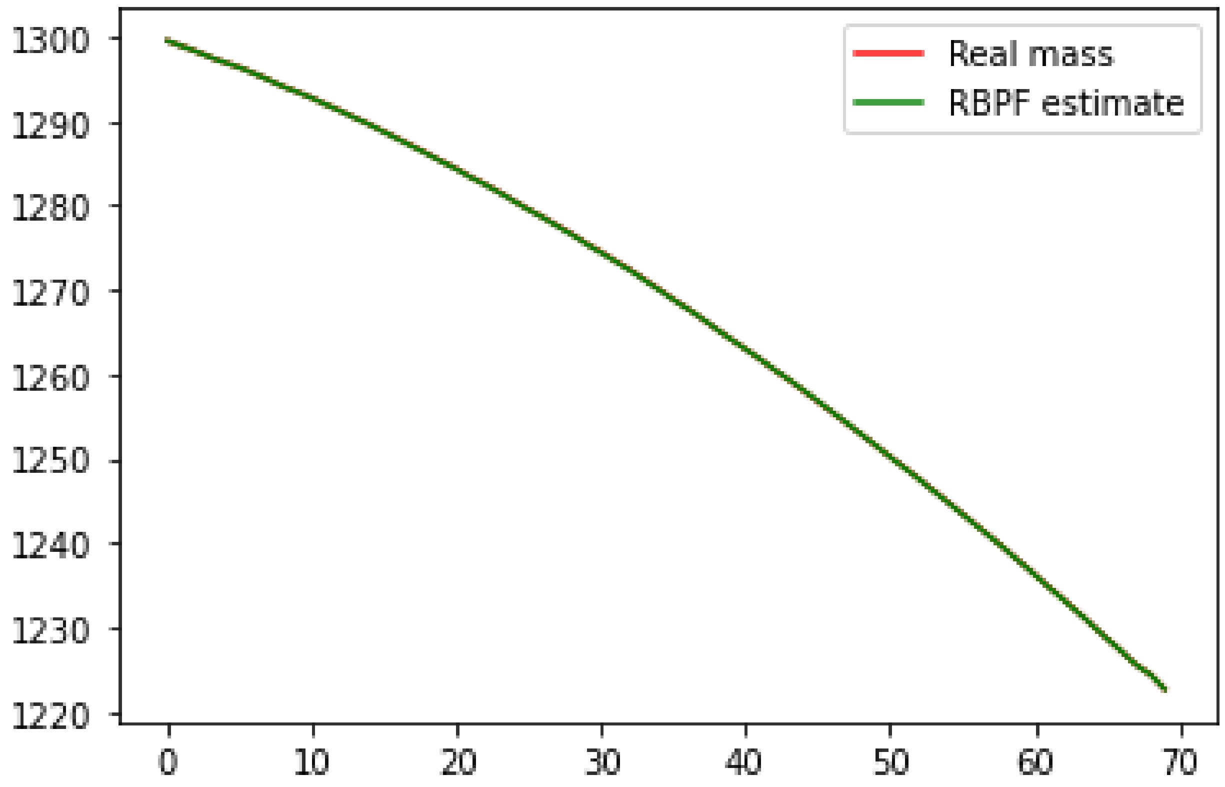


Fig. 7 Mass over time for a RBPF trajectory

Clearly, the RBPF performed very well, and from the collected data, it was able to successfully get close to the targeted position and velocity in all of the valid training trajectories. When looking at the difference of the data real states and the estimates, along with estimates of standard deviation, it becomes clearer that there is a lot of noise added by the particle filter's lack of particles however:

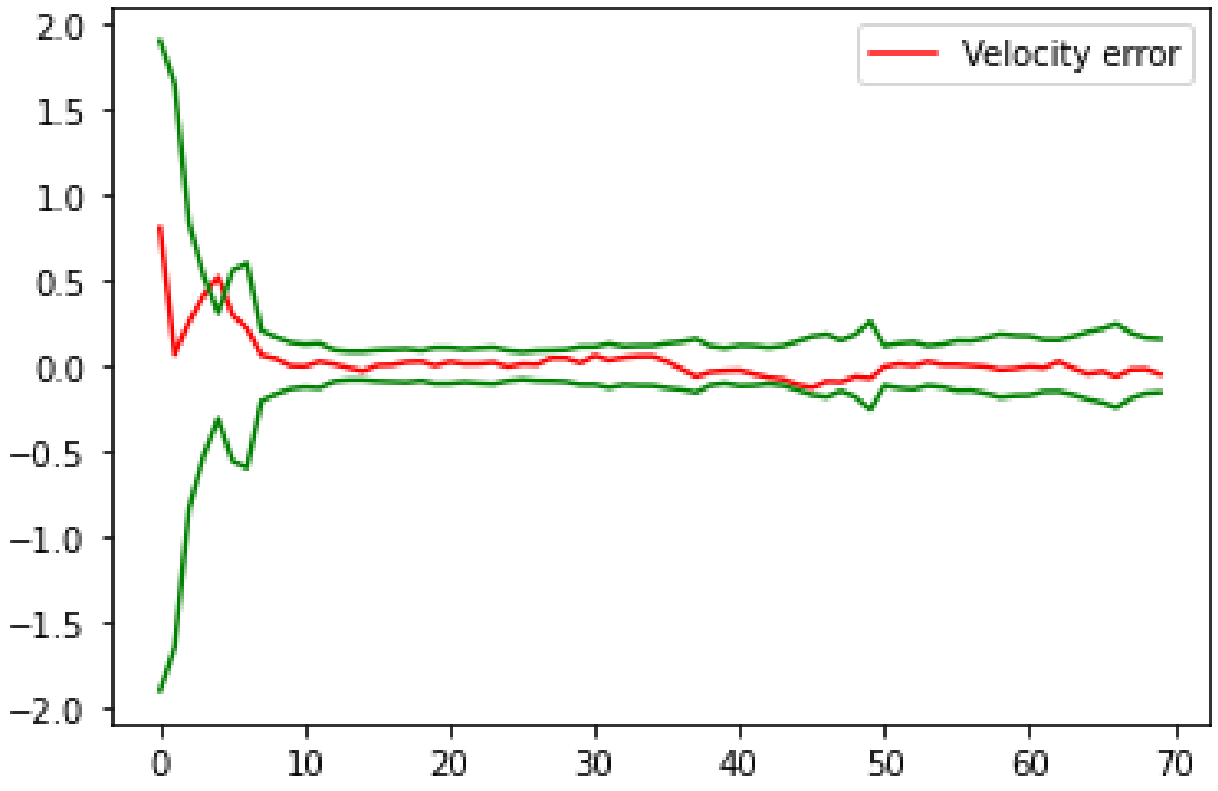


Fig. 8 Error in position over time for a RBPF trajectory

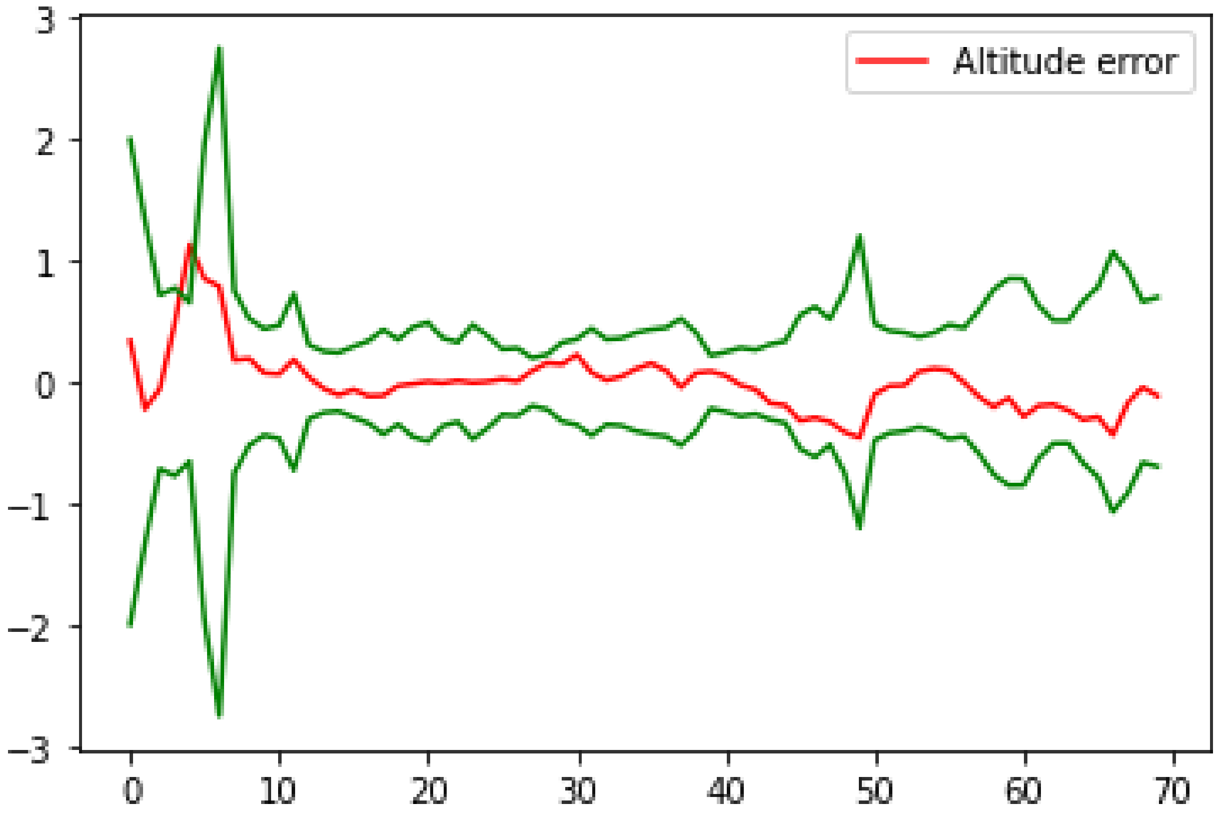


Fig. 9 Error in velocity over time for a RBPF trajectory

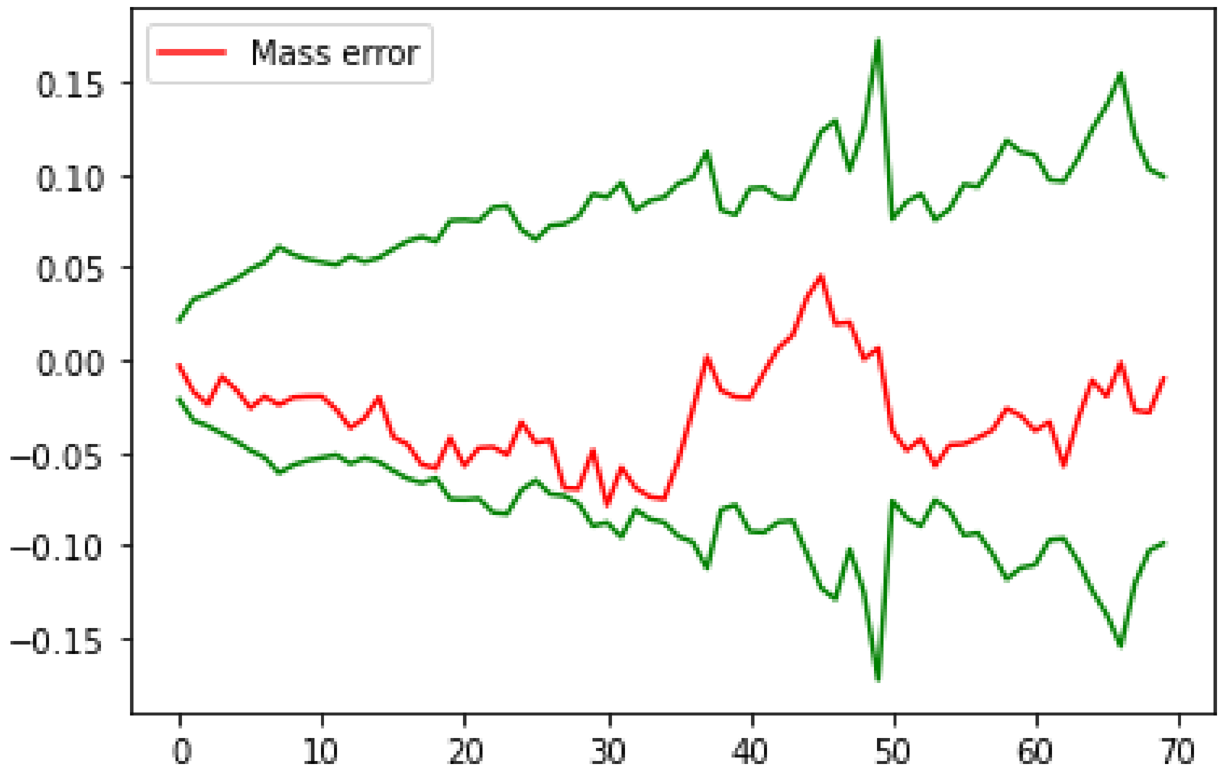


Fig. 10 Error in mass over time for a RBPF trajectory

Below are graphs of loss over time as the network trained, for both training data and validation data. Both graphs are in the log-scale in order to highlight how it progressed over time, as the initial loss would've been high enough that all detail in losses after would be lost. There is a noticeable increase in the graphs when it was switched to weigh times near the end and start more than those in the middle, which is to be expected as changing the weighing makes them no longer comparable.

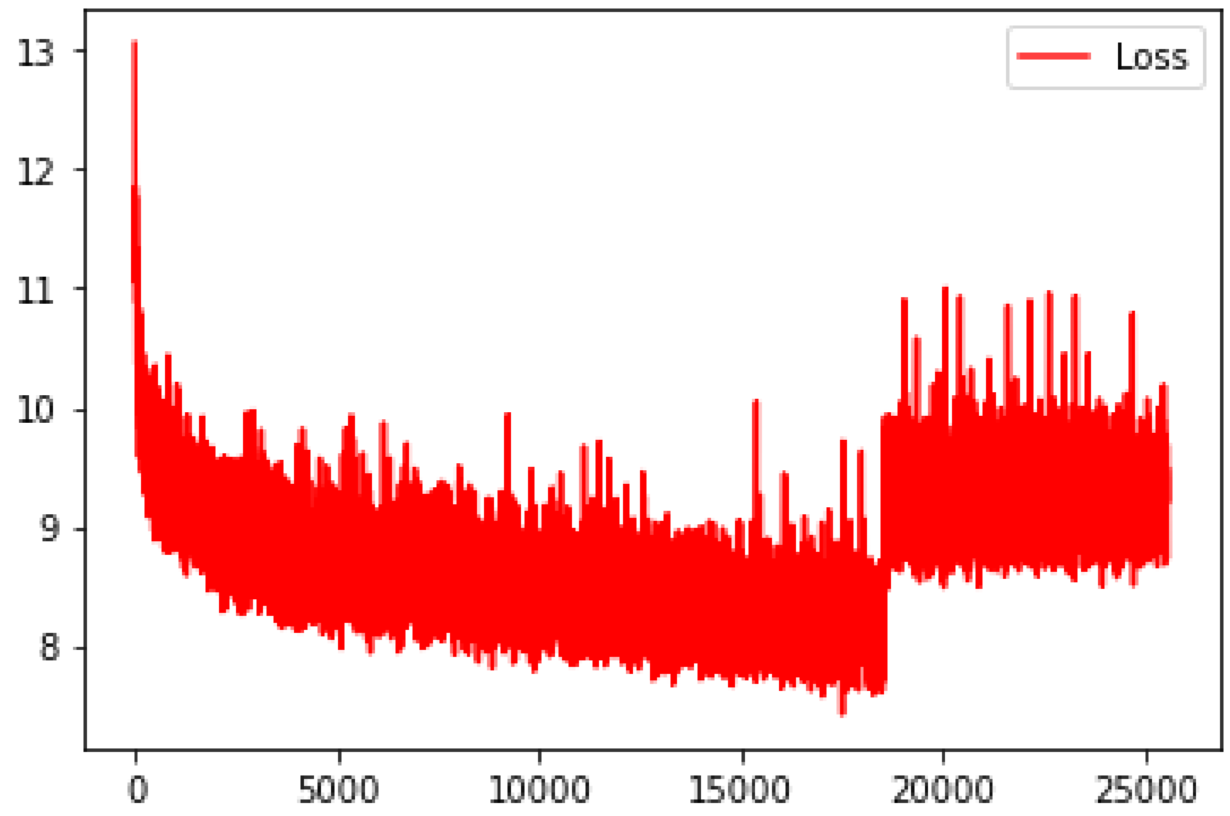


Fig. 11 Graph of training loss of the network over time

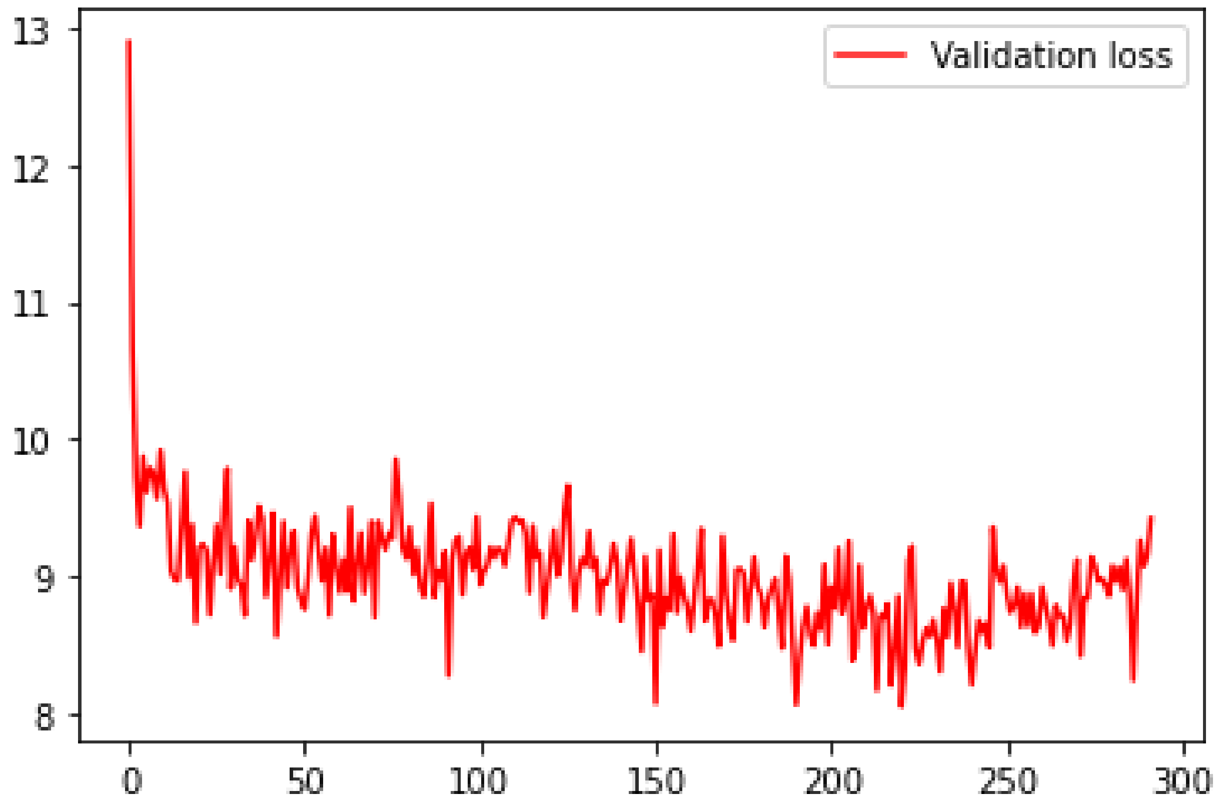


Fig. 12 Graph of validation loss of the network over time

With the network trained, the particle filter in the original code was replaced by the network. Below is a representative sample of the performance:

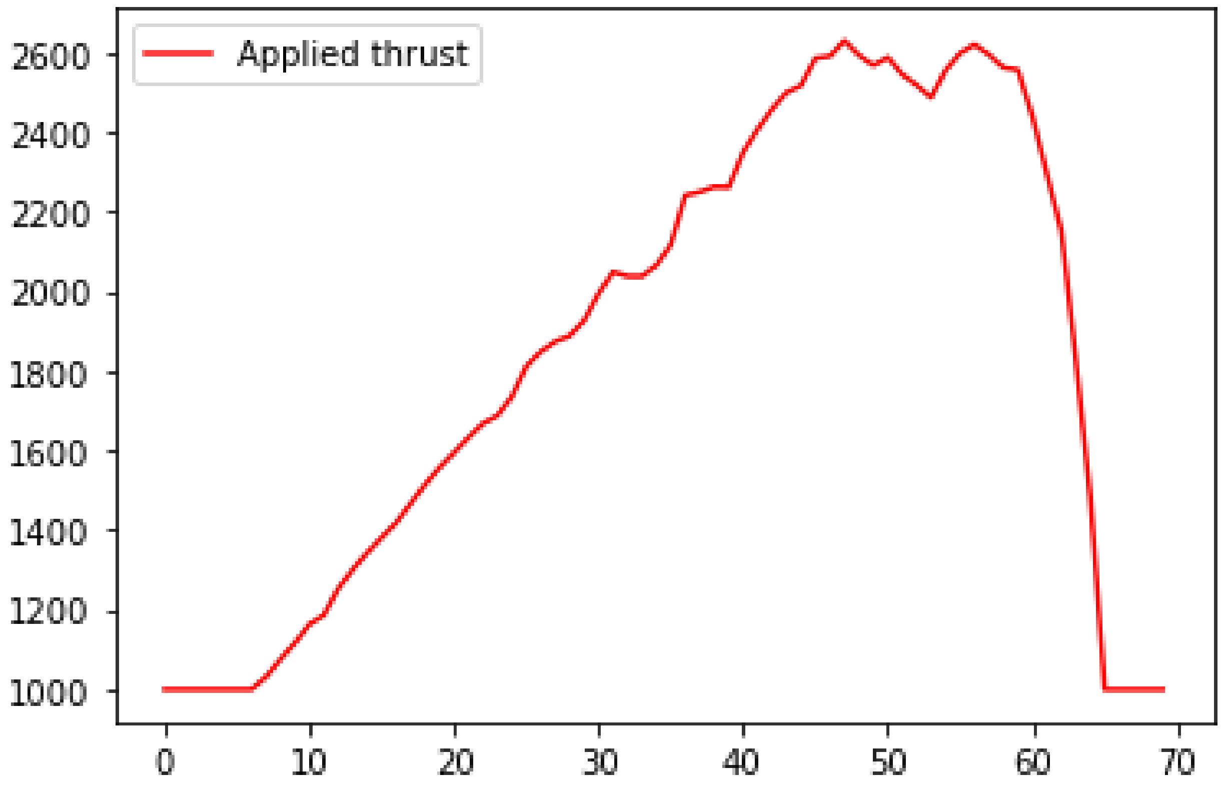


Fig. 13 Graph of thrust over time when using neural network estimation

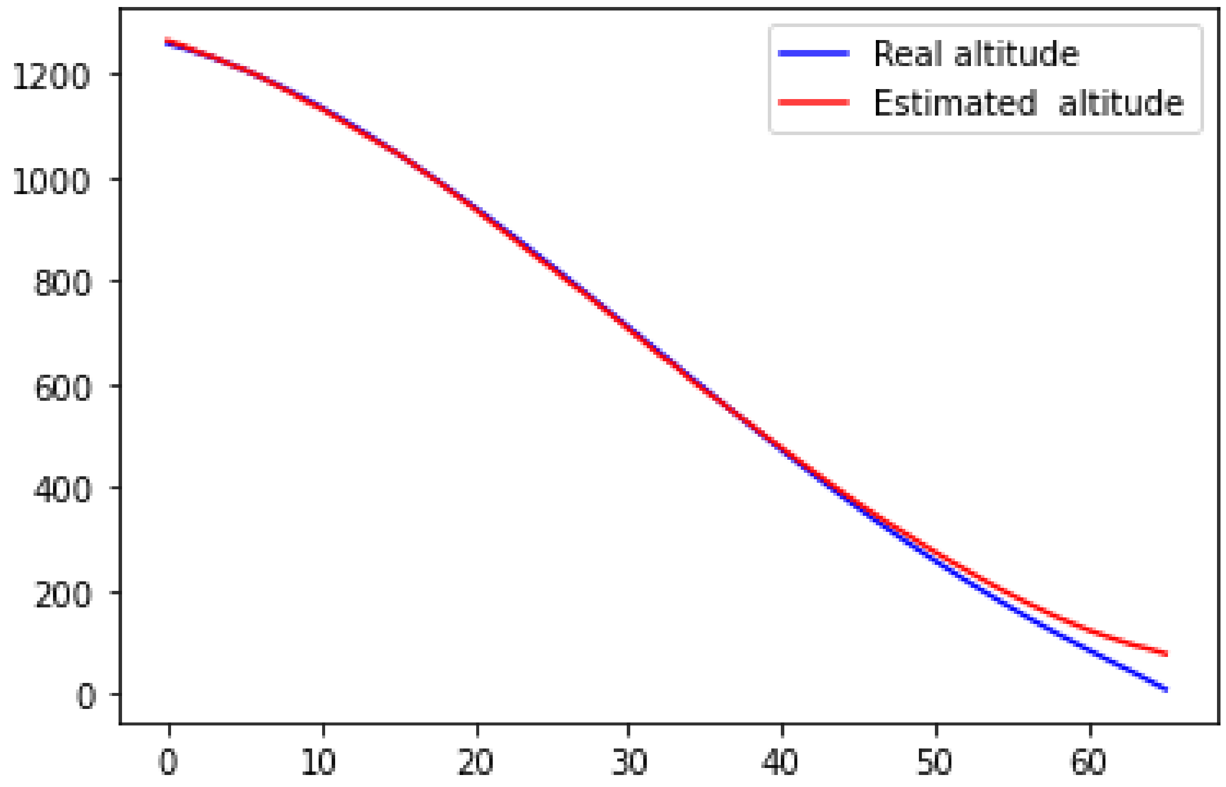


Fig. 14 Graph of position over time when using neural network estimation

These graphs show that the thrust unexpecedly decreases when compared to the RBPF thrust profile, and this change caused the neural network to be unable to accurately estimation position anymore as it diverged slowly. The real altitude descents below zero meaning the lander crashed, while the estimated altitude does not.

One fix that was found for this was to modify the control law so that once the thrust went above the minimum 1kN thrust, it would always increase by at least 25 Newtons. When using this new control law, the neural network was able to more frequently see amounts of thrust it was used to, and was able to effectively stay above the surface for all 10 test trials. Below is one of these trajectories:

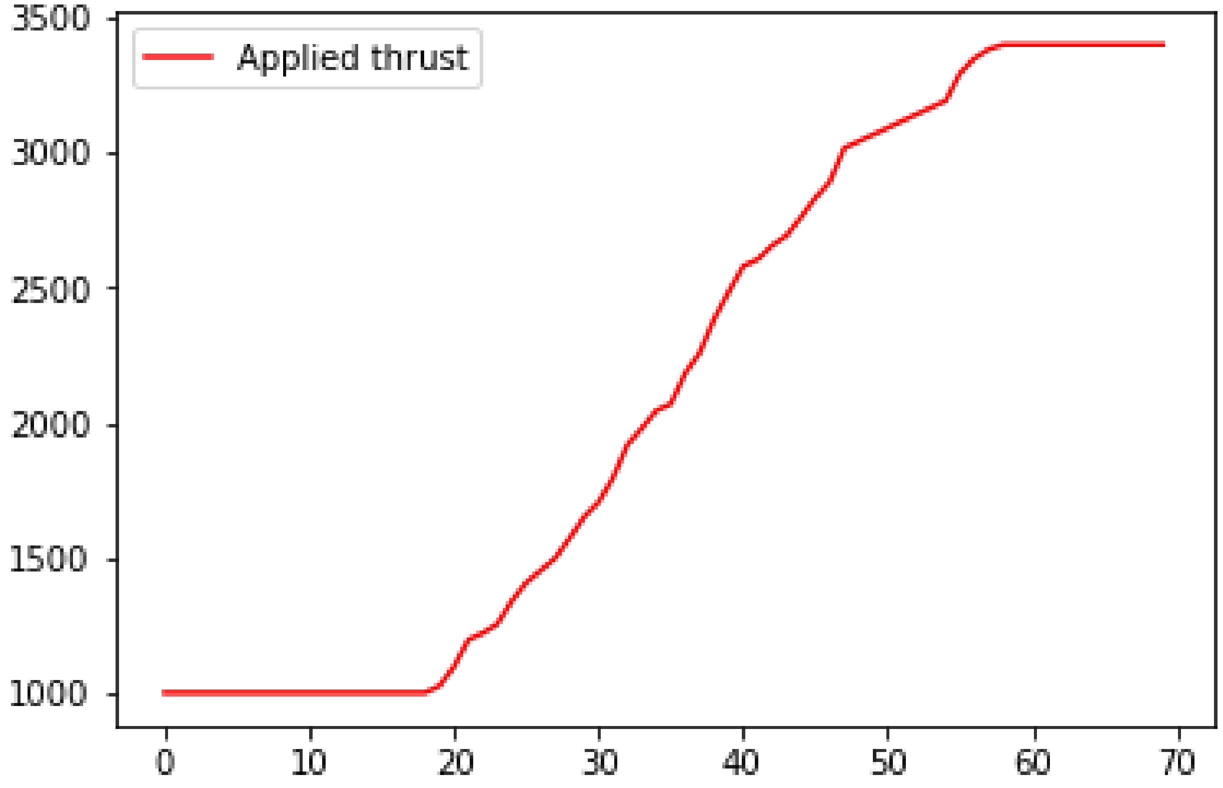


Fig. 15 Graph of thrust over time when using neural network estimation with corrected control law

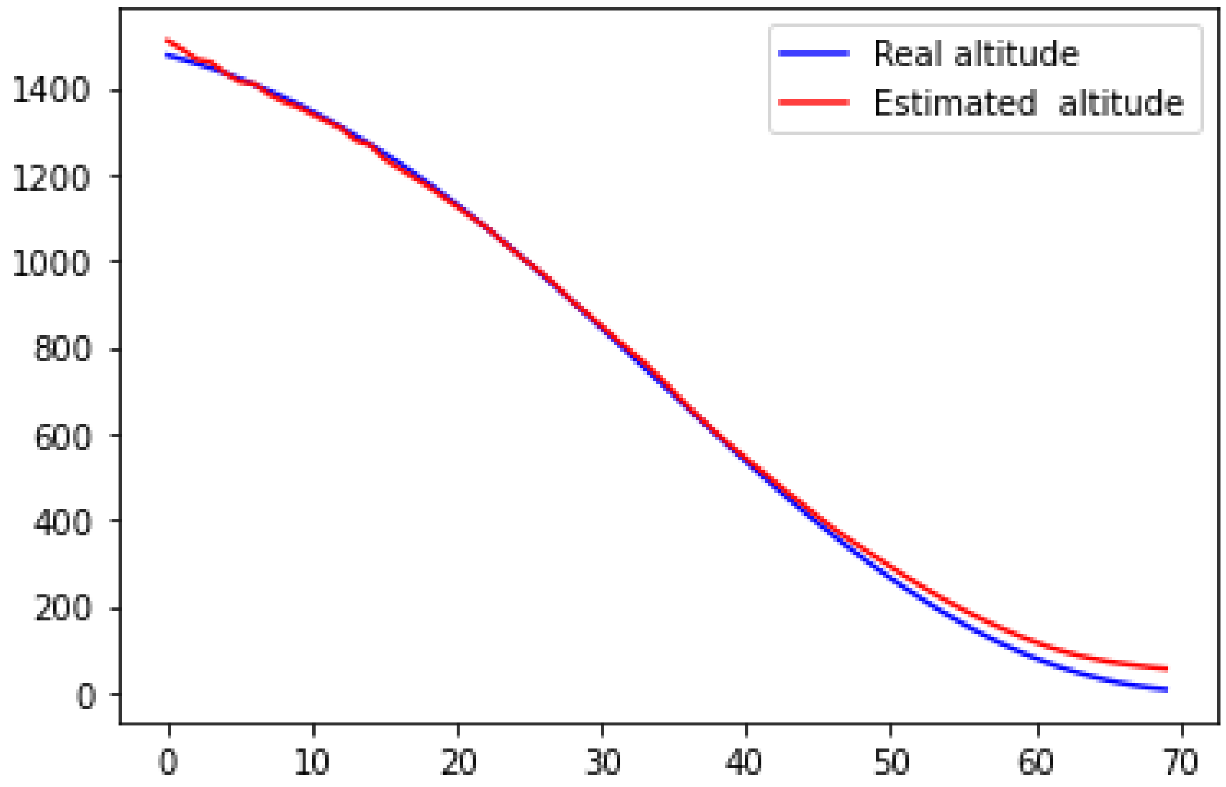


Fig. 16 Graph of position over time when using neural network estimation with corrected control law

In this case, the correction to the control law can be seen at around 40 seconds, when there is a constant slope in the applied thrust.

IV. Improvements to Be Made

This project started as a school assignment for the first author's Research class at BASIS Tucson North High School, and as such it had a tight schedule and the results obtained so far are only preliminary. Now that the assignment is over, the authors will spend more time improving the results obtained in this paper to make them more significant. Comparing the results of the RBPF estimation and the neural network estimation reveal that the NN is not nearly as good, although it isn't as computationally intense. This is especially seen towards the end of the trajectory, despise efforts to mitigate it by weighing those times more heavily. A well trained network shouldn't need to have a modified control law in order to get a similar performance to the RBPF's. As such, the authors will be continuing to improve the results. For example, more network architectures will be tested. Particularly, the amount of convolutional layers, along with their hyperparameters was not explored as well as the rest of the architecture. The loss function will also be changed, as this particularly odd function was only chosen when standard ones such as a weighted mean square error didn't seem to work well. The authors speculate those didn't work well because of the network architecture.

In addition, the authors will increase the quality of the training data in a few ways ways. Firstly, they will increase the number of particles used in order to smooth out noise in the particle filter estimates. Secondly, they may also cut down on the amount of simplifications in the image generation process in order to demonstrate the algorithm could work with more realistic images. They may also incorporate other sensors into the final product to show a neural network could handle more than optical measurements. Additionally, because generating images took a long amount of time, given how many particles were needed, the authors will also explore using the same concepts for a neural network trained on a RBPF that used a LIDAR flash as its measurement input. LIDAR flashes are easier to simulate compared to images, so substantially more particles can be used with the same amount of time for generating training data. It's possible that the convolutional layers will also be able to process LIDAR flash measurements.

If the authors have time, they may also explore adding a second dimension to the landing to demonstrate neural networks would be able to perform Bayesian state estimation with more than just a vertical dimension. However, this would require an order of magnitude more particles and possibly more training time. If the authors have time, they may also explore adding a second dimension to the landing to demonstrate neural networks would be able to perform Bayesian state estimation with more than just a vertical dimension.If the authors have time, they may also explore adding a second dimension to the landing to demonstrate neural networks would be able to perform Bayesian state estimation with more than just a vertical dimension.

Also learning this covariance matrix was initially explored, but that was abandoned in order to have more time to train a network that could at least learn the mean state. However, now that they have more time, learning the covariance can be explored again. This would be done by learning the diagonal and lower triangular matrices in the Cholesky decomposition of the covariance. This decomposition would guarantee positive semi-definiteness of the outputted covariance, which has been used in other studies for covariance estimation, such as Liu et al. [20]

Acknowledgments

This material is based upon High Performance Computing (HPC) resources supported by the University of Arizona TRIF, UITS, and Research, Innovation, and Impact (RII) and maintained by the UArizona Research Technologies department.

References

[1] Johnson, A. E., and Montgomery, J. F., "Overview of terrain relative navigation approaches for precise lunar landing," *2008 IEEE Aerospace Conference*, IEEE, 2008, pp. 1–10.

[2] Johnson, A., and Ivanov, T., "Analysis and testing of a lidar-based approach to terrain relative navigation for precise lunar landing," *AIAA Guidance, Navigation, and Control Conference*, 2011, p. 6578.

[3] Gaudet, B., Linares, R., and Furfaro, R., "Deep reinforcement learning for six degree-of-freedom planetary landing," *Advances in Space Research*, Vol. 65, No. 7, 2020, pp. 1723–1741.

[4] Johnson, A., and Matthies, L., "Precise Image-Based Motion Estimation for Autonomous Small Body Exploration," *Artificial Intelligence, Robotics and Automation in Space*, Vol. 440, 1999, p. 627.

[5] Gaudet, B., and Furfaro, R., "A navigation scheme for pinpoint mars landing using radar altimetry, a digital terrain model, and a particle filter," *AAS/AIAA Astrodynamics Specialist Conference, Hilton Head Island, SC, USA*, 2013, pp. 2537–2556.

[6] Roumeliotis, S. I., Johnson, A. E., and Montgomery, J. F., "Augmenting inertial navigation with image-based motion estimation," *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02Ch37292)*, Vol. 4, IEEE, 2002, pp. 4326–4333.

[7] Bilodeau, V. S., Clerc, S., Drai, R., and de Lafontaine, J., "Optical navigation system for pin-point lunar landing," *IFAC Proceedings Volumes*, Vol. 47, No. 3, 2014, pp. 10535–10542.

[8] Izzo, D., Märtens, M., and Pan, B., "A survey on artificial intelligence trends in spacecraft guidance dynamics and control," *Astrodynamics*, 2019, pp. 1–13.

[9] Kothari, V., Liberis, E., and Lane, N. D., "The Final Frontier: Deep Learning in Space," *Proceedings of the 21st International Workshop on Mobile Computing Systems and Applications*, 2020, pp. 45–49.

[10] Furfaro, R., Bloise, I., Orlandelli, M., Di Lizia, P., Topputo, F., Linares, R., et al., "Deep learning for autonomous lunar landing," *2018 AAS/AIAA Astrodynamics Specialist Conference*, 2018, pp. 1–22.

[11] Campbell, T., Furfaro, R., Linares, R., and Gaylor, D., "A deep learning approach for optical autonomous planetary relative terrain navigation," *27th AAS/AIAA Space Flight Mechanics Meeting*, 2017, pp. 3293–3302.

[12] Proença, P. F., and Gao, Y., "Deep learning for spacecraft pose estimation from photorealistic rendering," *2020 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2020, pp. 6007–6013.

[13] Roberts, A., "Modify the Improved Euler scheme to integrate stochastic differential equations," *arXiv preprint arXiv:1210.0933*, 2012.

[14] Douc, R., and Cappé, O., "Comparison of resampling schemes for particle filtering," *ISPA 2005. Proceedings of the 4th International Symposium on Image and Signal Processing and Analysis, 2005.*, IEEE, 2005, pp. 64–69.

[15] Schon, T., Gustafsson, F., and Nordlund, P.-J., "Marginalized particle filters for mixed linear/nonlinear state-space models," *IEEE Transactions on signal processing*, Vol. 53, No. 7, 2005, pp. 2279–2289.

[16] Chang, G., "Marginal unscented Kalman filter for cross-correlated process and observation noise at the same epoch," *IET Radar, Sonar & Navigation*, Vol. 8, No. 1, 2014, pp. 54–64.

[17] Hawkins, M., Guo, Y., and Wie, B., "ZEM/ZEV feedback guidance application to fuel-efficient orbital maneuvers around an irregular-shaped asteroid," *AIAA Guidance, Navigation, and Control Conference*, 2012, p. 5045.

[18] Wu, J., "Introduction to convolutional neural networks," *National Key Lab for Novel Software Technology. Nanjing University. China*, Vol. 5, 2017, p. 23.

[19] Kingma, D. P., and Ba, J., "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[20] Liu, K., Ok, K., Vega-Brown, W., and Roy, N., "Deep inference for covariance estimation: Learning gaussian noise models for state estimation," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, 2018, pp. 1436–1443.