# A Novel Approach to Calculating Orbital Mechanics with Machine Learning

Rylan Andrews

Satellite High School

## Table of Contents

## Abstract

The *n*-body problem, which describes the motion of three or more gravitational bodies around each other, has perplexed scientists for centuries. Notably, no analytical solution exists for it; Newton's Law of Universal Gravitational Motion is not directly applicable. Instead, a time-consuming strategy known as numerical integration must be employed. This study explored machine learning as an alternative. Machine learning has been shown to be effective at solving partial differential equations, such as those used in *n*-body problems.

The project focused on solving four-body problems. Training the machine learning model required a dataset; as no datasets of solved problems exist, one was generated with a classical solution-based program. The machine learning model was trained on that dataset. To evaluate accuracy, the model was incorporated into a program that compared the machine learning solution to the classical solution. The program was then able to calculate the divergence between the predicted trajectories of each solution. This divergence was considered to be the best measure of accuracy of the machine learning model.

Results revealed that as the simulation progresses, the two solutions diverged, although the model did showcase evidence of learning. No statistically significant difference was found between the solutions. A physical analysis was also performed to assess whether the model recognized the concept of a system center of mass, and the model was shown to be successful. In conclusion, while the model would benefit from further optimization to improve accuracy, machine learning is a potential alternative to classical solutions for *n*-body problems.

## Introduction

Determining a way to estimate the motions of celestial bodies has plagued mathematicians and scientists for centuries. Predicting the motion of gravitational bodies can provide insight into such phenomena as the seasons, trajectories for spacecraft bound for other planets, and the motion of asteroids. As humanity turns its eyes to the skies, knowledge of the motions of gravitational bodies will only become more important.

For centuries, scientists and mathematicians have studied the two-body problem. The two-body problem deals with the movement of two gravitationally relevant bodies around each other, and can be solved using defined equations. Johannes Kepler first proposed the two-body problem in 1609, and Isaac Newton discovered an analytical solution for it in 1687 (Wolfram, 2002). Trajectories can be precisely predicted infinitely into the future. However, there exist more than two gravitationally relevant bodies in the universe, so in most cases, the *n*-body problem is more relevant. The *n*-body problem describes the motion of *n* number of gravitationally relevant bodies in motion around each other. Unlike the two-body problem, there exists no analytical solution to this problem.

There is no definitive solution partly because the *n*-body problem describes a chaotic system. Chaotic systems describe a scenario in which slight changes in initial state can cause drastic changes later on in the scenario (Halpern, 2018). Some examples include the weather and rolling dice. In the case of the *n*-body problem, changes in the masses, initial positions, or initial velocities of the gravitational bodies (planets, stars, etc) will result in large variations as the simulation progresses.

## Literature Review

The *n*-body problem is generally simplified into the three-body problem, which doesn't have an analytical solution either (except in a few isolated scenarios). The three-body problem focuses on the motion of three gravitationally relevant bodies around each other (The Editors of Encyclopaedia Britannica, 2016). Approximate solutions to the three-body problem have been proposed, but these either suffer in accuracy or require significant amounts of computing power, or both. Some strategies include disregarding the smallest body in calculations, and solving it as a two-body problem. Other strategies leverage computers, and include series expansions, which involve thousands of algebraic variables, or numerical integration, which involves running hundreds of calculations for miniscule increments of time.

In numerical integration, the approach employed by this study, the forces on each body in the system are calculated, and this force is then assumed to be constant for a small period of time. The force is attained using Newton's Law of Gravitation, which can be distilled into a form usable in an ordinary differential equation solver. The vector form of the equation (figure 1a) is used in lieu of the standard form. This allows the unit vector to be broken into the magnitude and direction, resulting in equation b (figure 1). The force can be broken into $F = ma$, using Newton's Second Law, and acceleration can be calculated with the first derivative of velocity $r$ (figure 1c). The mass of the body whose motion is being calculated can be canceled out from both sides. Using summation notation, this equation can be extended out for as many gravitational bodies as is necessary (figure 1d). An equation must be solved for each body in the problem (Deshmukh, 2019). The motion of the body is then calculated based on the force over the small time interval. This process is repeated, stepping forward in time very slightly each time (Allain, 2017).

$$a. \quad \vec{F} = G\frac{m_1 m_2}{r^2}\hat{r}, \qquad G = 6.67 \times 10^{-11}$$

$$b. \quad \vec{F} = G\frac{m_1 m_2}{r^3}\vec{r}_{12}$$

$$c. \quad \frac{d\vec{v}_1}{dt} = G\frac{m_2}{r^3}\vec{r}_{12}$$

$$d. \quad \frac{d\vec{v}_1}{dt} = \sum_{j}^{n} G\frac{m}{r^3}\vec{r}$$

*Figure 1: Newton's Law of Gravitation can be adapted to describe the motion of a multi-body gravitational system.*

These approaches run into accuracy issues as well (albeit to a lesser extent than disregarding the third body), since modern computers don't have the memory to store the large number of variables or decimals to the required precision. With numerical integration, forces are assumed to be constant over small intervals of time; however, forces are dynamic in reality, so over time inaccuracies will become more pronounced. These assumptions result in less accuracy, especially as the simulation continues into the distant future. These issues are only compounded as the scale of the simulation increases. Simulating four, five, six, or more bodies becomes prohibitively computationally expensive. The equations used in the calculation not only grow in number, but also complexity as more bodies are introduced.

Considering that our solar system - with its planets, moons, asteroids, dwarf planets, and more - contains a number of gravitationally relevant bodies that reaches into the hundreds, a more energy and time efficient method of calculating these motions needs to be found.

To accomplish this goal without sacrificing accuracy, a machine-learning based approach will be explored. Datasets will be calculated using classical solutions (through numerical integration), and then the model can be trained on this dataset (Deshmukh, 2019). Instead of calculating trajectories, the model can recognize and generate patterns, saving large amounts of computing power. Moreover, the amount of computing power is constant - classical approaches can run into issues in certain simulations when bodies closely approach each other, resulting in large decimal place calculations.

Machine learning models consist of layers of nodes (meant to loosely model the way neurons in the brain work). Each node takes data as an input from the previous layer, sends the data through a function, and passes the result on to the next layer of nodes. The behavior of each node can be adjusted by an optimization function that alters constants and coefficients used in the function of the node. These constants and coefficients are known as biases and weights, respectively. Biases and weights are tweaked based on a loss function, which measures how far off the model is from the correct output. Models are generally trained in ‘rounds’ known as epochs, in which the model is trained on the entire dataset. Epochs are generally broken into batches, where the data is grouped into sets. The machine learning model is given a batch of data, and its loss function on that batch is calculated. The optimization function then uses the loss function to adjust weights, and then the process is repeated until the model attains a satisfactory accuracy. Another common technique used in training models is to break the dataset into two sets: a training dataset and a validation dataset. The training data set is much larger than the validation dataset, and is used to train the model with the previously discussed method involving epochs and batches. The validation dataset, often much smaller, is used to assess the ‘real-world’ accuracy of the model. This data has not been seen by the model before, and no adjustments are made to the model based on the outputs of the model when fed this data; validation data is simply used to obtain a measure of the model's accuracy.

Applying machine learning to the 3-body problem has been proven to be successful to a limited extent by researchers at the University of Edinburgh, although this study aims to expand on the concept by introducing additional complexity. The researchers generated datasets using classical methods to train and validate the model. The model used in the University of Edinburgh study consisted of 10 layers of 128 nodes each. They employed an Adaptive Moment Estimation optimizer (a commonly used algorithm that adjusts the biases and weights of the nodes to maximize the model's accuracy) while training the model. The model was trained on 9,900 three-body problems (100 problems were set aside as a validation dataset, for a total of 10,000 problems). The model was trained for 10,000 epochs, organized into batches of 5,000 data points. (Zhang, 2020). A rectified linear activation function (ReLU) was also used. The ReLU function is used in each node, and consists of a piecewise function that will output the input if the input is positive, otherwise it will output zero (Brownlee, 2020). The researchers at the University of Edinburgh limited their research to three equally sized bodies starting at a velocity of zero (Breen, Foley, Boekholt, & Zwart, 2020). This is not a broadly realistic scenario, so this inquiry will explore a greater number of bodies (four), different sized bodies, and bodies with starting velocities. Somewhat similar methods will be used, albeit more complex. The benefits in this area stand to be even more significant, as the number of data points is so much greater (*x* and *y* coordinates, initial speeds, directions, and masses), and so the computational power required would be significantly higher.

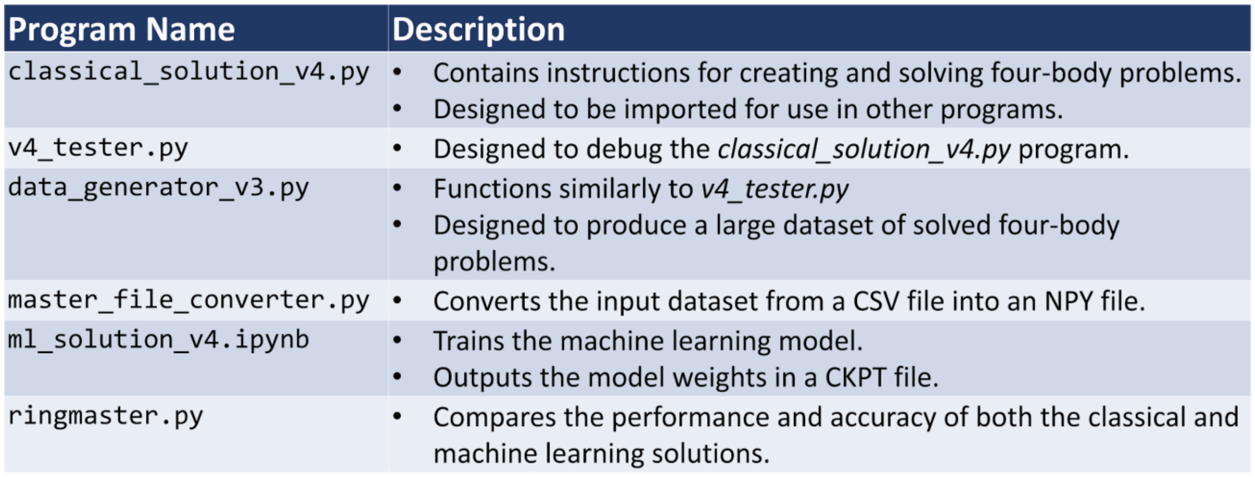
The machine learning model could then be used for a variety of additional use cases, such as calculating spaceship trajectories more efficiently. Current spaceship trajectories are generally calculated using a Hohmann Transfer, which minimizes the amount of fuel needed, but takes a long time to execute. The crux of the Hohmann Maneuver is that it takes advantage of the existing velocity of the body from which it leaves, applying only a small amount of force to correct its orbit. This new orbit, if timed properly, eventually intersects with the desired destination. The craft then readjusts its motion to match that of the destination. For this reason, the Hohmann Transfer is also somewhat inflexible, being usable in only certain common maneuvers.

Gravity-assisted maneuvers also provide benefits to fuel efficiency, on top of those that the Hohmann Transfer grants. In this process, a spacecraft flies past a gravitationally relevant body and uses the gravity of the body to ‘slingshot’ itself. A modified Hohmann transfer with a gravity assist could result in more efficient trajectories that use less fuel and less time. In certain scenarios, this strategy could even allow for more flexibility in course plotting, as Hohmann Transfers can only be executed during certain time periods (Doody, 2017).

This is not the only application for a machine-learning powered solution to predicting orbital motion. These programs could form the foundation for a new way to calculate and estimate orbital mechanics, providing opportunities for additional research into gravitational waves, black hole movements, and more.

## Materials and Methods

This study required the development of a significant number of related, but discrete programs that share logic. The flow of information may become difficult to track, so the following figures (figure 2 and figure 3) have been included to clarify the structure and dependencies of each program developed and employed by this study.



| Program Name | Description |
|---|---|
| `classical_solution_v4.py` | • Contains instructions for creating and solving four-body problems.<br>• Designed to be imported for use in other programs. |
| `v4_tester.py` | • Designed to debug the *classical_solution_v4.py* program. |
| `data_generator_v3.py` | • Functions similarly to *v4_tester.py*<br>• Designed to produce a large dataset of solved four-body problems. |
| `master_file_converter.py` | • Converts the input dataset from a CSV file into an NPY file. |
| `ml_solution_v4.ipynb` | • Trains the machine learning model.<br>• Outputs the model weights in a CKPT file. |
| `ringmaster.py` | • Compares the performance and accuracy of both the classical and machine learning solutions. |

*Figure 2: An overview of the programs developed for this project.*

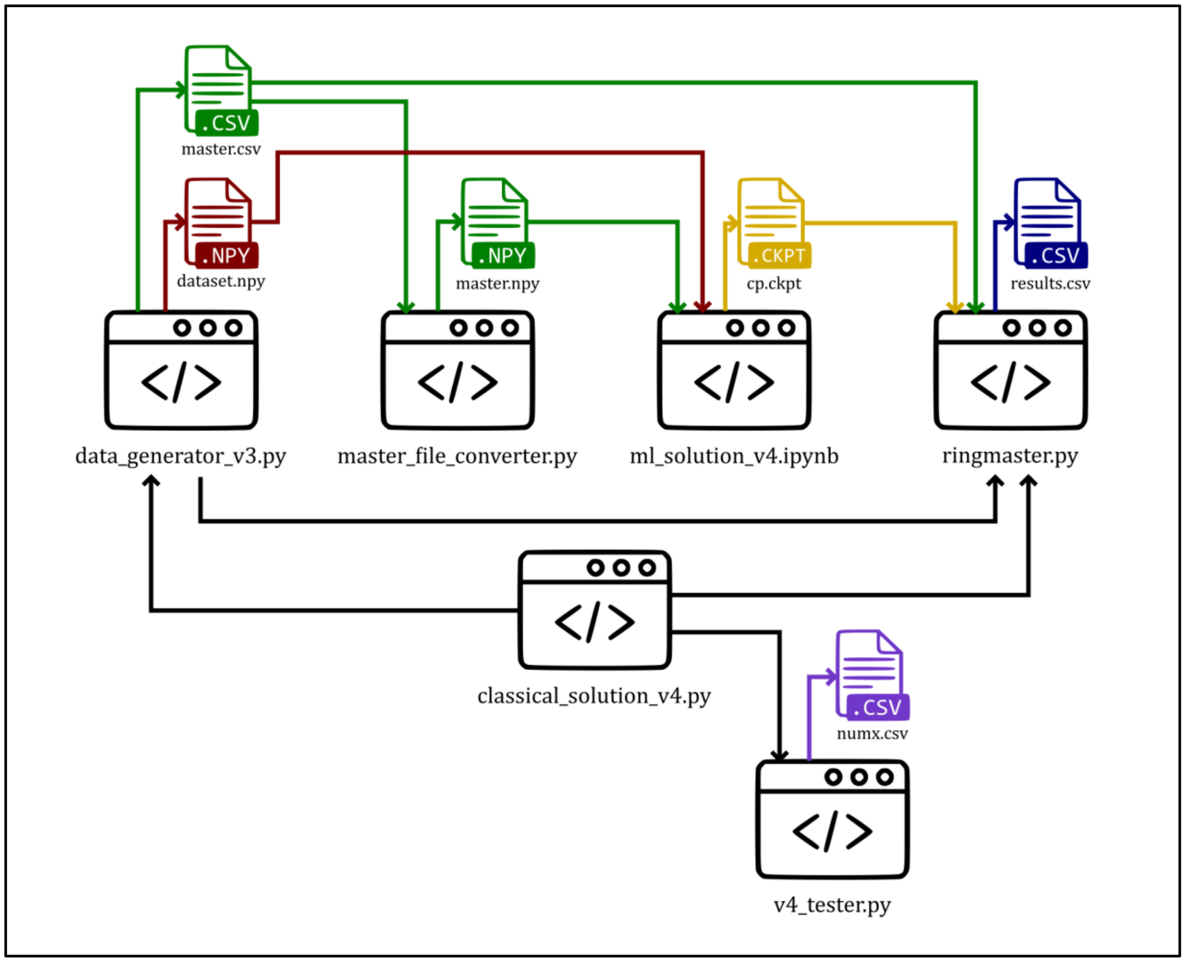


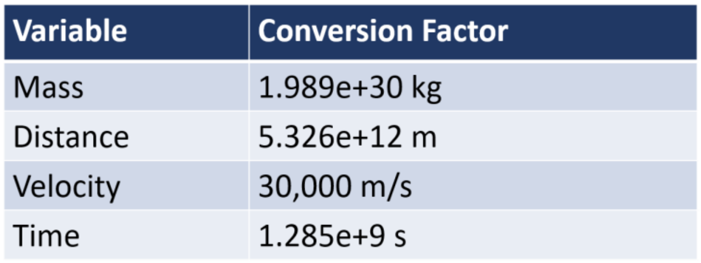
*Figure 3: This study involves a suite of programs that exchange data and logic to evaluate the viability of a machine learning approach to the* n*-body problem.*

**Developing *classical_solution_v4.py***

Phase one of the project involves creating a program to solve the *n*-body problem with a classical solution. To avoid excessive computational strain, the solution is limited to four bodies. A four body system is useful in such scenarios as launching a rocket from Earth to another planet, as the Earth-Moon-Sun-planet system would be a four-body system. Problems are solved two-dimensions, not three; while this may seem unrealistic, the Solar System is coplanar, meaning that all the planets orbit the Sun on roughly the same plane (NASA, 2017). Solving in two dimensions also significantly reduces the number of variables necessary in the calculations.

A program is written in Python to employ SciPy to calculate matrix norms (SciPy developers, 2021). Implementing SciPy's ordinary differential equation integrator (scipy.integrate.odeint) enables the program to numerically integrate the 4-body problem. Then, NumPy is utilized to convert vectors into usable arrays, as well as to perform operations on arrays (such as concatenation and flattening), and to create time interval arrays (NumPy, 2020). MatPlotLib's pyplot and animation modules are useful for visualizing the outputs of the classical solution (Matplotlib development team, 2020). Exporting the results of the program can be accomplished with Python's CSV module (Python Software Foundation, 2021). Python's time module is used to assess the performance of the program.

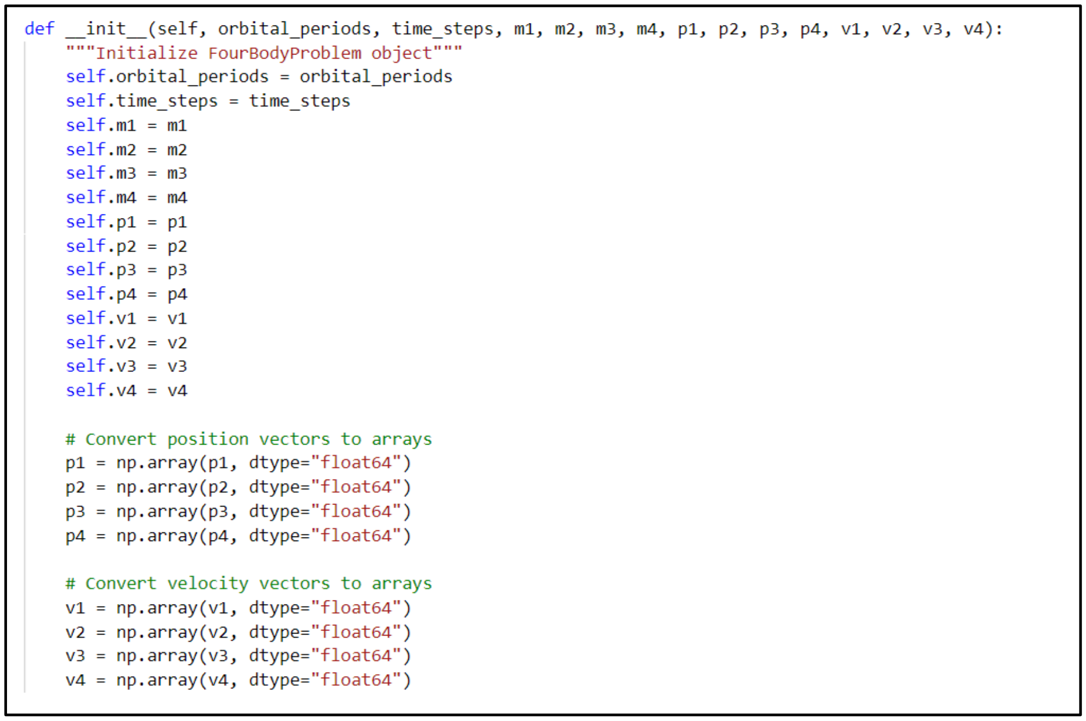
The heart of the classical solution is created by writing a class that acts as a core 'engine' for the additional programs built on top. Within this class, reference quantities are defined (see figure 4 for a list of all reference quantities). Reference quantities are the measurements of commonly understood objects or phenomena in the universe. In this case, they are the mass of the Sun and distance between Alpha Centauri A and Alpha Centauri B, and the time for one orbit of the Alpha Centauri system. Dividing values used in the equations by these quantities reduces their size. Instead of the mass variable for a body having a value in the billions of kilograms, it may have a value of 1.3, which can be understood as equivalent to 1.3 times the mass of the Sun. Not only are these values more easily understood by humans, they also prevent the use of unnecessarily large values that may exceed the memory limits imposed by the programming language, interpreter, or operating system.



| Variable | Conversion Factor |
|---|---|
| Mass | 1.989e+30 kg |
| Distance | 5.326e+12 m |
| Velocity | 30,000 m/s |
| Time | 1.285e+9 s |

*Figure 4: Conversion factors based on quantities found in the universe.*

Within the class, an __init__() function is defined to create instances of the four-body class (a class contains constants, instructions, and behaviors for how to accomplish different tasks with data specified by the user). The __init__() function intakes several parameters: number of orbital periods, the masses of the bodies, the initial positions of the bodies, and the initial velocities of the bodies (see figure 5). The function then creates a new 'instance' of the four-body problem with these parameters by assigning the function's parameters to instance variables.

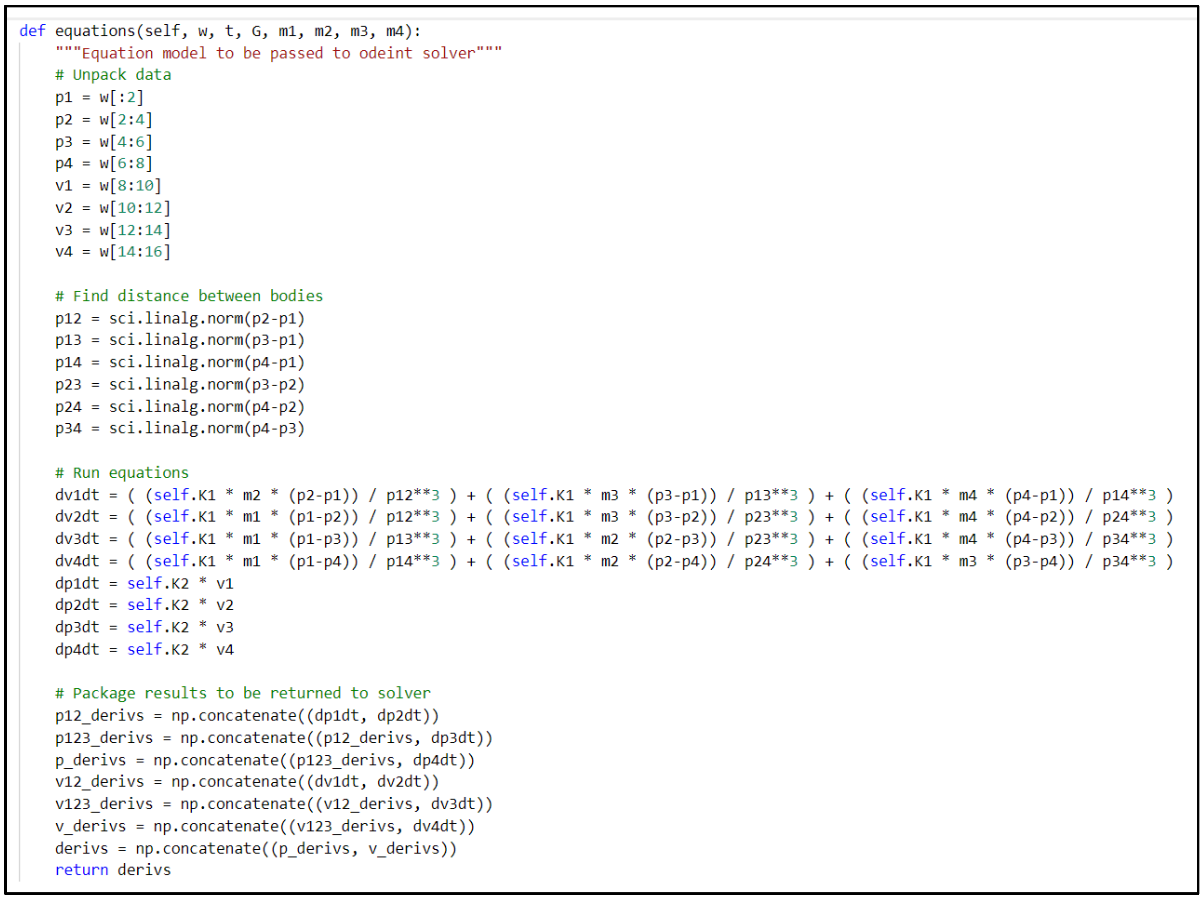


```
def __init__(self, orbital_periods, time_steps, m1, m2, m3, m4, p1, p2, p3, p4, v1, v2, v3, v4):
    """Initialize FourBodyProblem object"""
    self.orbital_periods = orbital_periods
    self.time_steps = time_steps
    self.m1 = m1
    self.m2 = m2
    self.m3 = m3
    self.m4 = m4
    self.p1 = p1
    self.p2 = p2
    self.p3 = p3
    self.p4 = p4
    self.v1 = v1
    self.v2 = v2
    self.v3 = v3
    self.v4 = v4

    # Convert position vectors to arrays
    p1 = np.array(p1, dtype="float64")
    p2 = np.array(p2, dtype="float64")
    p3 = np.array(p3, dtype="float64")
    p4 = np.array(p4, dtype="float64")

    # Convert velocity vectors to arrays
    v1 = np.array(v1, dtype="float64")
    v2 = np.array(v2, dtype="float64")
    v3 = np.array(v3, dtype="float64")
    v4 = np.array(v4, dtype="float64")
```

*Figure 5: The __init__() function intakes parameters and creates a new four-body problem.*

The process of creating a new instance from a class can be thought of as analogous to building a house from a blueprint. The blueprint can be used to create many different houses (think of each house as an instance) where each house has a unique color, just as the four-body class can be used to create many different simulations, each with unique starting conditions. The __init__() function is like the general contractor; it assigns the parameters to variables within the class to create a new instance of the class.

After an instance has been initialized, the calculate_trajectories() function can be called, which uses the SciPy ordinary differential equation solver to determine the positions of the bodies at defined time intervals. It then returns the position and velocity vectors as a two dimensional array. Adapt Newton's Law of Gravitation for these calculations (figure 1). The resulting differential equations can be packaged in the equations() method, which takes an array of initial conditions and returns a set of differential equations (figure 6). Packaging the equations in this manner enables them to be passed into the ordinary differential equation solver as a function argument.



```
def equations(self, w, t, G, m1, m2, m3, m4):
    """Equation model to be passed to odeint solver"""
    # Unpack data
    p1 = w[:2]
    p2 = w[2:4]
    p3 = w[4:6]
    p4 = w[6:8]
    v1 = w[8:10]
    v2 = w[10:12]
    v3 = w[12:14]
    v4 = w[14:16]

    # Find distance between bodies
    p12 = sci.linalg.norm(p2-p1)
    p13 = sci.linalg.norm(p3-p1)
    p14 = sci.linalg.norm(p4-p1)
    p23 = sci.linalg.norm(p3-p2)
    p24 = sci.linalg.norm(p4-p2)
    p34 = sci.linalg.norm(p4-p3)

    # Run equations
    dv1dt = ( (self.K1 * m2 * (p2-p1)) / p12**3 ) + ( (self.K1 * m3 * (p3-p1)) / p13**3 ) + ( (self.K1 * m4 * (p4-p1)) / p14**3 )
    dv2dt = ( (self.K1 * m1 * (p1-p2)) / p12**3 ) + ( (self.K1 * m3 * (p3-p2)) / p23**3 ) + ( (self.K1 * m4 * (p4-p2)) / p24**3 )
    dv3dt = ( (self.K1 * m1 * (p1-p3)) / p13**3 ) + ( (self.K1 * m2 * (p2-p3)) / p23**3 ) + ( (self.K1 * m4 * (p4-p3)) / p34**3 )
    dv4dt = ( (self.K1 * m1 * (p1-p4)) / p14**3 ) + ( (self.K1 * m2 * (p2-p4)) / p24**3 ) + ( (self.K1 * m3 * (p3-p4)) / p34**3 )
    dp1dt = self.K2 * v1
    dp2dt = self.K2 * v2
    dp3dt = self.K2 * v3
    dp4dt = self.K2 * v4

    # Package results to be returned to solver
    p12_derivs = np.concatenate((dp1dt, dp2dt))
    p123_derivs = np.concatenate((p12_derivs, dp3dt))
    p_derivs = np.concatenate((p123_derivs, dp4dt))
    v12_derivs = np.concatenate((dv1dt, dv2dt))
    v123_derivs = np.concatenate((v12_derivs, dv3dt))
    v_derivs = np.concatenate((v123_derivs, dv4dt))
    derivs = np.concatenate((p_derivs, v_derivs))
    return derivs
```

*Figure 6: The equations() function contains several adapted equations used to determine position and velocity of the four bodies. This is the heart of the classical solution program.*

Once the data has been generated, it must be visualized and exported. Creating a display_trajectories() method to visualize the trajectories with MatPlotLib accomplishes the first of these tasks. The method allows for the option to display an animated graph or a static graph and include an option to export the animation as a GIF. Finally, a to_csv() method is written that exports the results to a CSV file. While this to_csv() function could be used to train the machine learning model (and was initially intended for that purpose), a more efficient method contained in the data generator program will be explored shortly. This function is still critical during the debugging phase of program development, however, and should thus still be included.

## Developing *v4_tester.py*

A testing program is developed to evaluate the effectiveness of the four body problem engine. It imports the engine, calculates a problem, displays an animated rendition of the problem, and exports the results to a CSV file. This program is primarily useful for debugging *classical_solution_v4.py*.

## Developing *data_generator_v3.py*

Next, 10,000 solutions must be generated to train the machine learning model. However, generating thousands of simulations by hand for use in training the AI is not feasible, so an additional program must be built using the engine as a base. It randomly generates starting conditions (limit conditions to certain ranges to ensure that simulations reflect real-world phenomena) using Python’s random module, and constructs a four body problem with those conditions. The time period is not randomly generated; calculate all problems to three orbital periods with 1500 time points each. Then the program calculates the trajectories and verifies that the trajectories are valid. The simulation must be verified because some simulations suffer from a bug that manifests itself when two bodies in the simulation approach each other very closely. Because the program is unable to calculate to an infinite number of decimal places, the calculations break down during a close approach. This results in the erratic behavior displayed in figure 7. Throw out these invalid simulations and calculate a new simulation.

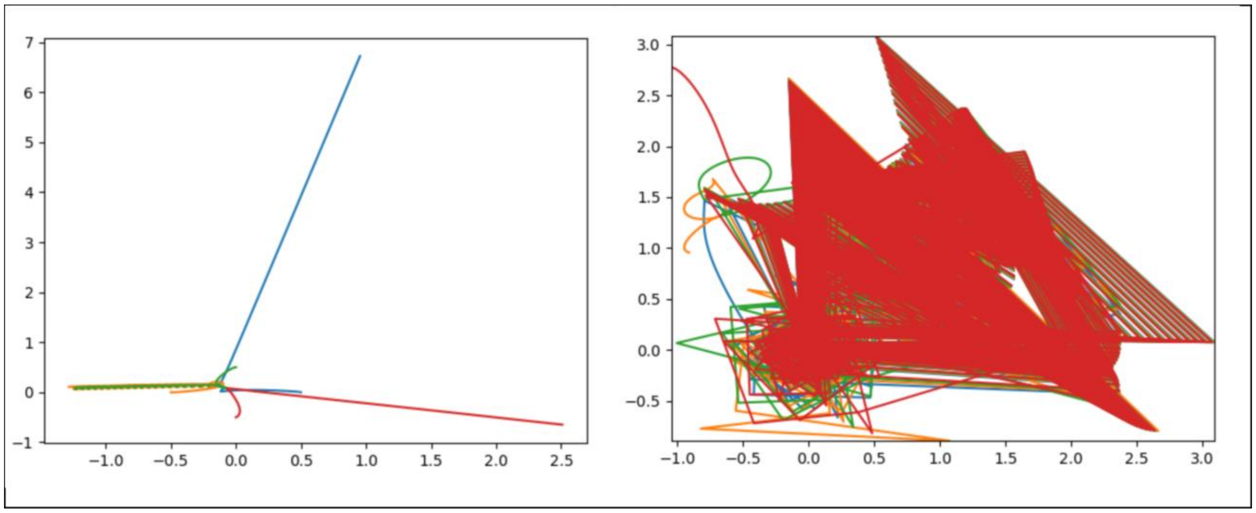


*Figure 7: The right graph demonstrates an example of the erratic behavior manifested following a close approach of two bodies. Compare this to a 'correct' simulation on the left.*

After verification, the data is exported to files to be transferred to the machine learning model program. Two files are generated: a master file with the initial conditions of each solved problem and a dataset of the generated solutions.

The first file is saved using the CSV file format, with one line for each solved problem. Each line should contain 20 floating point values describing the initial conditions of the problem: four mass values, eight position coordinates (two per vector), and eight velocity values (two per vector). There should be as many lines as there were problem solutions generated; in this case, 10,000. This data will serve as an input for the machine learning model.

The solutions to the problems should be saved to a separate file, since they will serve as the outputs to train the machine learning model on. Because 15 million lines of data will be generated by *data_generator_v3.py* for this project (each line containing the *x* and *y* coordinates for four gravitational bodies), the data is saved as an NPY file. Through project development it was determined that there were significant performance and file size benefits associated with using NPY files over CSV files.

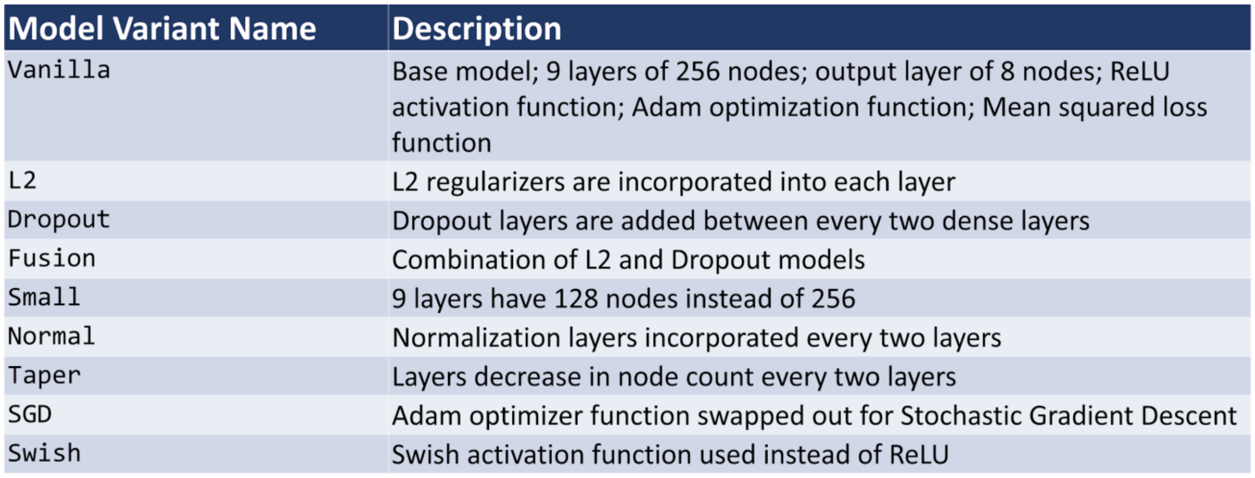
Given these benefits of NPY files over CSV files, it may seem as though the master file could also be saved in the NPY file format. Indeed, the CSV file will need to be converted for importation into *ml_solution_v4.ipynb*. Beforehand though, the output from *data_generator_v3.py* should be visually verified, which can only be done with a CSV file. Performance is less of an issue here, as the master file has only 10,000 lines of data compared to the dataset's 15 million. This format will also more easily be imported into the *ringmaster.py* program in the future.

## Developing *master_file_converter.py*

As previously discussed, the contents of the master CSV file must be copied into an NPY file so that the large amount of data can be efficiently handled. The *master_file_converter.py* script is developed to accomplish this. The program also processes the data to ease the importation process. This processing consists of expanding each line of initial conditions into 1,500 lines and appending a time point to each line. Following this processing, the master file will consist of 15 million lines of data, similar to the dataset file. Each line of data in the master file should correspond to the same line in the dataset file (e.g. line 10 in the master file should correspond to line 10 in the dataset file). Once processed and saved, preparation of the training dataset can be concluded.

## Developing *ml_solution_v4.py*

Now, the NPY files are imported into a Google Colaboratory notebook. Developing the machine learning model on Google Colaboratory leverages the performance benefits of the platform. The model is constructed with nine hidden layers of 256 nodes and an output layer of eight nodes as a Sequential model using the TensorFlow library (a Sequential model refers to the way the layers of nodes are stacked sequentially) (Google, 2020). The input shape of the model is 21 nodes: one node for the time point, four nodes for the masses, eight nodes for the position vectors, and eight nodes for the velocity vectors. The model's output is smaller, at only eight nodes total for the position vectors (four *x*, *y* coordinate pairs). Use an Adaptive Moment Estimation algorithm, and a ReLU activation function on each node. A mean-squared-error loss function is employed for model optimization. Train the model for 300 of epochs, batching the data into sets of 15,000 data points. Other model types are explored; the variations are stated in figure 8.



| Model Variant Name | Description |
|---|---|
| `Vanilla` | Base model; 9 layers of 256 nodes; output layer of 8 nodes; ReLU activation function; Adam optimization function; Mean squared loss function |
| `L2` | L2 regularizers are incorporated into each layer |
| `Dropout` | Dropout layers are added between every two dense layers |
| `Fusion` | Combination of L2 and Dropout models |
| `Small` | 9 layers have 128 nodes instead of 256 |
| `Normal` | Normalization layers incorporated every two layers |
| `Taper` | Layers decrease in node count every two layers |
| `SGD` | Adam optimizer function swapped out for Stochastic Gradient Descent |
| `Swish` | Swish activation function used instead of ReLU |

*Figure 8: The variations of models used in the study. The 'vanilla' model is a base for other variants.*

To transfer the model between programs (the model will be assessed in a separate program), the model weights are periodically saved in a checkpoint file. This functionality is built into Keras, and allows weights to be applied to a model of the same structure through the use of a simple text file.

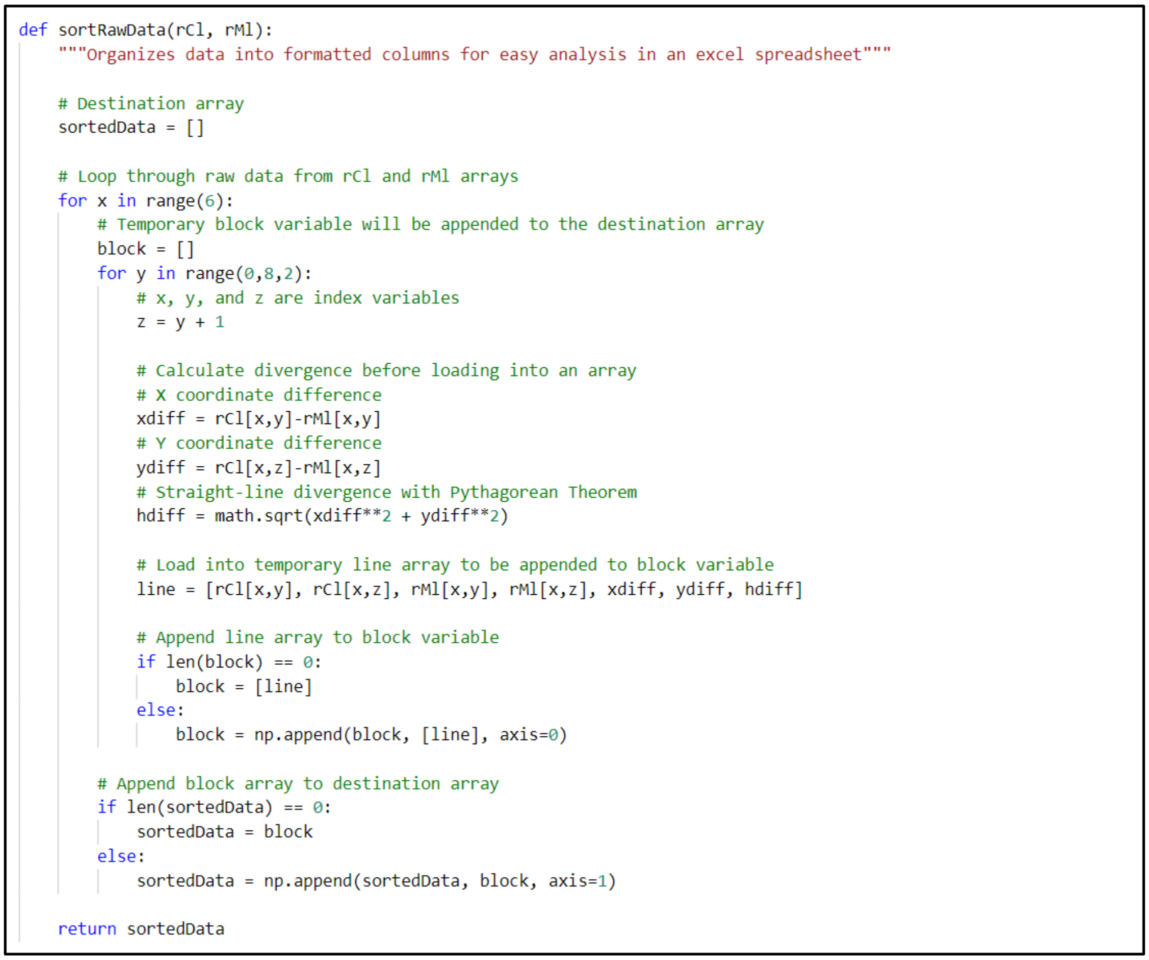
## Developing *ringmaster.py*

While TensorFlow includes a built-in accuracy function that yields a percentage, the formula (and resulting percent accuracy) are largely meaningless in the context of this study. More important is the model's divergence from the classical solution; this metric will allow a conclusion to be reached as to whether a machine learning model is capable of recognizing the patterns in these four-body problems. It is also critical to consider the computational performance of each solution, something not captured by a simple percent accuracy. To augment the statistical analysis discussed later, the motion of the system center of mass should also be captured, as this data will provide valuable insight into the model's understanding of the laws of physics. To obtain these metrics for 1,000 trials (each requiring nearly two dozen floating point values to be inputted), an additional program must be developed.

A main() method is developed to load the machine learning model from its weights and iteratively assess both the classical and machine learning solutions on as many problems as is specified. A loop that repeatedly calls the assess() function is implemented. Within the assess() function, the initial conditions of each problem must be generated; for this purpose, the getInput() function is written.

This function reuses code from *data_generator_v3.py* to randomly create initial conditions. In addition to this functionality, optional logic is coded into the getInput() function to pull data from the master file used in training the model.

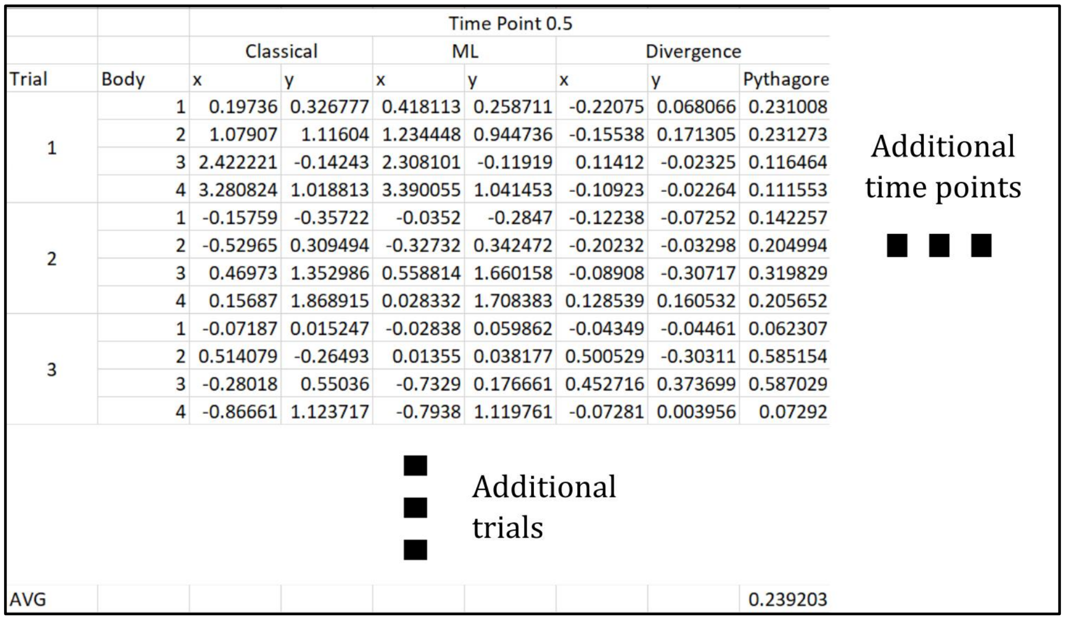
Once created, the getInput() function returns the initial conditions to the assess() function to be inputted into the classical solution (using *classical_solution_v4.py*) and the machine learning model. Each solution returns an array of four *x, y* coordinate pairs. To easily analyze these coordinates in comparison to each other, the sortRawData() function calculates absolute divergence using the Pythagorean Theorem (figure 9).



```
def sortRawData(rCl, rMl):
    """Organizes data into formatted columns for easy analysis in an excel spreadsheet"""

    # Destination array
    sortedData = []

    # Loop through raw data from rCl and rMl arrays
    for x in range(6):
        # Temporary block variable will be appended to the destination array
        block = []
        for y in range(0,8,2):
            # x, y, and z are index variables
            z = y + 1

            # Calculate divergence before loading into an array
            # X coordinate difference
            xdiff = rCl[x,y]-rMl[x,y]
            # Y coordinate difference
            ydiff = rCl[x,z]-rMl[x,z]
            # Straight-line divergence with Pythagorean Theorem
            hdiff = math.sqrt(xdiff**2 + ydiff**2)

            # Load into temporary line array to be appended to block variable
            line = [rCl[x,y], rCl[x,z], rMl[x,y], rMl[x,z], xdiff, ydiff, hdiff]

            # Append line array to block variable
            if len(block) == 0:
                block = [line]
            else:
                block = np.append(block, [line], axis=0)

        # Append block array to destination array
        if len(sortedData) == 0:
            sortedData = block
        else:
            sortedData = np.append(sortedData, block, axis=1)

    return sortedData
```

*Figure 9: The sortRawData() function uses a nested for loop to reorganize data from the classical and machine learning solutions. It uses the Pythagorean Theorem to determine the straight-line divergence of the machine-learning solution.*

The sortRawData() function can also organize the data in columns. The advantage of sorting the data in this fashion is made apparent in figure 10. Data is organized by type of solution and time period, allowing for averages to be easily calculated in Excel. To view the arrays of data, they are written to a CSV file (not an NPY file; these cannot be viewed in Excel).



| | | Time Point 0.5 | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | Classical | | ML | | Divergence | | |
| Trial | Body | x | y | x | y | x | y | Pythagore |
| 1 | 1 | 0.19736 | 0.326777 | 0.418113 | 0.258711 | -0.22075 | 0.068066 | 0.231008 |
| | 2 | 1.07907 | 1.11604 | 1.234448 | 0.944736 | -0.15538 | 0.171305 | 0.231273 |
| | 3 | 2.422221 | -0.14243 | 2.308101 | -0.11919 | 0.11412 | -0.02325 | 0.116464 |
| | 4 | 3.280824 | 1.018813 | 3.390055 | 1.041453 | -0.10923 | -0.02264 | 0.111553 |
| 2 | 1 | -0.15759 | -0.35722 | -0.0352 | -0.2847 | -0.12238 | -0.07252 | 0.142257 |
| | 2 | -0.52965 | 0.309494 | -0.32732 | 0.342472 | -0.20232 | -0.03298 | 0.204994 |
| | 3 | 0.46973 | 1.352986 | 0.558814 | 1.660158 | -0.08908 | -0.30717 | 0.319829 |
| | 4 | 0.15687 | 1.868915 | 0.028332 | 1.708383 | 0.128539 | 0.160532 | 0.205652 |
| 3 | 1 | -0.07187 | 0.015247 | -0.02838 | 0.059862 | -0.04349 | -0.04461 | 0.062307 |
| | 2 | 0.514079 | -0.26493 | 0.01355 | 0.038177 | 0.500529 | -0.30311 | 0.585154 |
| | 3 | -0.28018 | 0.55036 | -0.7329 | 0.176661 | 0.452716 | 0.373699 | 0.587029 |
| | 4 | -0.86661 | 1.123717 | -0.7938 | 1.119761 | -0.07281 | 0.003956 | 0.07292 |
| AVG | | | | | | | | 0.239203 |

*Figure 10: The data is sorted into columns so that statistics such as averages can be easily calculated.*

Lastly, the calcCOM() function takes the outputs of each solution and determines the system center of mass at each time point. The results are run through the sortRawData() function and then saved to a CSV file.

## Testing the Solutions

Randomly generating new initial conditions in the *ringmaster.py* program is advantageous as the initial conditions are considered a validation dataset. Validation data is data that the model has not been trained on; evaluating model accuracy with this data therefore provides a measure of real-world performance. The same initial conditions and time points are inputted into both the AI and the classical solution, and the divergence of the AI from the classical model is assessed. Straight-line divergence is calculated using the Pythagorean Theorem, and divergence along the *x* and *y* axes to identify any bias in the model is noted. In addition, the time required to compute these solutions in milliseconds is recorded. 1000 trials are evaluated.

While testing the model on a validation dataset is important for assessing real-world performance, testing the model on the training dataset can be helpful in identifying problems such as overfitting (a behavior in which the model learns the dataset, not the patterns in the data). For this purpose, another 1,000 trials are run, this time on the training dataset. The same metrics (straight-line divergence, *x* and *y* axis divergence, and computation time) are recorded.

Statistical analysis is accomplished through a matched-pairs t-test that compares the *x* values for the classical and machine learning solutions to determine whether there exists any statistically significant difference. This is repeated for the *y* values.

The system center of mass (COM), calculated by *ringmaster.py*, is also inspected. Consistent with the laws of physics, the motion of the system COM for each of the solutions should be the same, so evaluating the performance of the machine learning model with a physics-based analysis may provide further insight into it’s accuracy.

## Results

For the sake of brevity, this section will focus on results from the ‘Vanilla’ model variant. Data pertaining to the other model variations can be found in Appendix A.

The machine learning model was found to be partially adept at identifying patterns in gravitational motion. Graphs of the mean divergence over time reveal a trend of increasing divergence, shown in figure 11. This is true for both the training and validation assessments. Evident here is also a significant amount of overfitting, characterized by the sharp difference between the validation and training assessments.

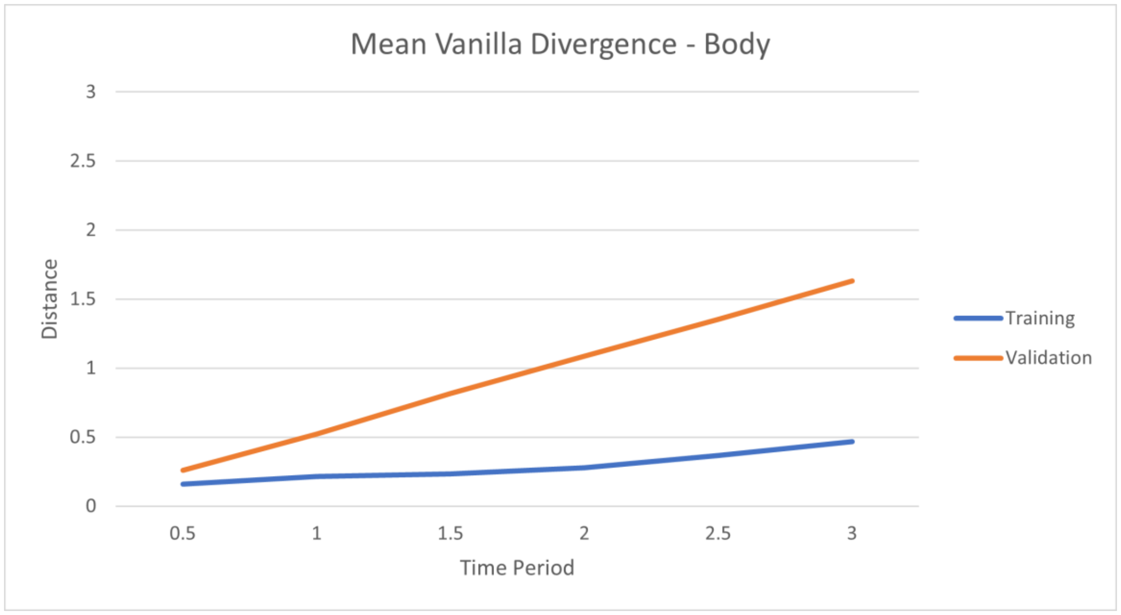


*Figure 11: Divergence is graphed against time. Recall that the distance and time values are non-dimensionalized; refer to figure 4 for conversion factors.*

Figure 12 compares the performance of all variations of the model. Similar trends can be found here as were evident in the ‘Vanilla’ model variant. Notable exceptions include the ‘Dropout’ and ‘Fusion’ models, which demonstrated some resistance to the trend of increasing time and increasing divergence.

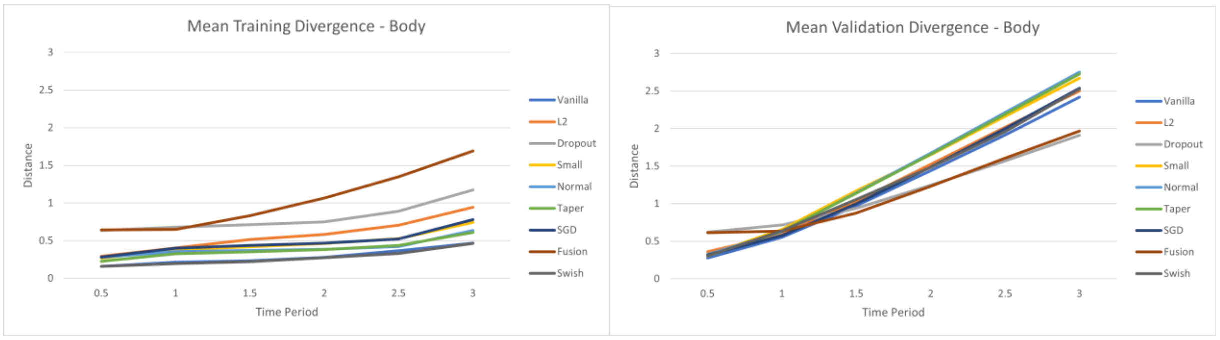


*Figure 12: Comparing the performance of all model variations.*

A matched-pairs t-test was determined to be the best statistical analysis of the data (figure 13). Applying these tests to the ‘Vanilla’ model yielded high p-values, indicating no statistically significant difference.

| t-Test: Paired Two Sample for Means | | |
|---|---|---|
| | *Variable 1* | *Variable 2* |
| Mean | -0.006515197 | -0.029307191 |
| Variance | 5.223436292 | 3.197411181 |
| Observations | 3984 | 3984 |
| Pearson Correlation | 0.299022479 | |
| Hypothesized Mean Di | 0 | |
| df | 3983 | |
| t Stat | 0.588447812 | |
| P(T<=t) one-tail | 0.27813256 | |
| t Critical one-tail | 1.645236285 | |
| P(T<=t) two-tail | 0.556265119 | |
| t Critical two-tail | 1.960559762 | |

| t-Test: Paired Two Sample for Means | | |
|---|---|---|
| | *Variable 1* | *Variable 2* |
| Mean | 0.088824038 | 0.053352034 |
| Variance | 7.467627678 | 3.020874325 |
| Observations | 3984 | 3984 |
| Pearson Correlation | 0.241393424 | |
| Hypothesized Mean Di | 0 | |
| df | 3983 | |
| t Stat | 0.782093841 | |
| P(T<=t) one-tail | 0.217102958 | |
| t Critical one-tail | 1.645236285 | |
| P(T<=t) two-tail | 0.434205916 | |
| t Critical two-tail | 1.960559762 | |

*Figure 13: The left t-test compares* x *values, while the right t-test compares* y *values at time 3.0*

With that in mind, a physics-based approach to data analysis was also applied. Graphed in figure 14 is the mean divergence of the model solution’s system center of mass (COM) from the classical solution’s COM. The trends in this analysis mirror those in the body divergence data, but with less divergence. This indicates the model has gained some understanding of the broad laws of physics.

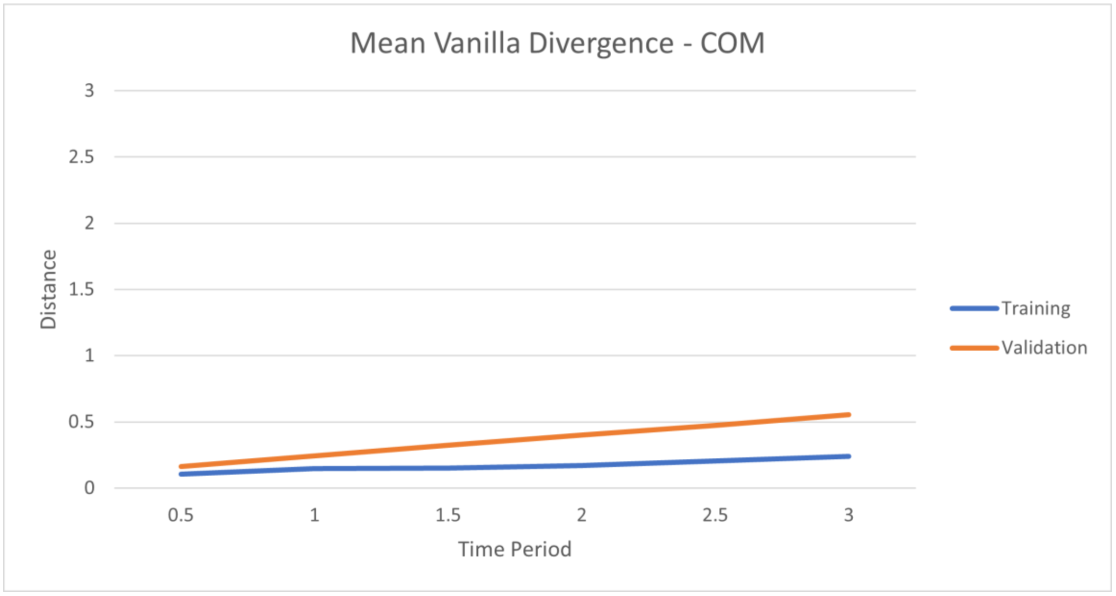


*Figure 14: Analysis of the system COM revealed surprising accuracy.*

Again, the other models demonstrated similar trends, seen in figure 15.

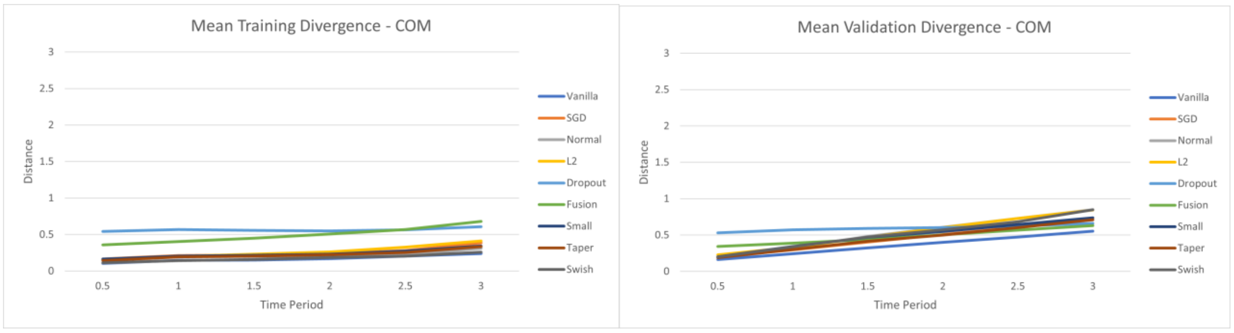


*Figure 15: System center of mass divergence of other models mirrors that of the 'Vanilla' model.*

It is also important to consider the performance of the model in terms of computing efficiency. Here, the model dramatically outperformed the classical solution. Figure 16 showcases this success.

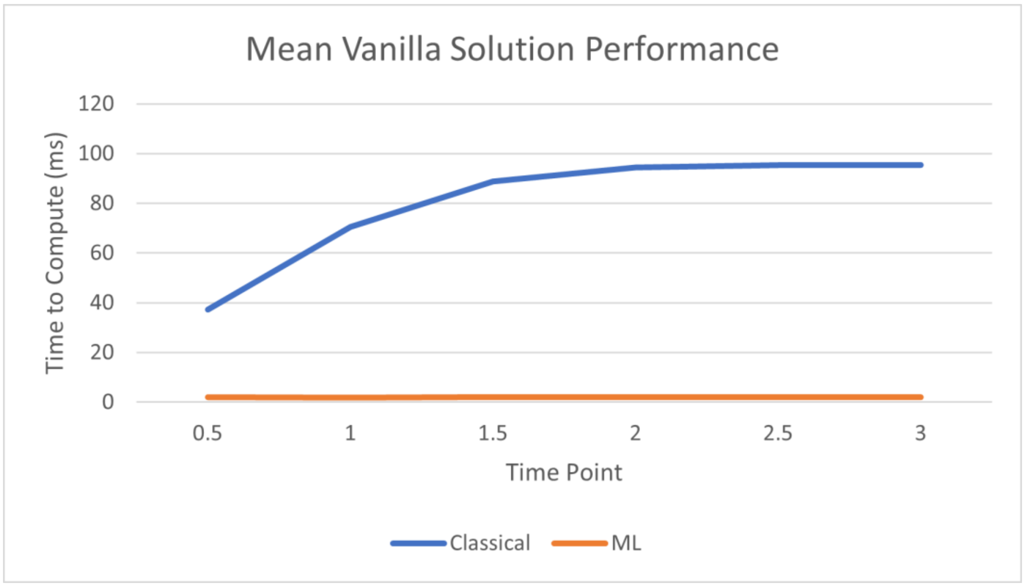


*Figure 16: Compare the compute time of the machine learning model to the classical solution. The machine learning solution shows near-negligible calculation times, contrasted by the compute time of the classical solution.*

## Discussion

### Implications

The machine learning model demonstrated some success in calculating the motions of four-body systems. The statistical analysis performed on the data corroborates this assertion; no statistically significant difference was found, so the analysis failed to reject the null hypothesis that the mean difference equaled zero. The model also demonstrated increased success in tracking the system center of mass, implying a deeper understanding of the underlying laws of physics at play in the *n*-body problem. See figure 17 for a visual example of this success.

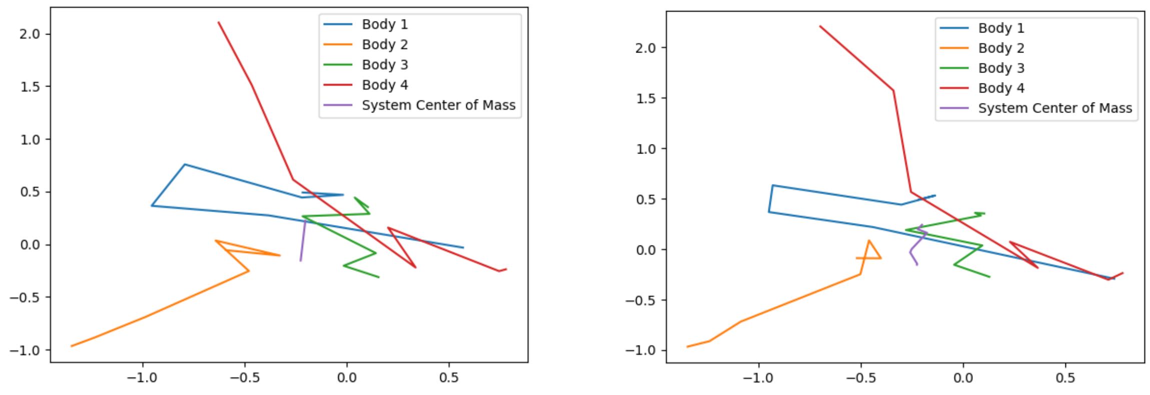


*Figure 17: Comparing the outputs of the classical solution (left) and the machine learning solution (right). Notice the subtle similarities and differences. These plots were created on training data by the vanilla model.*

However, the machine learning solution also demonstrated opportunities for improvement. The significant amount of overfitting represents a route to explore in increasing the accuracy of the model. Some success was found in accomplishing this through the use of dropout layers, employed in the 'Dropout' model variant.

The solutions built in this study are not ready for practical application but continue to support the use of machine learning in the realm of gravitational motion. This technology can be applied in the realm of interplanetary travel, assisting in the plotting of Hohmann trajectories, gravity-assisted courses, as well as in the study of other star systems. Alpha Centauri is one example of a three-star system. And, importantly, this novel solution can accomplish these feats with significant performance benefits.

## Assumptions and Limitations

The scope of this study was confined to two dimensions. As previously noted, this generally represents a significant portion of real-world gravitational motion and was necessary to reduce the number of variables involved in the project (motivated by a lack of computing power). Nonetheless, simulating a three-dimensional world with two dimensions does not completely capture real-world behavior.

Computing power also limited the project in other ways. The dataset carried some error; without a supercomputer, the dataset had to be calculated more granularly than would be desirable, resulting in increased levels of error. Moreover, the machine learning solution could not be fine-tuned to the level that would be preferable due to time constraints imposed by the lack of computing resources. Training and further adjustment of the model could result in improved accuracy that was unobtainable by this study.

## Future Studies

This study represents a conceptual foundation; it has demonstrated that machine learning based solutions are capable of being applied to more complex variations of the *n*-body problem with drastic performance benefits. However, with the lack of resources noted in the previous section, the levels of accuracy obtained by this study are unsatisfactory for a practical application. Future studies should thus focus on refining the progress made here, with larger, more random datasets to address the overfitting of the model.

## Conclusion

This study has demonstrated that machine learning is capable of modeling the complex partial differential equations that describe the motion of gravitational bodies. The model has demonstrated particularly deep understanding of the underlying laws of physics, showcased in the ability of the model to track the system center of mass. And, the model can do so significantly more efficiently than classical solutions.

Breen, Foley, Boekholt, & Zwart demonstrated at the University of Edinburgh that machine learning can be applied to the modeling of gravitational motion, and this study has taken the next step to establish it as a practical application by expanding upon the complexity of the gravitational systems being modeled (2020). However, further progress must be made before this technology can be practically applied to the real-world. The issue of overfitting must be addressed and the accuracy must be improved. Nonetheless, this study holds promise for future innovation in the fields of spacecraft course calculations, the study of other star systems, as well as better analysis of the motions of other gravitational bodies such as black holes.

## References

Allain, R. (2017, June 03). This Is the Only Way to Solve the Three-Body Problem. Retrieved October 20, 2020, from https://www.wired.com/2016/06/way-solve-three-body-problem/

Breen, P. G., Foley, C. N., Boekholt, T., & Zwart, S. P. (2020). Newton versus the machine: Solving the chaotic three-body problem using deep neural networks. Monthly Notices of the Royal Astronomical Society, 494(2), 2465-2470. doi:10.1093/mnras/staa713

Brownlee, J. (2020, August 20). A Gentle Introduction to the Rectified Linear Unit (ReLU). Retrieved October 22, 2020, from https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/

Deshmukh, G. (2019, July 02). Modelling the Three Body Problem in Classical Mechanics using Python. Retrieved September 23, 2020, from https://towardsdatascience.com/modelling-the-three-body-problem-in-classical-mechanics-using-python-9dc270ad7767

Doody, D. (2017, February). Basics of Space Flight - Solar System Exploration: NASA Science. Retrieved September 17, 2020, from https://solarsystem.nasa.gov/basics/chapter4-1

Google. (2020). TensorFlow (Version 2.4) [Computer software]. Retrieved from https://www.tensorflow.org/

Halpern, P. (2018, February 13). Chaos Theory, The Butterfly Effect, And The Computer Glitch That Started It All. Retrieved September 23, 2020, from https://www.forbes.com/sites/startswithabang/2018/02/13/chaos-theory-the-butterfly-effect-and-the-computer-glitch-that-started-it-all/

Matplotlib development team. (2020). MatPlotLib (Version 3.3.3) [Computer software]. Retrieved from https://matplotlib.org/

NASA. (2017). Basics of Space Flight - Solar System Exploration: NASA Science. Retrieved January 15, 2021, from https://solarsystem.nasa.gov/basics/chapter4-1/

NumPy. (2020). NumPy (Version 1.19) [Computer software]. Retrieved from https://numpy.org/

Python Software Foundation. (2021). Python (Version 3.9.1) [Computer software]. Retrieved from https://www.python.org/

SciPy developers. (2021). SciPy (Version 1.19.5) [Computer software]. Retrieved from https://www.scipy.org/

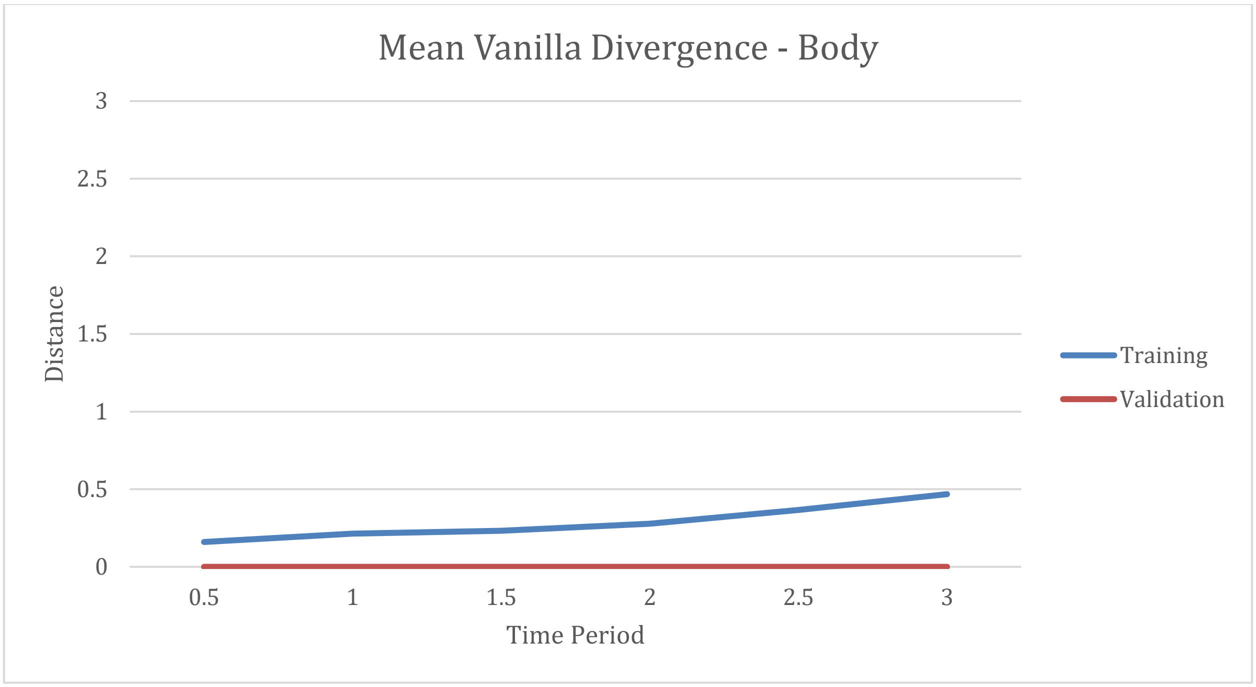
The Editors of Encyclopaedia Britannica. (2016, October 24). Three-body problem. Retrieved September 23, 2020, from https://www.britannica.com/science/three-body-problem

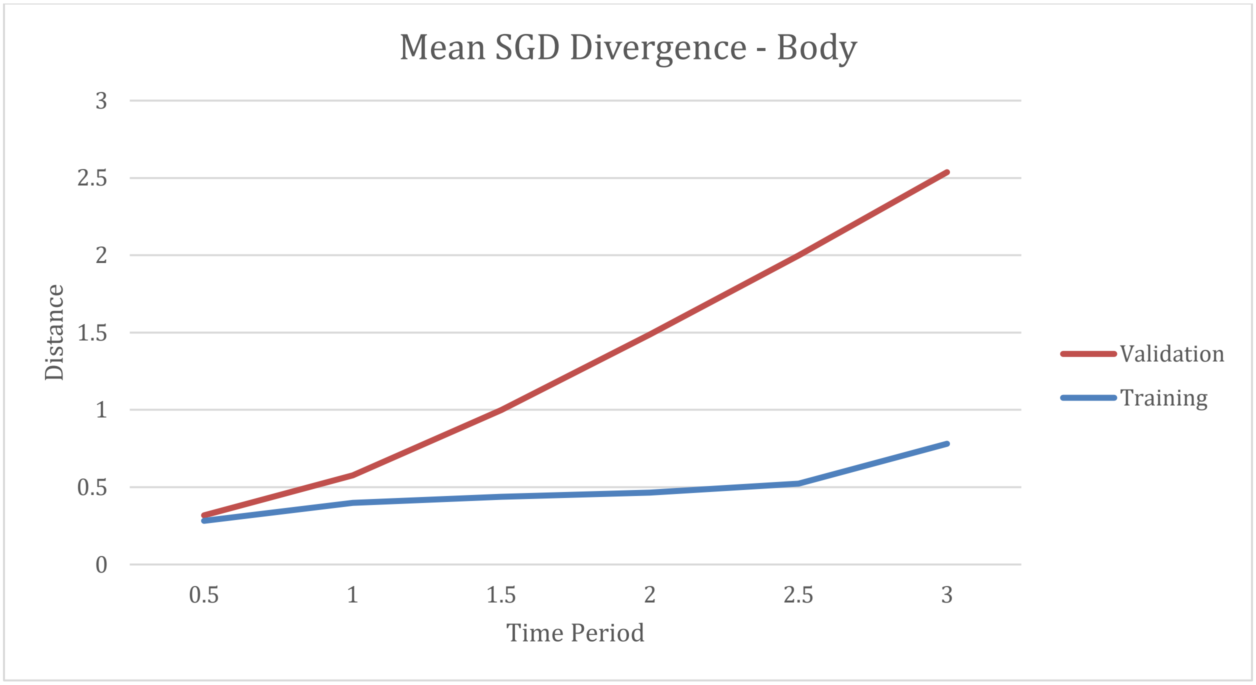
Wolfram, S. (2002). Chapter 7: Mechanisms in Programs and Nature. In *A New Kind of Science* (p. 972). Champaign, IL: Wolfram Media. doi:https://www.wolframscience.com/reference/notes/972d#:~:text=The%20three-body%20problem%20was,to%20so-called%20Lagrange%20points.

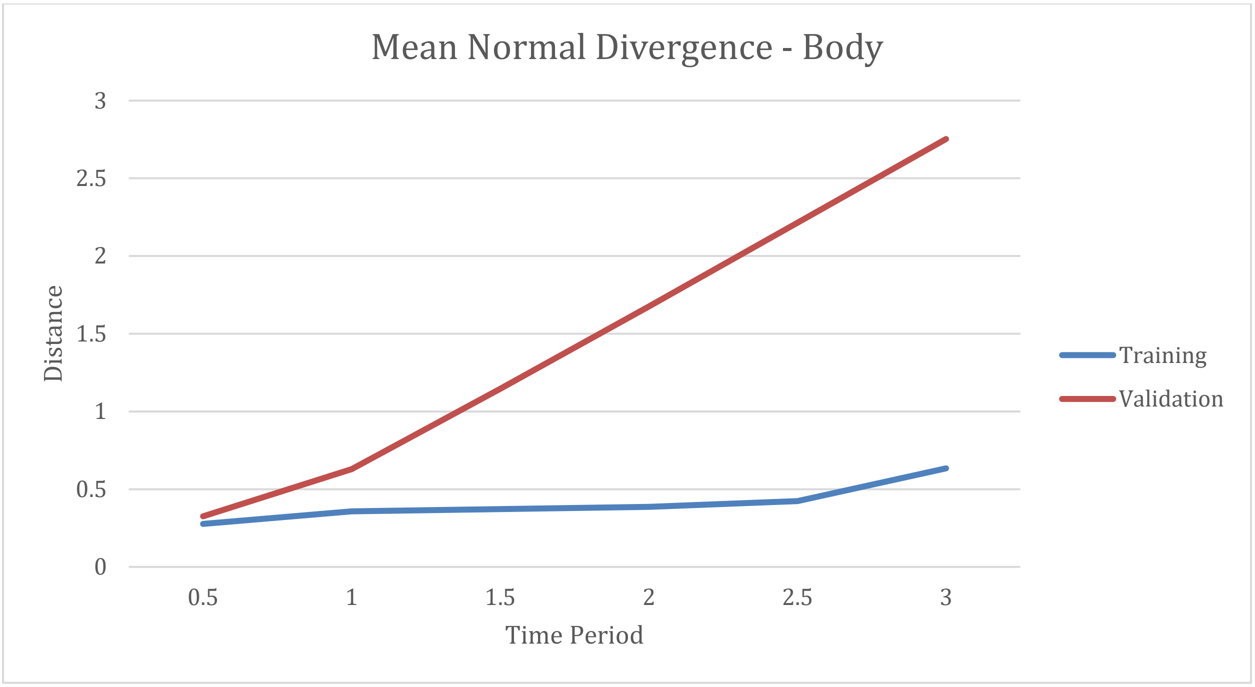
Zhang, J. (2020, January 21). Optimisation Algorithm-Adaptive Moment Estimation(Adam). Retrieved October 22, 2020, from https://towardsdatascience.com/optimisation-algorithm-adaptive-moment-estimation-adam-92144d75e232

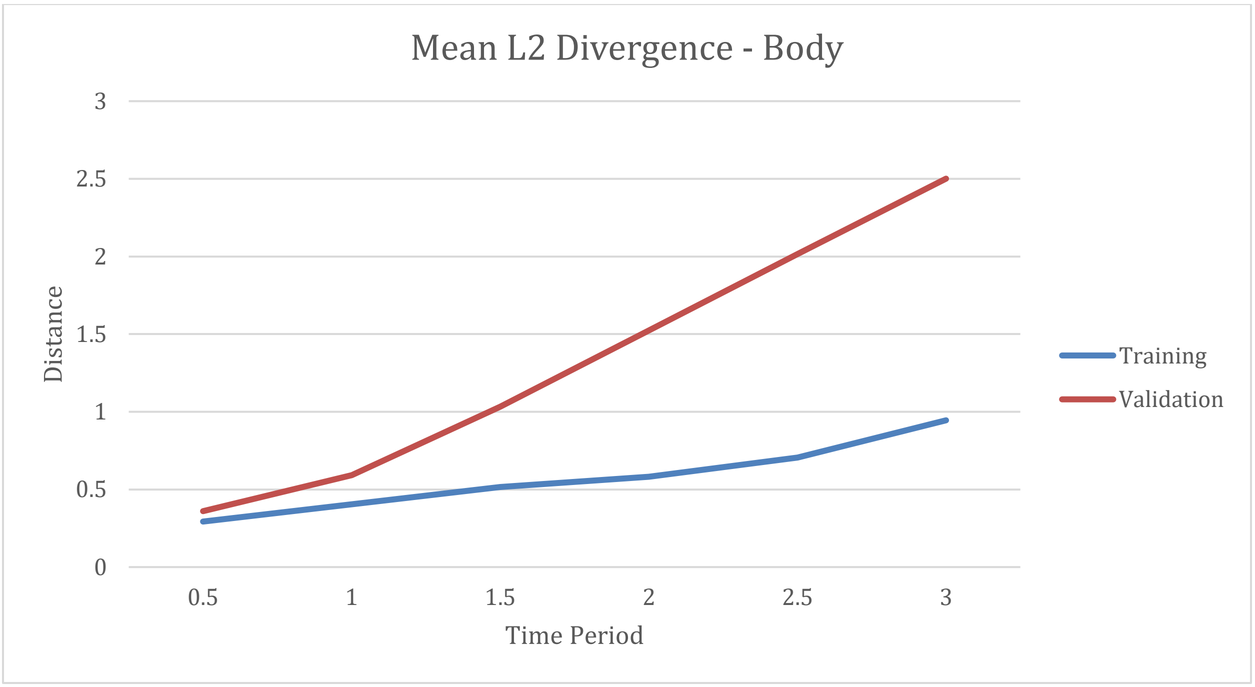
## Appendix A: Data

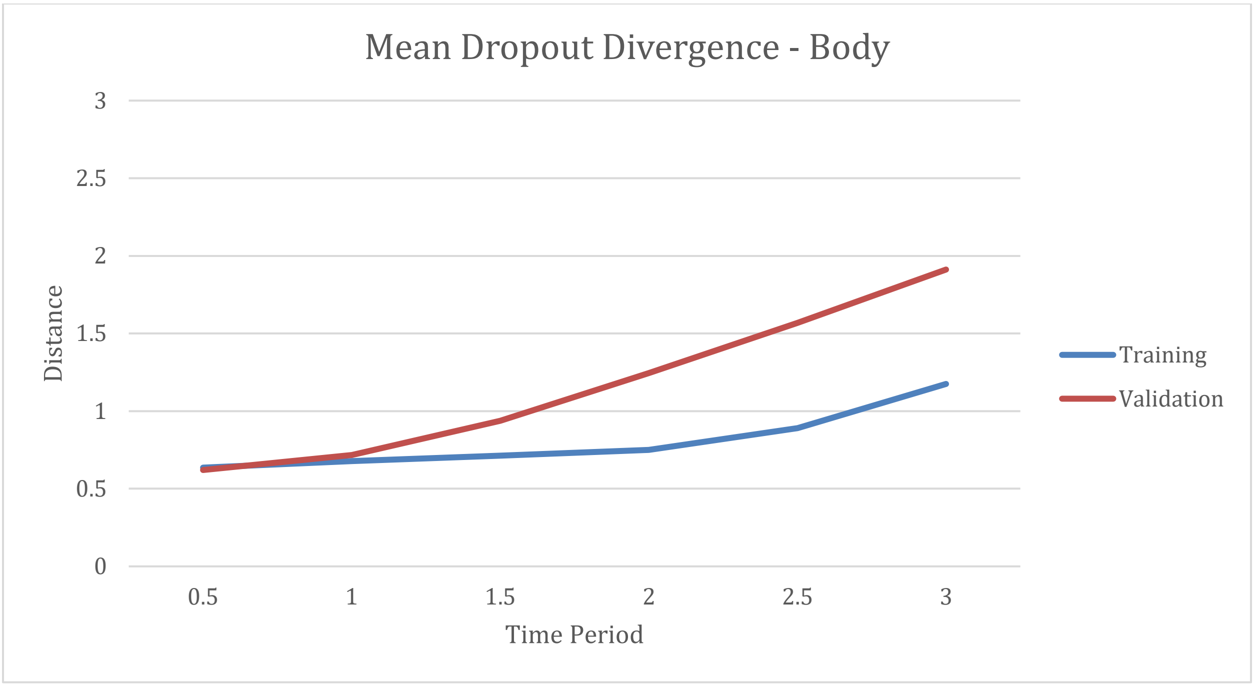
### Body Trajectory Divergence

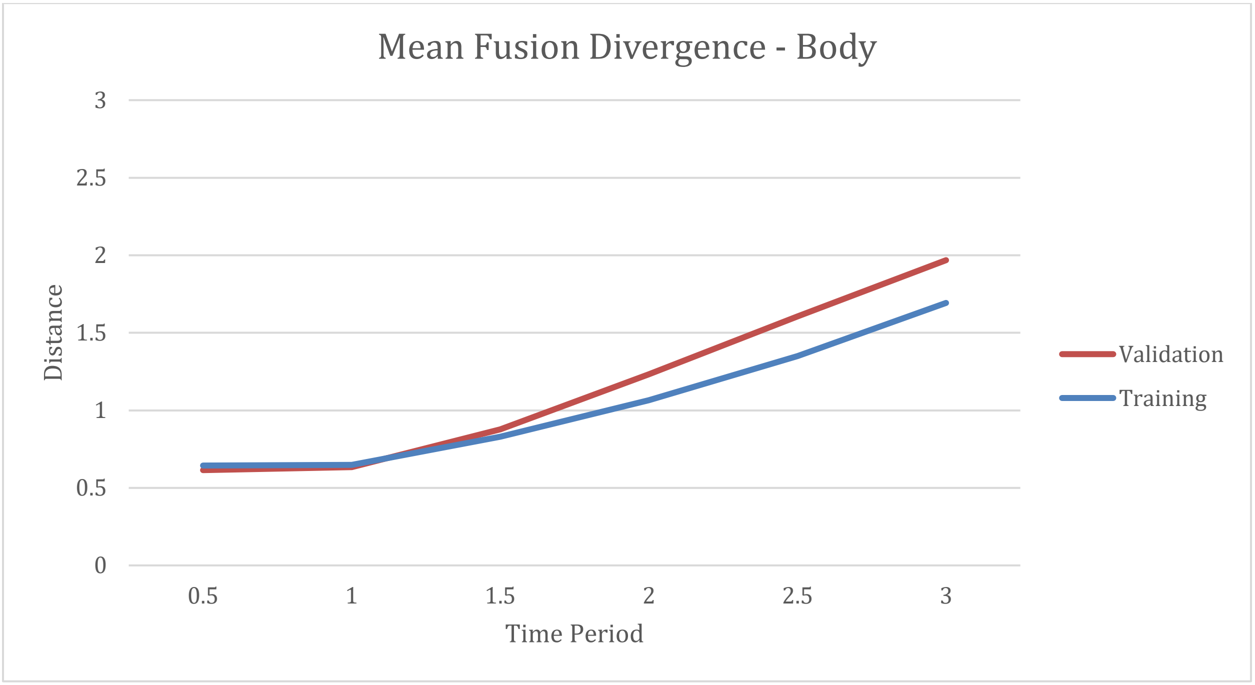


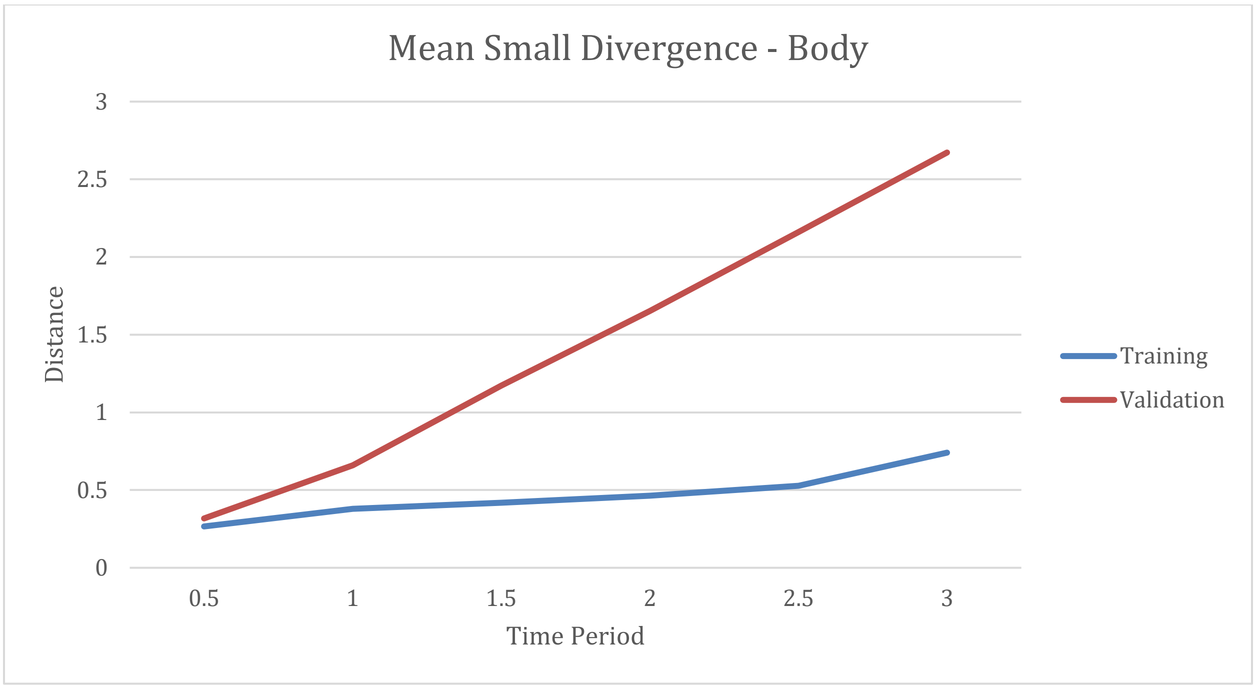


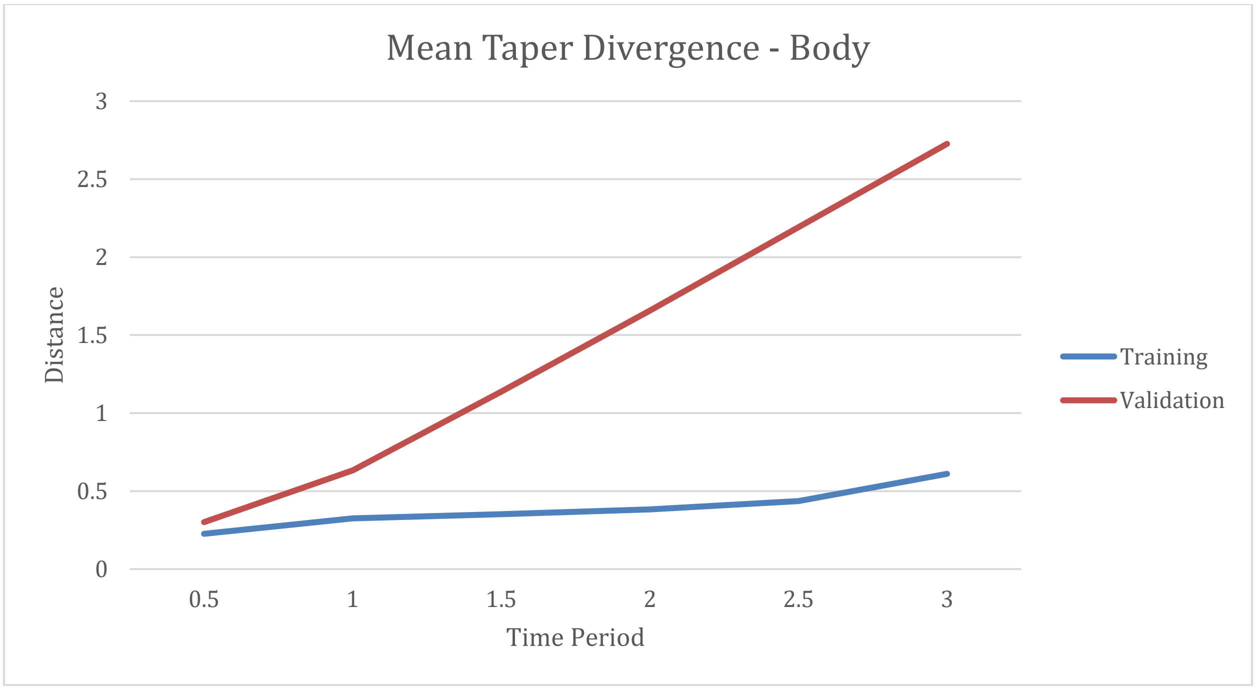

Mean Normal Divergence - Body
Distance
Time Period
Training
Validation
0
0.5
1
1.5
2
2.5
3

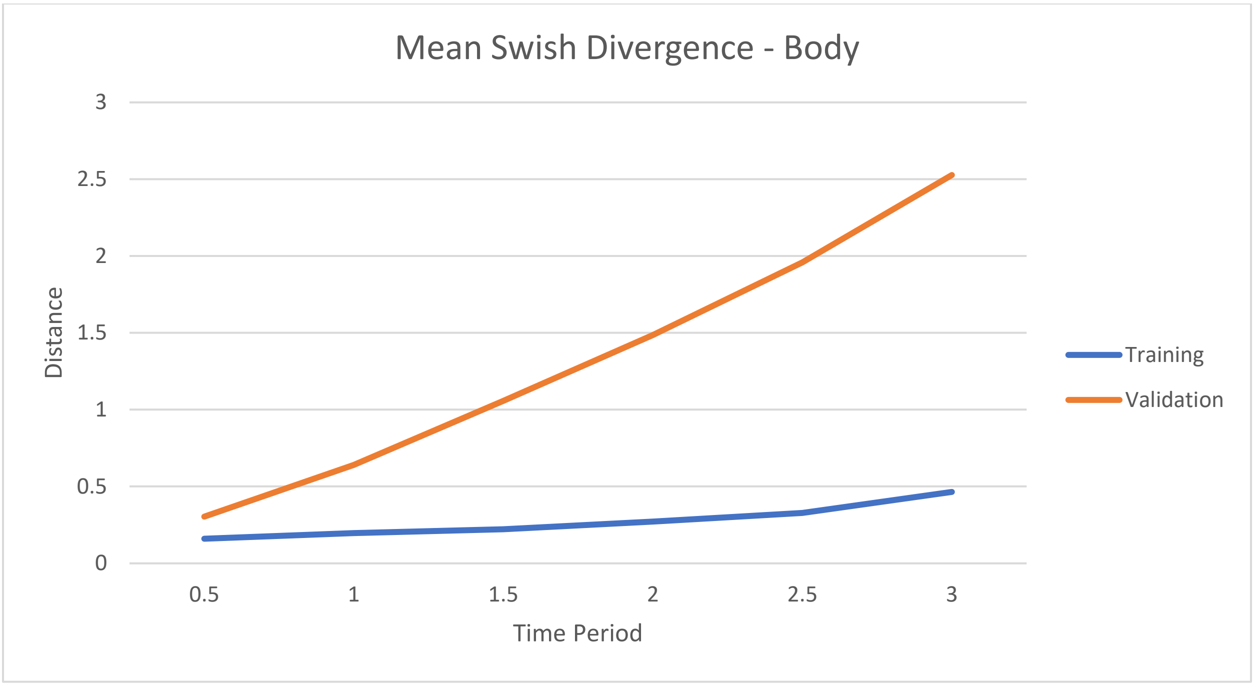

Mean L2 Divergence - Body
Distance
Time Period
Training
Validation
0
0.5
1
1.5
2
2.5
3


Mean Dropout Divergence - Body
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Training
Validation

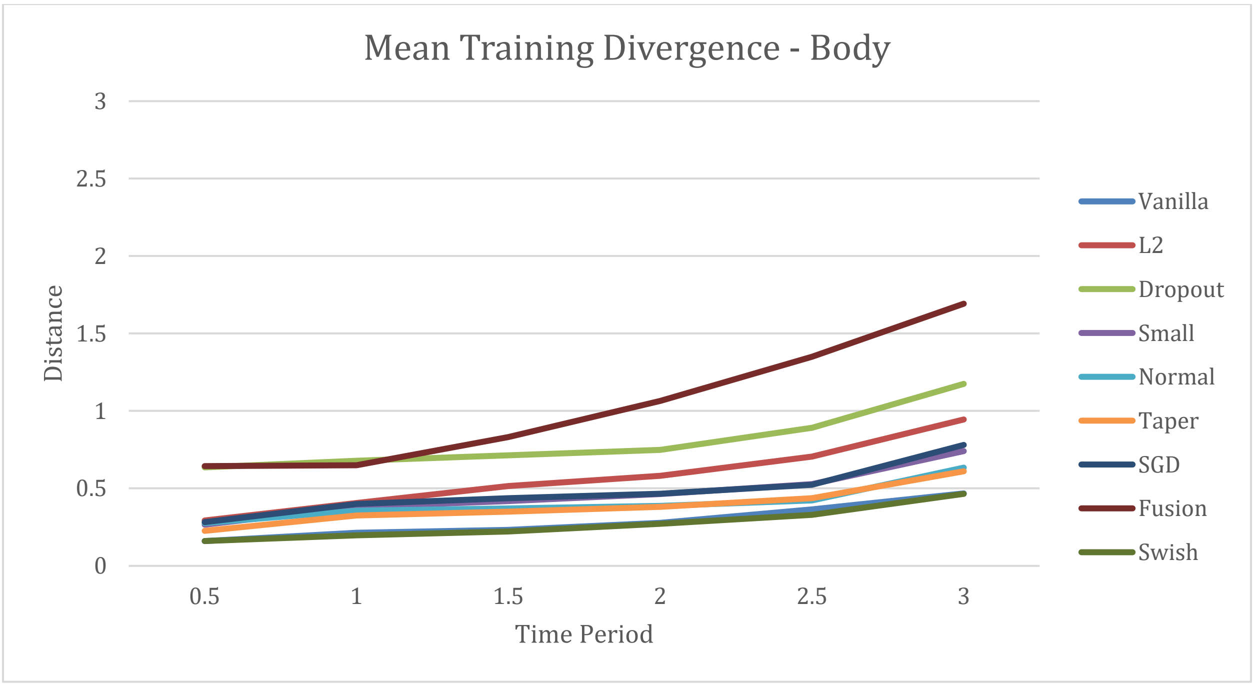

Mean Fusion Divergence - Body
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Validation
Training


Mean Small Divergence - Body
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Training
Validation

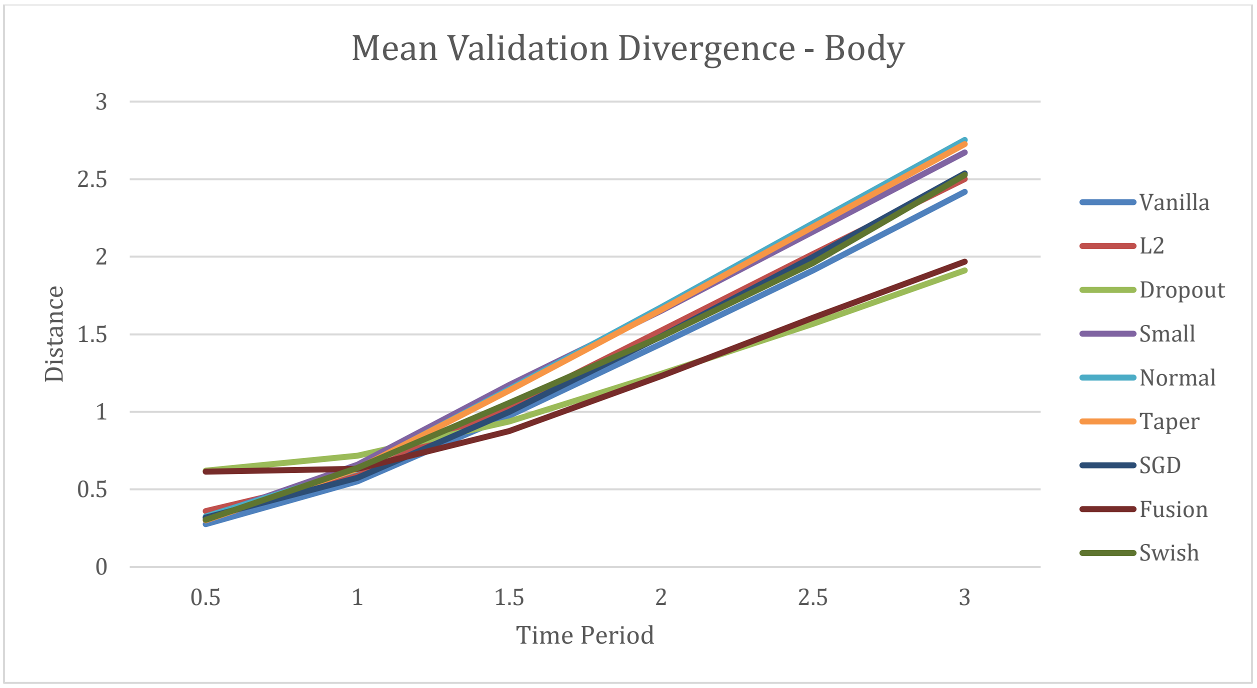

Mean Taper Divergence - Body
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Training
Validation



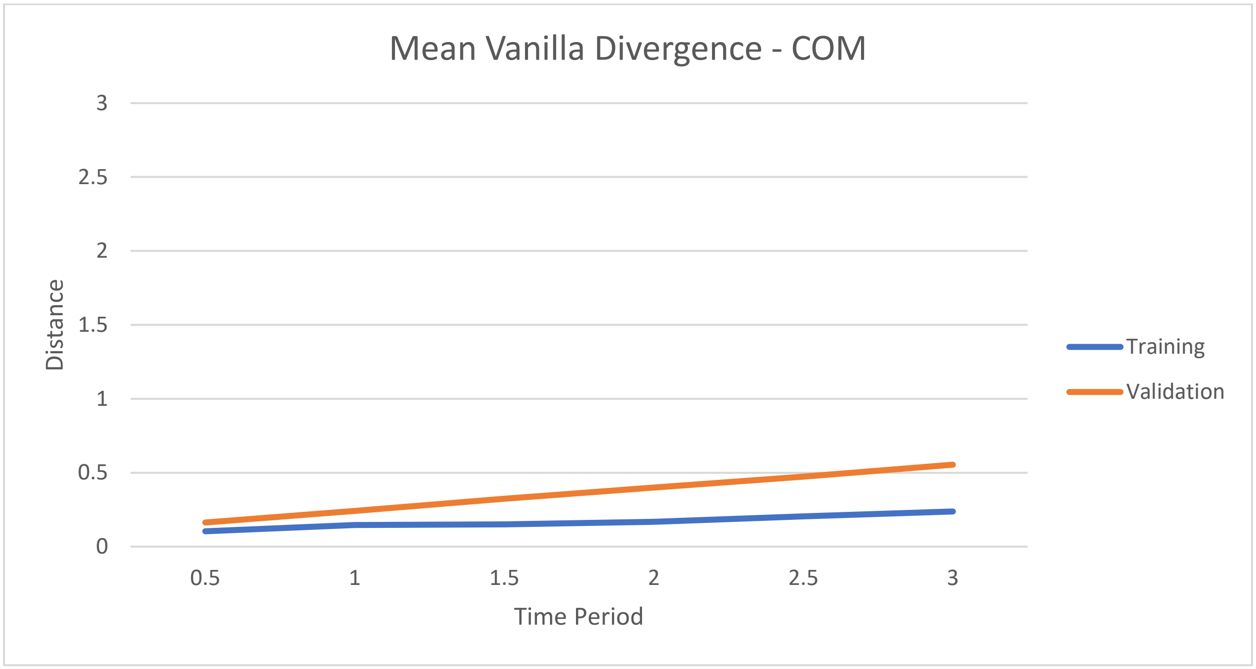
Mean Swish Divergence - Body
Distance
Time Period
0
0.5
1
1.5
2
2.5
3
0.5
1
1.5
2
2.5
3
Training
Validation

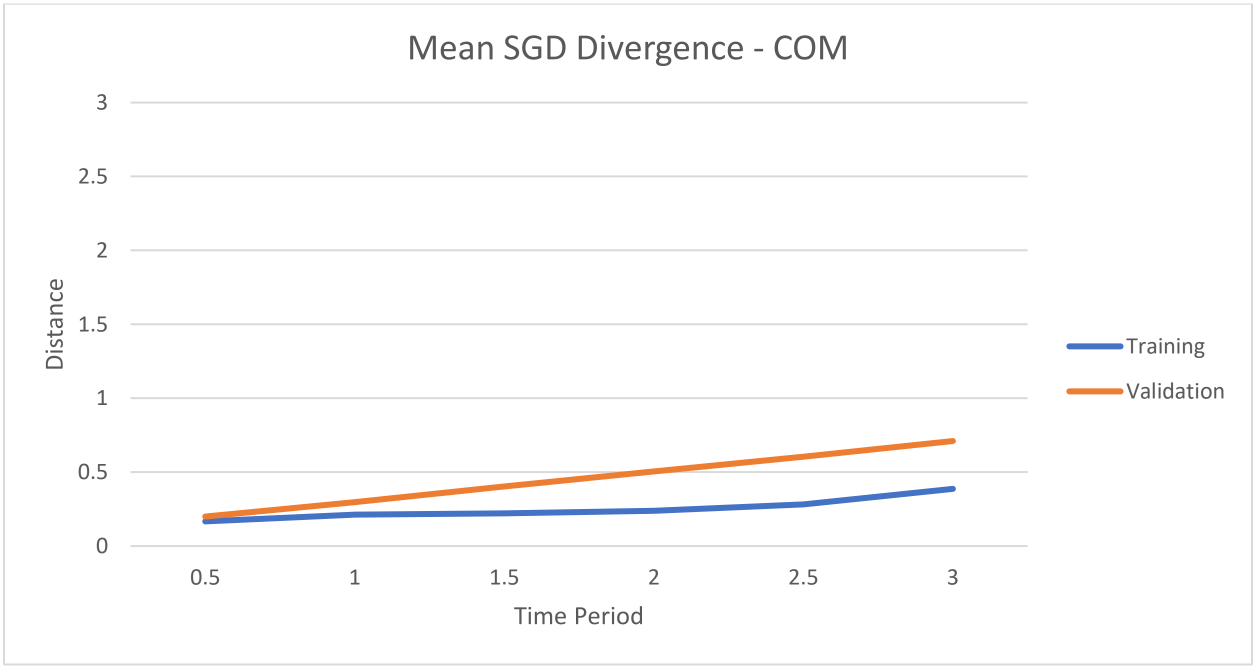


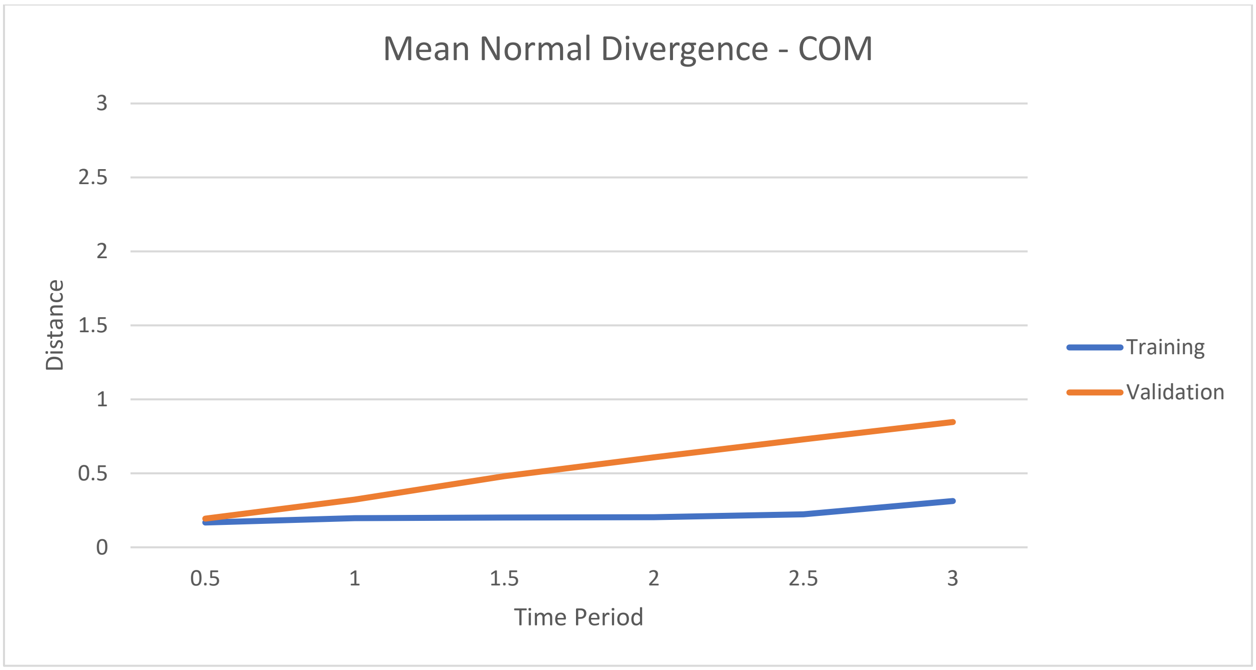
Mean Training Divergence - Body
Distance
Time Period
0
0.5
1
1.5
2
2.5
3
0.5
1
1.5
2
2.5
3
Vanilla
L2
Dropout
Small
Normal
Taper
SGD
Fusion
Swish

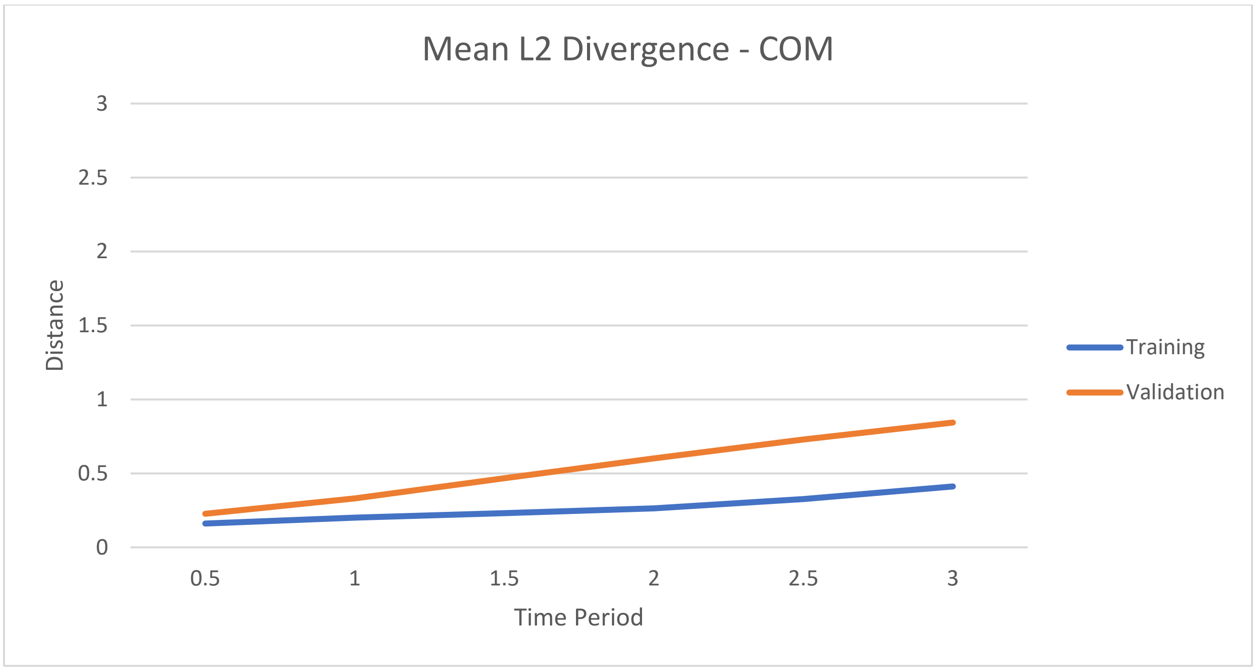

Mean Validation Divergence - Body
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Vanilla
L2
Dropout
Small
Normal
Taper
SGD
Fusion
Swish

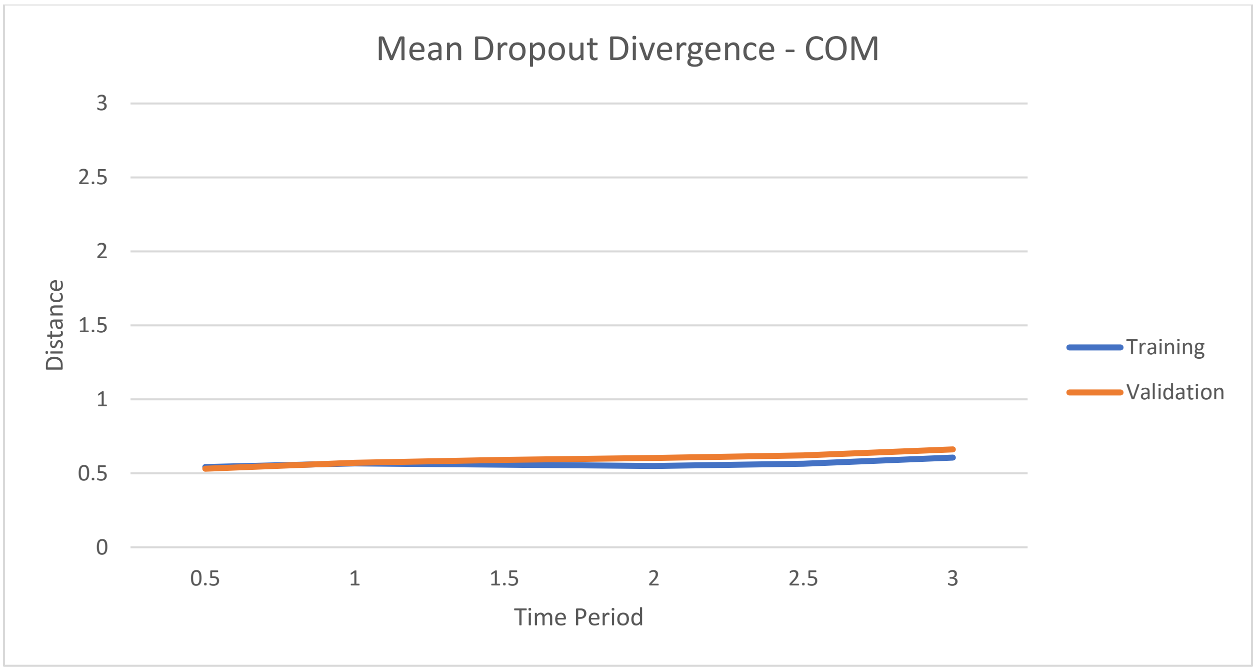
**System Center of Mass Divergence**

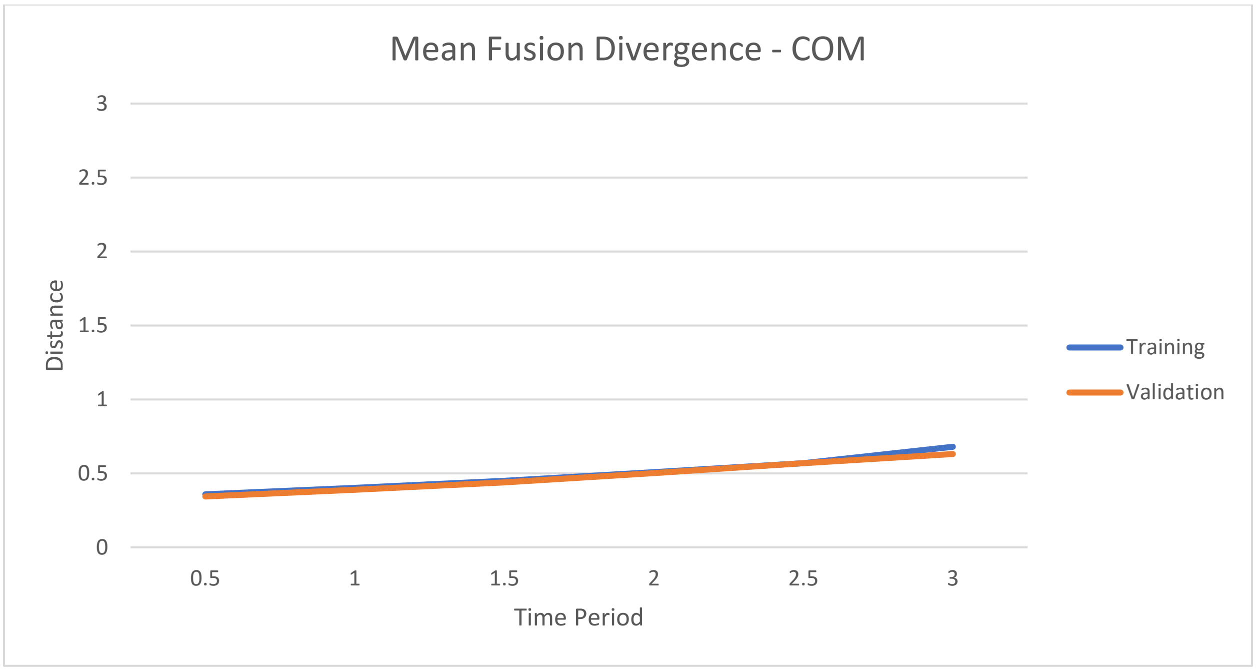


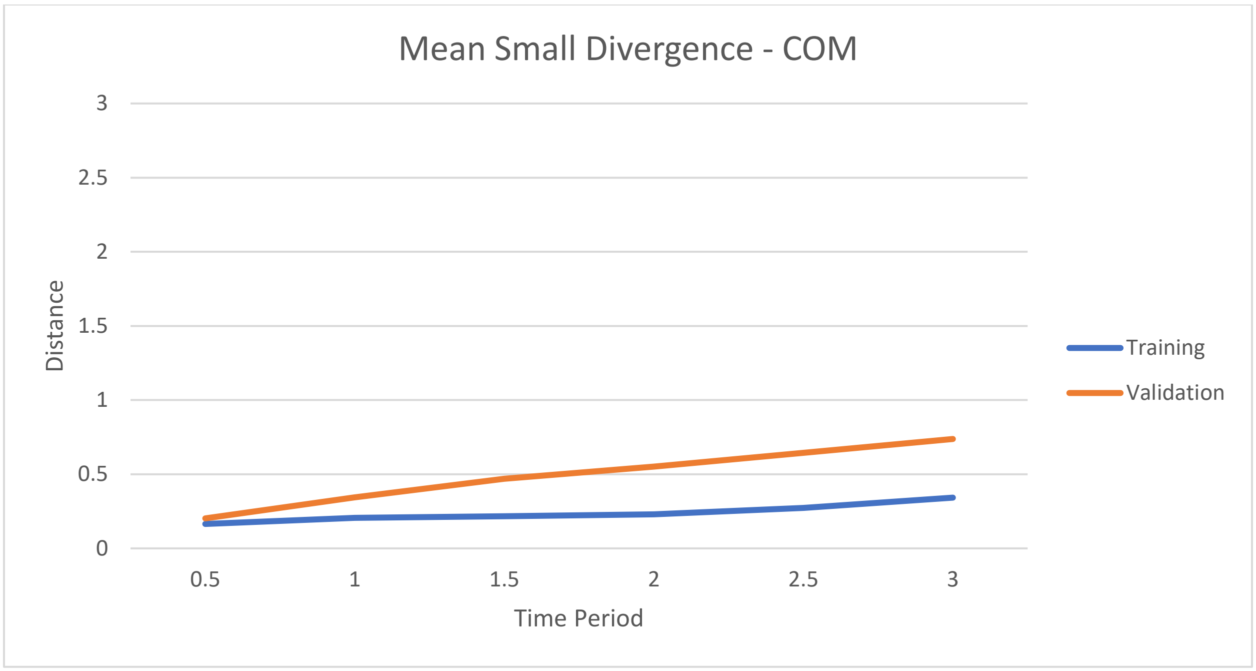


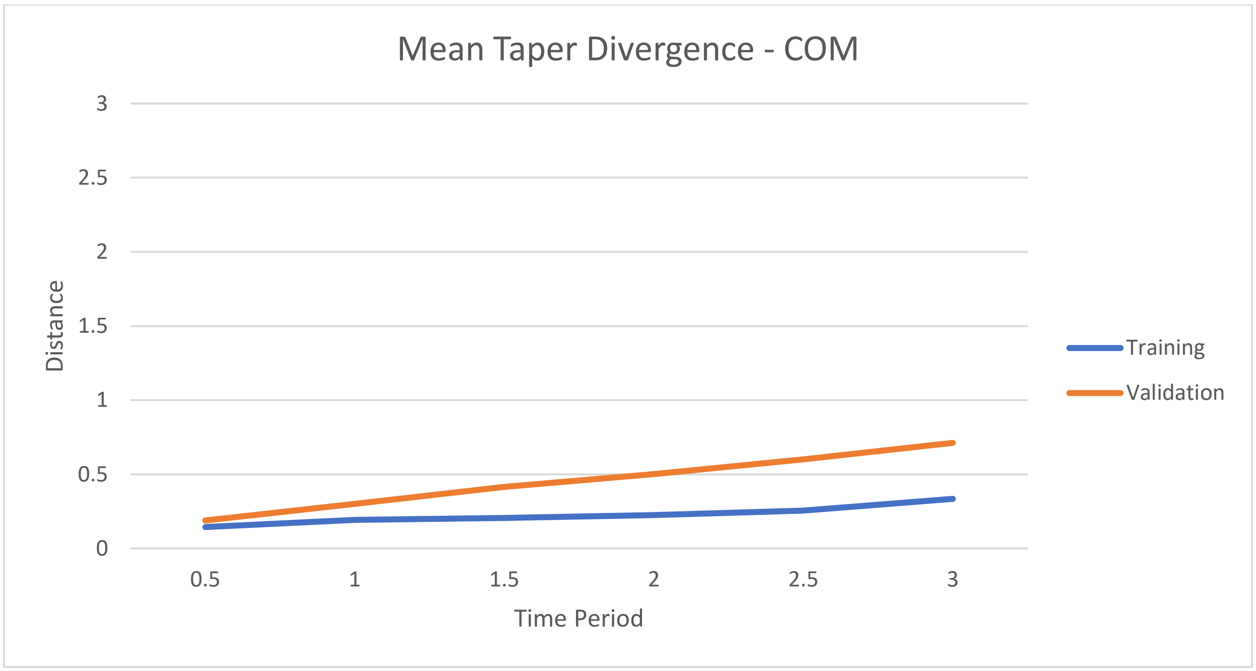

Mean Normal Divergence - COM
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Training
Validation

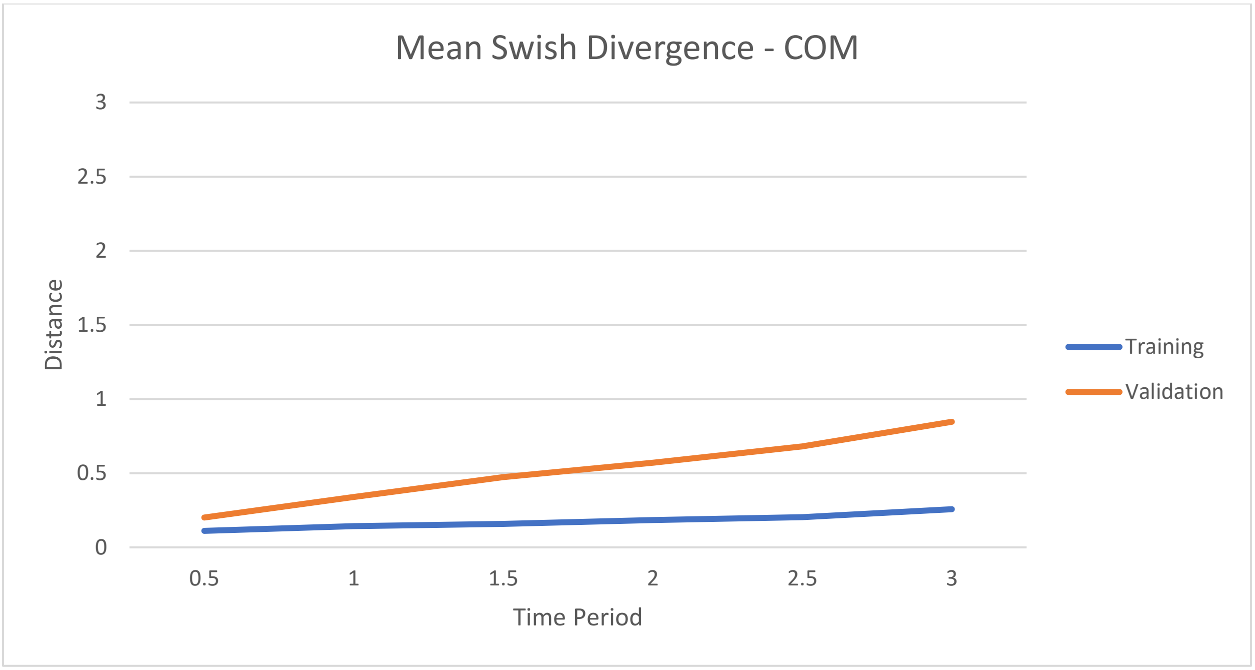

Mean L2 Divergence - COM
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Training
Validation

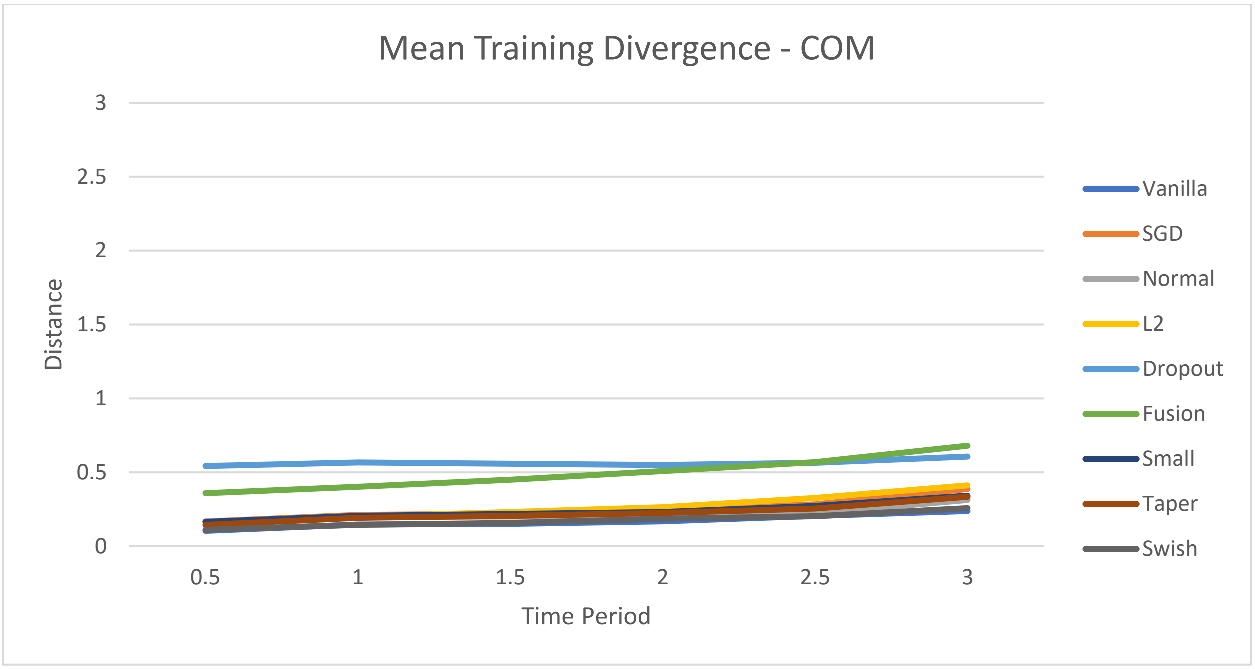

Mean Dropout Divergence - COM
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Training
Validation

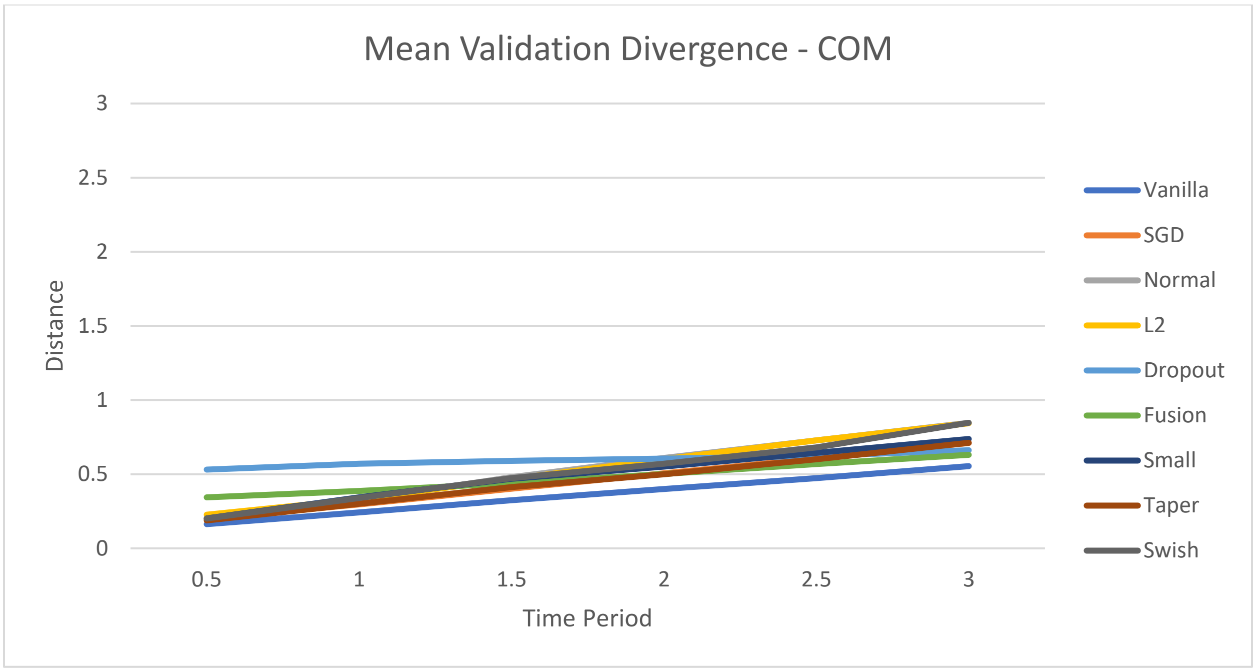

Mean Fusion Divergence - COM
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Training
Validation


Mean Small Divergence - COM
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Training
Validation


Mean Taper Divergence - COM
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Training
Validation


Mean Swish Divergence - COM
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Training
Validation


Mean Training Divergence - COM
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Vanilla
SGD
Normal
L2
Dropout
Fusion
Small
Taper
Swish


Mean Validation Divergence - COM
Distance
3
2.5
2
1.5
1
0.5
0
0.5
1
1.5
2
2.5
3
Time Period
Vanilla
SGD
Normal
L2
Dropout
Fusion
Small
Taper
Swish

**Matched-pairs T-tests on Vanilla Validation**

Time Point 0.5

| t-Test: Paired Two Sample for Means | | | t-Test: Paired Two Sample for Means | | |
|---|---|---|---|---|---|
| | | | | | |
| | *Variable 1* | *Variable 2* | | *Variable 1* | *Variable 2* |
| Mean | 0.003715828 | 0.001529637 | Mean | 0.020936711 | 0.038651552 |
| Variance | 1.653129821 | 1.612274082 | Variance | 1.551122376 | 1.504495435 |
| Observations | 3988 | 3988 | Observations | 3984 | 3984 |
| Pearson Correlati | 0.970978421 | | Pearson Correlati | 0.973123236 | |
| Hypothesized Me | 0 | | Hypothesized Me | 0 | |
| df | 3987 | | df | 3983 | |
| t Stat | 0.44788734 | | t Stat | -3.893546966 | |
| P(T<=t) one-tail | 0.32712942 | | P(T<=t) one-tail | 5.02036E-05 | |
| t Critical one-tail | 1.645235901 | | t Critical one-tail | 1.645236285 | |
| P(T<=t) two-tail | 0.65425884 | | P(T<=t) two-tail | 0.000100407 | |
| t Critical two-tail | 1.960559164 | | t Critical two-tail | 1.960559762 | |

Time Point 1.0

| t-Test: Paired Two Sample for Means | | | t-Test: Paired Two Sample for Means | | |
|---|---|---|---|---|---|
| | | | | | |
| | *Variable 1* | *Variable 2* | | *Variable 1* | *Variable 2* |
| Mean | -0.004478069 | -0.006284093 | Mean | 0.030798319 | 0.037828274 |
| Variance | 1.526559053 | 1.462842778 | Variance | 1.494391017 | 1.361605051 |
| Observations | 3984 | 3984 | Observations | 3984 | 3984 |
| Pearson Correlati | 0.899126112 | | Pearson Correlati | 0.880406705 | |
| Hypothesized Me | 0 | | Hypothesized Me | 0 | |
| df | 3983 | | df | 3983 | |
| t Stat | 0.207377865 | | t Stat | -0.756237329 | |
| P(T<=t) one-tail | 0.417862685 | | P(T<=t) one-tail | 0.22477582 | |
| t Critical one-tail | 1.645236285 | | t Critical one-tail | 1.645236285 | |
| P(T<=t) two-tail | 0.835725369 | | P(T<=t) two-tail | 0.44955164 | |
| t Critical two-tail | 1.960559762 | | t Critical two-tail | 1.960559762 | |

Time Point 1.5

| t-Test: Paired Two Sample for Means | | | t-Test: Paired Two Sample for Means | | |
|---|---|---|---|---|---|
| | | | | | |
| | *Variable 1* | *Variable 2* | | *Variable 1* | *Variable 2* |
| Mean | -0.007042102 | -0.009749422 | Mean | 0.040486881 | 0.04058894 |
| Variance | 1.782093794 | 1.573483286 | Variance | 1.996993572 | 1.456997121 |
| Observations | 3984 | 3984 | Observations | 3984 | 3984 |
| Pearson Correlati | 0.737343736 | | Pearson Correlati | 0.662983802 | |
| Hypothesized Me | 0 | | Hypothesized Me | 0 | |
| df | 3983 | | df | 3983 | |
| t Stat | 0.181528938 | | t Stat | -0.005899811 | |
| P(T<=t) one-tail | 0.427980829 | | P(T<=t) one-tail | 0.497646477 | |
| t Critical one-tail | 1.645236285 | | t Critical one-tail | 1.645236285 | |
| P(T<=t) two-tail | 0.855961658 | | P(T<=t) two-tail | 0.995292954 | |
| t Critical two-tail | 1.960559762 | | t Critical two-tail | 1.960559762 | |

Time Point 2.0

| t-Test: Paired Two Sample for Means | | | t-Test: Paired Two Sample for Means | | |
|---|---|---|---|---|---|
| | | | | | |
| | *Variable 1* | *Variable 2* | | *Variable 1* | *Variable 2* |
| Mean | -0.001196128 | -0.004520942 | Mean | 0.059462942 | 0.047943801 |
| Variance | 2.491232822 | 1.899620298 | Variance | 3.168156456 | 1.769504056 |
| Observations | 3984 | 3984 | Observations | 3984 | 3984 |
| Pearson Correlati | 0.548266579 | | Pearson Correlati | 0.468530375 | |
| Hypothesized Me | 0 | | Hypothesized Me | 0 | |
| df | 3983 | | df | 3983 | |
| t Stat | 0.148190829 | | t Stat | 0.44093805 | |
| P(T<=t) one-tail | 0.441099835 | | P(T<=t) one-tail | 0.329640889 | |
| t Critical one-tail | 1.645236285 | | t Critical one-tail | 1.645236285 | |
| P(T<=t) two-tail | 0.88219967 | | P(T<=t) two-tail | 0.659281778 | |
| t Critical two-tail | 1.960559762 | | t Critical two-tail | 1.960559762 | |

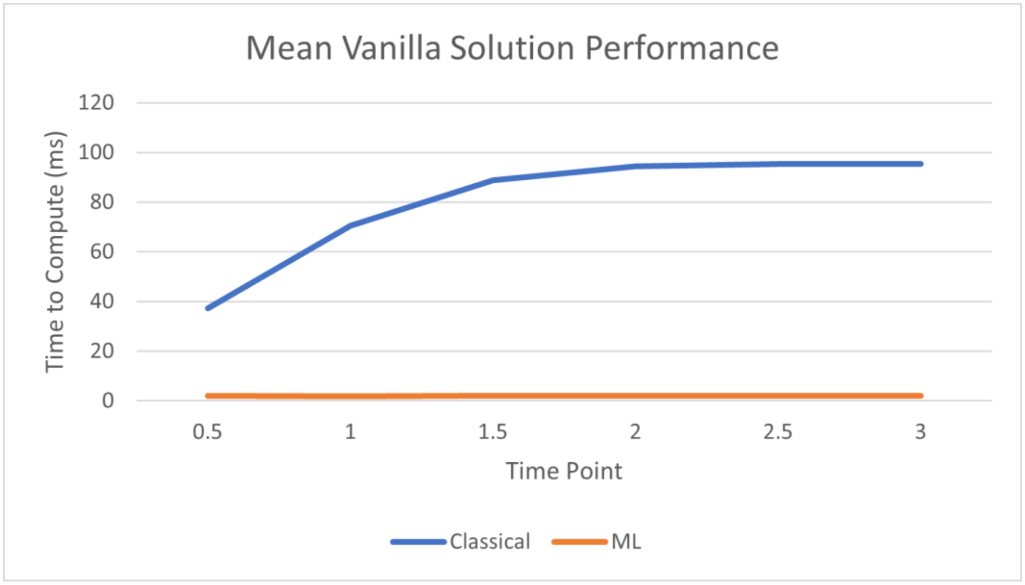
Time Point 2.5

| t-Test: Paired Two Sample for Means | | | t-Test: Paired Two Sample for Means | | |
|---|---|---|---|---|---|
| | | | | | |
| | *Variable 1* | *Variable 2* | | *Variable 1* | *Variable 2* |
| Mean | -0.005609234 | -0.015655289 | Mean | 0.073951672 | 0.059491815 |
| Variance | 3.646625742 | 2.429729463 | Variance | 4.994810762 | 2.269128723 |
| Observations | 3984 | 3984 | Observations | 3984 | 3984 |
| Pearson Correlati | 0.402920759 | | Pearson Correlati | 0.335873003 | |
| Hypothesized Me | 0 | | Hypothesized Me | 0 | |
| df | 3983 | | df | 3983 | |
| t Stat | 0.330650472 | | t Stat | 0.408067684 | |
| P(T<=t) one-tail | 0.370462954 | | P(T<=t) one-tail | 0.341622959 | |
| t Critical one-tail | 1.645236285 | | t Critical one-tail | 1.645236285 | |
| P(T<=t) two-tail | 0.740925909 | | P(T<=t) two-tail | 0.683245918 | |
| t Critical two-tail | 1.960559762 | | t Critical two-tail | 1.960559762 | |

Time Point 3.0

| t-Test: Paired Two Sample for Means | | | t-Test: Paired Two Sample for Means | | |
|---|---|---|---|---|---|
| | | | | | |
| | *Variable 1* | *Variable 2* | | *Variable 1* | *Variable 2* |
| Mean | -0.006515197 | -0.029307191 | Mean | 0.088824038 | 0.053352034 |
| Variance | 5.223436292 | 3.197411181 | Variance | 7.467627678 | 3.020874325 |
| Observations | 3984 | 3984 | Observations | 3984 | 3984 |
| Pearson Correlati | 0.299022479 | | Pearson Correlati | 0.241393424 | |
| Hypothesized Me | 0 | | Hypothesized Me | 0 | |
| df | 3983 | | df | 3983 | |
| t Stat | 0.588447812 | | t Stat | 0.782093841 | |
| P(T<=t) one-tail | 0.27813256 | | P(T<=t) one-tail | 0.217102958 | |
| t Critical one-tail | 1.645236285 | | t Critical one-tail | 1.645236285 | |
| P(T<=t) two-tail | 0.556265119 | | P(T<=t) two-tail | 0.434205916 | |
| t Critical two-tail | 1.960559762 | | t Critical two-tail | 1.960559762 | |

## Vanilla Performance



## Appendix B: Code

For all code employed by this project, refer to the GitHub repository at https://github.com/rylanandrews/ml-and-orbits.