Brigham Young University

Dynamic Optimization

Powered Descent and Landing of an Orbital-Class Rocket

Ethan Gunnell, Eric Mansfield, Don Rodriguez & Miriam Medina

April 24, 2019

Contents

1 Executive Summary **3**

2 Introduction **4**

3 Problem description **5**
- 3.1 Constant Parameters . . . 5
- 3.2 Dynamic Parameters . . . 6
- 3.3 Feasibility . . . 6
- 3.4 Uncertainties / Contingencies . . . 6

4 Method and Results **7**
- 4.1 Simulation . . . 7
- 4.2 GEKKO Model . . . 8
- 4.3 Parameter Estimation . . . 11
- 4.4 Control . . . 13
  - 4.4.1 Offline MPC . . . 13
  - 4.4.2 Online Estimation . . . 14
  - 4.4.3 Online MPC . . . 15

5 Discussion **17**

6 Acknowledgements **19**

7 Appendix A: Preliminary Step Tests **20**
- 7.1 GEKKO Step Test: Thrust only . . . 20
- 7.2 GEKKO Step Test: Thrust and Gimbal . . . 21
- 7.3 GEKKO Step Test: Thrust and Gimbal . . . 22

8 Appendix B: Source Code **23**

9 References **24**

List of Figures

1 The Falcon 9 rocket after a successful landing. 3
2 Rocket diagram describing the parameters and variables for the system. . . 5
3 A screen shot from the Python simulation. 8
4 Results from a step-and-hold test of the Yaw manipulated variable after optimizing K_D and K_L. 11
5 Results from a binary step test of the Yaw manipulated variable after optimizing K_D and K_L. 12
6 Results from a step test of $Throttle$. 13
7 Optimized trajectory and manipulated variables Yaw, Pitch, and Throttle, of the rocket as predicted by the offline controller. 14
8 Comparison of exact versus estimated positions and velocities for a typical landing simulation. 15
9 Flowchart 16
10 Trajectory of the simulated rocket as guided by the online model predictive controller. 17
11 No-thrust step test. 20
12 Thrust-only step test. 21
13 Thrust and gimbal first step test. 22
14 Thrust and gimbal second step test. 23

List of Tables

1 Summary of MVs and respective ranges 10

1 Executive Summary

We created an algorithm for precise, powered descent and landing of an orbital-class rocket to a targeted landing zone. Using Python and the Bullet physics engine, we developed a physical simulation of the Falcon 9 booster. The simulation also estimates instantaneous vehicle position and velocity from a simulated stream of GPS position data. Wind is also present as a disturbance.

We also created an accompanying GEKKO model using simplified forms of the simulation equations. Both the simulation and model account for drag, lift, and thrust forces. Because the model represents a semi-empirical version of the more complex simulation, we performed step steps of the manipulated variables, and used batch parameter regression to fit the model parameters.

The controller which guides the rocket to the landing zone works on two levels; first, guidance via model predictive control which optimizes the desired yaw, pitch and thrust of the rocket; second, control via two PID loops which adjust the grid fins and engine thrust vectoring to achieve the desired yaw and pitch. With the controller running simultaneously with the simulation, we achieved a precise landing of the booster.

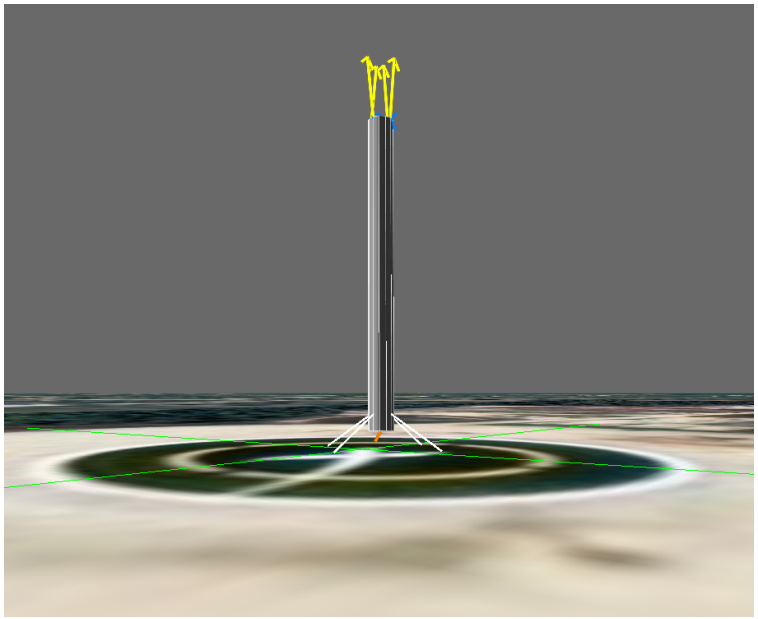

Figure 1: The Falcon 9 rocket after a successful landing.

2 Introduction

Normally, rockets are used once and discarded. Rapid recovery and reuse of orbital rockets is a key to lowering the cost of access to space. One technique that has been developed successfully by Space Exploration Technologies (SpaceX) and Blue Origin is powered, controlled descent of the booster stage to a landing zone. The non-triviality of the problem means that a successful algorithm has eluded government space programs for decades, and only recently has the problem been solved by these commercial companies.

We plan to replicate this feat by creating a controller that can accept simulated inputs from GPS receivers, gyros, and other instruments, that can perform a powered descent and landing of a rocket to a precise landing target from a variety of initial states, while being subjected to random disturbances. As part of the deliverable, we will include a 3D animation of a sample landing generated from the simulator telemetry. The animation will overlay thrust, drag, and lift vectors, along with readouts of measured, manipulated, and controlled variables, as well as estimated disturbance variables.

Although these rockets carry onboard autonomous controllers to guide them through all phases of launch, boostback, entry, descent, and landing, we are considering only the descent and landing phases. We believe these two phases alone offer enough complexity to be worthy of our attention. Although the controller could be extended generally to other phases–stitching together multiple pieces to simulate an entire mission, for instance–we don't think this would add significantly to the educational merit of the problem.

The difficulty of the problem is this: Using only six manipulated variables, corresponding to engine thrust, engine thrust vectoring, and grid fins (control surfaces), the controller must zero out 12 state variables (position, rotation, and their derivatives) while ensuring that a thirteenth variable, propellent mass, does not reach zero. Controlling 13 variables with only six means that the final state must depend not only on the present values of the manipulated variables, but their time evolution as well. No steady-state solutions exist–the rocket is always in motion–and the controller has one chance to land the rocket without rapidly disassembling it in the process.

3 Problem description

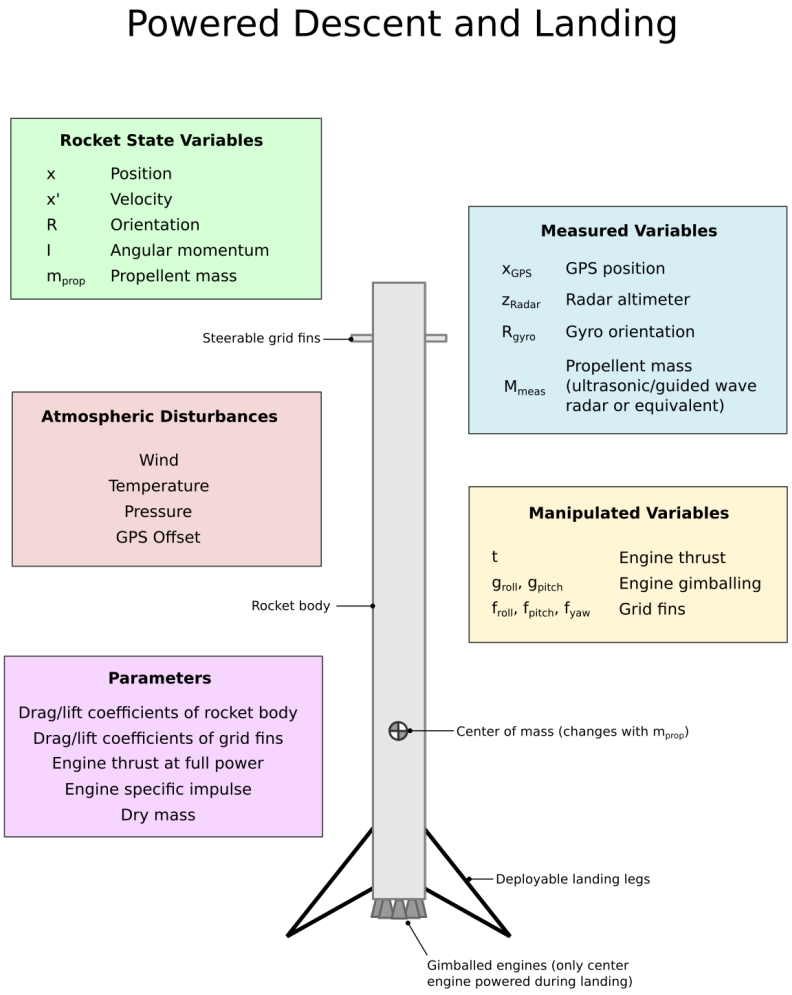

Figure 2: Rocket diagram describing the parameters and variables for the system.

3.1 Constant Parameters

We identified the following immutable factors that influence the dynamic outcome (constants / parameters):

Drag coefficient of rocket body	Lift coefficient of rocket body
Drag coefficient of grid fins	Lift coefficient of grid fins
Engine thrust at full power, sea level	Engine thrust at full power, vacuum
Engine specific impulse at sea level	Engine specific impulse in vacuum
Dry mass of rocket	

3.2 Dynamic Parameters

We identified the following factors that change throughout the descent and landing phase to influence the dynamic result (degrees of freedom or manipulated variables). These are the control inputs:

Engine power or thrust
Engine gimbal (x, y)
Grid fin position, one for each fin (xpos, xneg, ypos, yneg)

3.3 Feasibility

Each of the rocket's flight control elements has a finite range of operation. The vehicle we are modeling after, SpaceX's Falcon 9, does not have published specifications for these control limits. Some of these limits can be guessed or determined from photographs. We have chosen representative values that should result in a good test of our control algorithm.

The engine thrust vector control (gimballing) can rotate the engine up to a maximum of 7 degrees on two axes at 7 deg/s. The engine itself can be throttled from 57% to 100% of its full thrust, or can be shut down completely. Each of the four grid fins can rotate up to 30 degrees in either direction. There is no theoretical limit to the rate of change of the other state variables, although practical limits might necessitate adding penalty terms to the objective function for excessively fast vehicle rotation.

We originally wrote the GEKKO model to be as close of an analog to the simulation as possible. During the course of testing the controller, however, it became apparent that the model was not solving quickly enough to run in real-time. Because of this, we simplified the model, and then split the problem into two cascading control problems: the first in which the controller solves for the optimal yaw and pitch of the vehicle in order to target the landing zone (navigation) and a second in which the controller solves for the various control inputs to achieve the desired yaw and pitch (control).

3.4 Uncertainties / Contingencies

To simplify the model, and to avoid the need to include rigorous three-dimensional rigid-body dynamics, we are assuming that the rocket will experience only small angles of attack (the difference between the vehicle pointing axis and the velocity vector) and that one side of the rocket will always face the same direction. This will allow us to treat the $x-$ and $y-$dimensions independently. (Sorry, no barrel rolls.)

Unfortunately, we do not have the means to implement this controller on a real rocket. But this is actually fortunate for us, as we do not have to consider or worry about uncertainties with hardware, logistics, weather, or the many other issues that would certainly arise. Using a simulated environment, we can quickly iterate on controller tuning parameters, and crash as many rockets as necessary from the safety of our computers.

4 Method and Results

In the following subsections, we walk through the process phases we took to actually land a model of the Falcon 9 booster in simulation. First, we discuss the steps we took to build the Falcon 9 simulation, also known as a "digital twin." Creating a simulation allowed us to later improve our model parameters without physically building a rocket and enabled us to run a myriad of experiments, which saved us time. Second, we describe how we attempted to match the "digital twin" with a GEKKO model, identifying manipulated variables (MVs) and the desired outputs or control variables (CVs). Third, we describe our model parameter estimation process to increase fidelity to the "digital twin." Finally, we discuss our methods for producing a controller that accurately lands the Falcon 9 model in simulation.

4.1 Simulation

Using Python, the Panda3D rendering engine, and the Bullet physics engine, we developed a realistic physical simulation, or "digital twin," of the Falcon 9 booster. The physics engine handles linear and rotational dynamics out-of-the-box, and we added our own lift, drag, and thrust forces. Wind can also be simulated. The simulation also includes forces for aerodynamic manipulators (grid fins). As will be described below, the MPC outputs a desired yaw and pitch for the rocket. The simulation then uses two PID-controllers (one each for yaw and pitch) that adjust the grid fins to achieve the desired yaw and pitch. A third PID controller is also active that keeps the roll of the rocket at zero degrees–a necessity when separating the x- and y- coordinates in the GEKKO model.

The simulation is capable of beginning at any chosen initial condition. It can run a predetermined step test consisting of yaw, pitch, and throttle inputs, or it can accept inputs in real-time from the online MPC controller. Figure 3 shows a screen shot from the simulation.

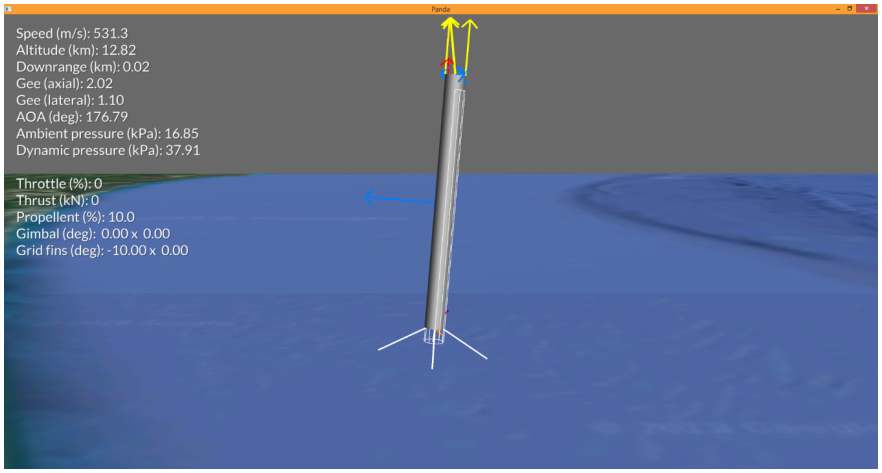

Figure 3: A screen shot from the Python simulation. The particular test case being run was a step test of the grid fins. Yellow vectors show the grid fin surface normals. Blue vectors are the lift generated by the rocket body and the four grid fins. The red vector originates from the center of the rocket and represents drag. Vectors that are not visible are gravity (purple) and thrust (orange). Landing legs are shown for visualization purposes only; they do not contribute to the physics.

4.2 GEKKO Model

The GEKKO model is a simplified form of the simulation. We built the model for the purposes of eventually becoming an estimator and controller. Our first attempt resulted in a very accurate model, but the computation time required for the model to converge was too great. Some of the initial step tests we performed for this early version of the model are shown in Appendix A.

We then created a second, simpler version of the model. Many assumptions were made to simplify some equations, and eliminate others altogether. First, we assumed that one side of the rocket always faces the same direction. This is accomplished by a PID controller that runs in the simulation and keeps the roll of the rocket at zero degrees. This allows us to simplify the GEKKO model by removing a degree of freedom, and gives the added benefit of being able to separate the x- and y-dimensions of control.

Second, we assumed the angle of attack, the angle between the velocity vector and the rocket's z-axis, is small. This allows us to use substitutions such as x for $\sin x$ and 1 for $\cos x$ to linearize the GEKKO model and improve computational speed.

Third, we removed grid fins and engine thrust vectoring as manipulated variables, and also removed the angular dynamics. We made Yaw and Pitch manipulated variables, and offloaded control of the grid fins and engine thrust vectoring to the simulation.

The model includes the basic equations of motion below, where $\vec{x}$ is the position of the

rocket, $\vec{v}$ is velocity, and $\vec{a}$ is acceleration:

$$\frac{d\vec{x}}{dt} = \vec{v}, \tag{1}$$

$$\frac{d\vec{v}}{dt} = \vec{a}, \tag{2}$$

$$\vec{a} = \vec{g} + \frac{\vec{D} + \vec{T} + \vec{L}}{m}, \tag{3}$$

In equation 3, $\vec{g}$ is the acceleration due to gravity, $\vec{D}$ is the drag force, $\vec{T}$ is thrust, $\vec{L}$ is lift, and m is the mass of the rocket and propellant. The drag force is given by:

$$\vec{D} = -P_{dyn} A K_D \hat{v}_{rel} \tag{4}$$

$$\hat{v}_{rel} = \frac{\vec{v} - \vec{W}}{\left\| \vec{v} - \vec{W} \right\|} \tag{5}$$

Where $\hat{v}_{rel}$ is the normalized velocity of the rocket relative to the air, P_{dyn} is the dynamic pressure $\frac{1}{2}\rho\,(\vec{v} \cdot \vec{v})$, A is the cross-sectional area of the booster in the airstream, K_D is an adjustable parameter, and $\vec{W}$ is wind. The cross-sectional area A is estimated from the angle of attack AOA of the rocket as $A = 10.8 + 163.5 * AOA$, with A in square meters and AOA in radians. The angle of attack is defined as the angle between $\hat{v}_{rel}$ and the vertical axis of the rocket. AOA can be estimated efficiently by defining two *pointing factors*, P_x and P_y. The angle of attack is then approximately equal to $\sqrt{{P_x}^2 + {P_y}^2}$.

$$\begin{aligned} P_x &= v_{rel,x} + Yaw, \\ P_y &= v_{rel,y} + Pitch, \end{aligned}$$

In the GEKKO model, Yaw and $Pitch$ are configured as *manipulated variables* (MVs). This means that Yaw and $Pitch$ are fixed at each timestep when the model is used for simulation, and they become degrees of freedom when the model is used for MPC. Treating Yaw and $Pitch$ in this way greatly simplifies the model, and removes rotational dynamics entirely.

The lift $\vec{L}$ defined in the model is simplified compared to the simulation, and also includes an adjustable parameter K_L. The effect of the lift term in equation 3 is to try and "straighten out" the rocket by applying a sideways force to the side of the rocket that is being hit by the incoming airstream:

$$\begin{aligned} L_x &= -P_x P_{dyn} K_L \\ L_y &= -P_y P_{dyn} K_L \\ L_z &= 0 \end{aligned}$$

The thrust in the model is also simplified compared to the simulation. In the simulation, the engine is gimballed so that thrust may be vectored to exert a torque on the rocket. A small lateral force is also introduced, but it is small compared to the main, vertical component of the thrust. Because the model excludes rotational dynamics, the gimbaling feature was deemed unnecessary for the model. Instead, the modeled thrust $\vec{T}$ only includes the principle Z-component of thrust, rotated to align with the rocket's Z-axis:

$$\begin{aligned} T_x &= T_{eng} \cdot Yaw \\ T_y &= T_{eng} \cdot Pitch \\ T_z &= T_{eng} \cdot \sqrt{1 - Yaw^2 - Pitch^2} \end{aligned}$$

Here, T_{eng} is the thrust produced by the engine, which varies linearly depending on the external atmospheric pressure p. Below, T_{SL} is the maximum thrust produced by the engine at sea level, T_{vac} is the thrust in a vacuum, and p_{std} is standard atmospheric pressure at sea level:

$$T_{eng} = Throttle \cdot \left(T_{SL} \frac{p}{p_{std}} + T_{vac} \left(1 - \frac{p}{p_{std}} \right) \right) \tag{6}$$

Finally, at full throttle propellant is consumed at a rate of 300 kg/s, so the mass m of the rocket decreases over time according to the equation:

$$\frac{dm}{dt} = -300 \cdot Throttle \tag{7}$$

$Throttle$ is the third manipulated variable in the model, and can vary between 0.57 and 1. Like Yaw and $Pitch$, it is adjusted by the controller to minimize the value of an objective function. The objective is to reach the landing target at the precise moment that all velocities become zero. Without loss of generality, we can position the landing target at the origin, so that all components of both the position and velocity should be zero when the objective is met. The controller uses the ℓ_2-norm (sum of squares) objective at the final point of the trajectory to perform the minimization:

$$\Omega = \|\vec{x}\| + \|\vec{v}\| \tag{8}$$

The optimization problem then becomes the following:

$$\min_{Yaw, Pitch, Throttle} \Omega(t_f), \tag{9}$$

subject to the equations above.

Table 1: Summary of MVs and respective ranges

$Yaw \in [-30°, 30°]$
$Pitch \in [-30°, 30°]$
$Throttle \in [57\%, 100\%]$

4.3 Parameter Estimation

To estimate the GEKKO model adjustable parameters K_D and K_L, we ran the simulation beginning at various initial conditions, applying step tests wherein we varied Yaw, $Pitch$, and $Throttle$. Then we performed the same step tests using the GEKKO model, and allowed K_D and K_L to vary until the model agreement was satisfactory. Figures 4 and 5 show the results of two of the Yaw step tests. (Step tests for $Pitch$ are similar and are not shown.)

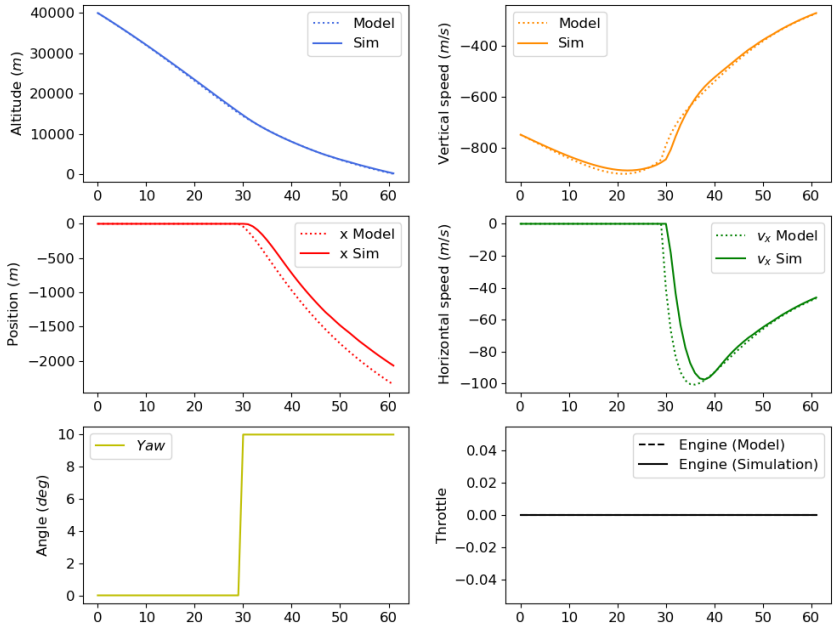

Figure 4: Results from a step-and-hold test of the Yaw manipulated variable after optimizing K_D and K_L. The solid lines are the results from the simulation; the dotted lines are the model predictions.

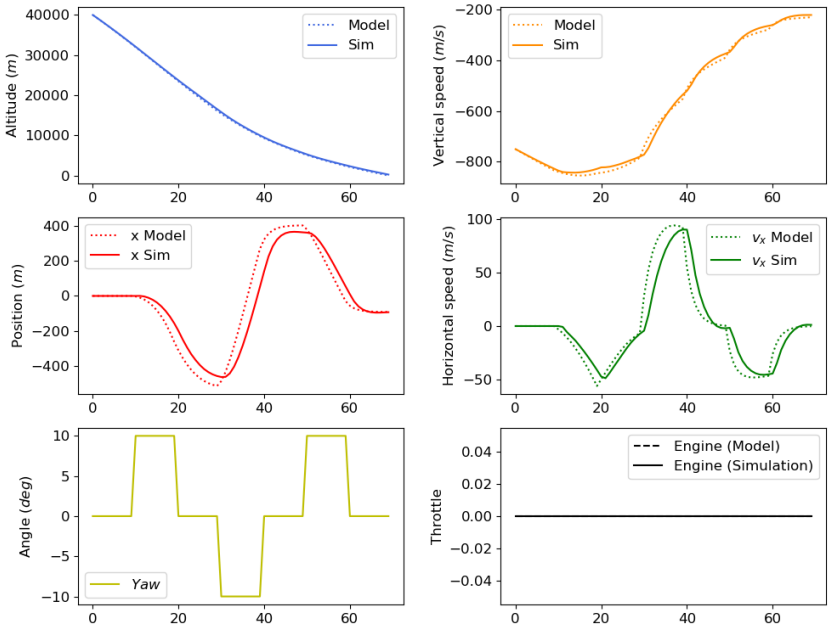

Figure 5: Results from a binary step test of the Yaw manipulated variable after optimizing K_D and K_L. The solid lines are the results from the simulation; the dotted lines are the model predictions.

In Figures 4 and 5, note that there is a difference in the speed of the response to changes in Yaw and $Pitch$. This is because the model assumes that when Yaw or $Pitch$ changes, the rocket's orientation changes instantaneously. However, the simulation uses a PID controller to adjust the orientation, so there is a delay in the response time. This is the cause of the discrepancy. What is more important, however, is the magnitude of the response to changes in Yaw and $Pitch$. As long as the magnitude of the response is in agreement, the controller may be sluggish, but it will not cause the rocket to undershoot or overshoot the landing zone.

Figure 6 shows a step test for $Throttle$, but no adjustments were necessary because the model equations already matched the simulation very closely:

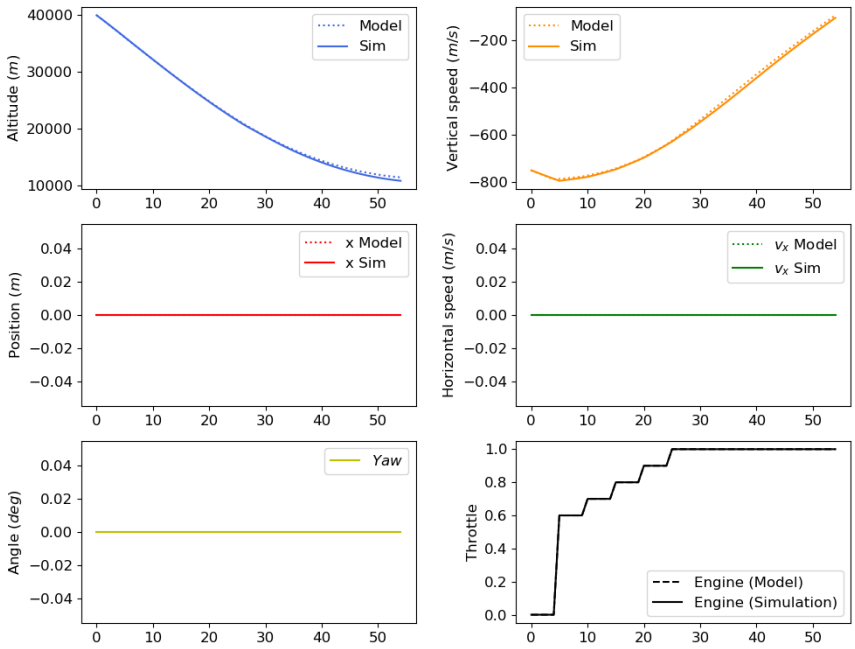

Figure 6: Results from a step test of *Throttle*. The solid lines are the results from the simulation; the dotted lines are the model predictions.

Why did we not add a time delay term into the model for changes in Yaw and *Pitch*? For the precise reason that it would over-complicate the model. As we will show below, the online controller performed satisfactorily despite this minor model disagreement.

4.4 Control

4.4.1 Offline MPC

With the model parameters fit to the simulation results, we moved on to attempting to control the rocket's descent. As a first test, we solved the model predictive controller offline to generate a single horizon of optimal Yaw, Pitch, and Throttle manipulated variables. For this and all subsequent tests, we used a set of initial conditions taken from SpaceX's NROL-76 mission [5]. The results of the offline controller are shown in Figure 7.

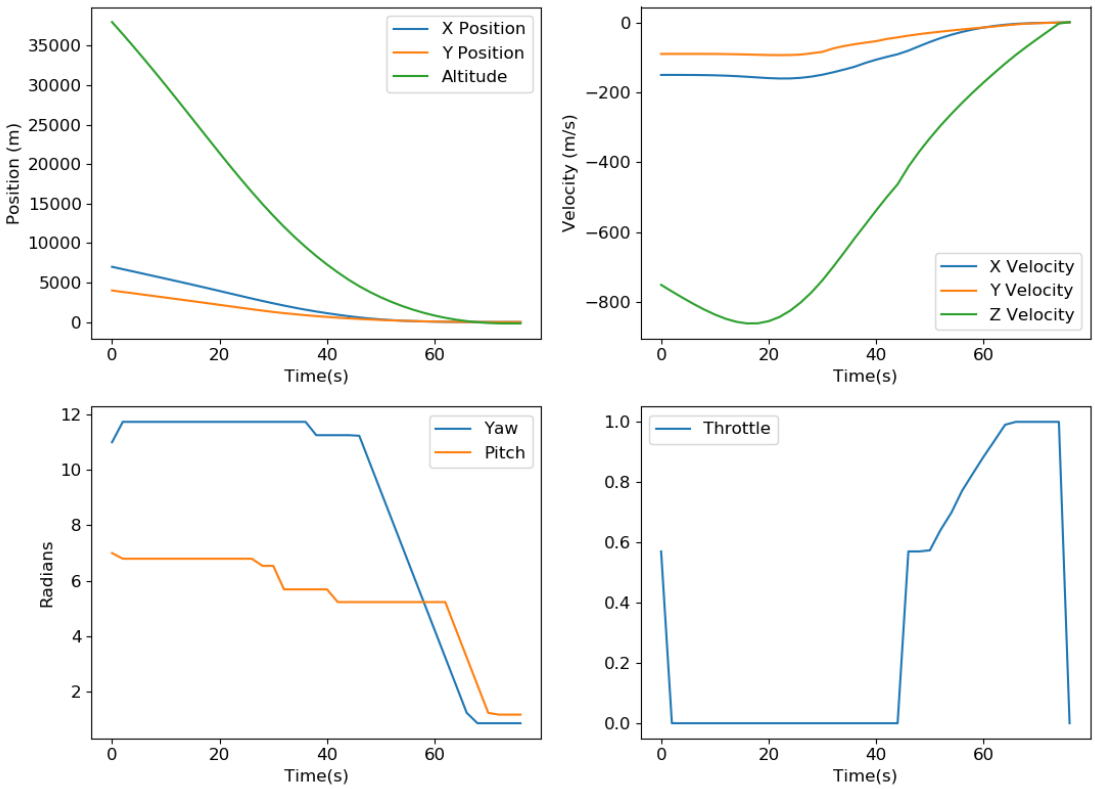

Figure 7: Optimized trajectory and manipulated variables Yaw, Pitch, and Throttle, of the rocket as predicted by the offline controller.

Note that the position and velocity of the rocket both converge to zero at the end of the time horizon, showing that the controller met the objective.

To verify that the controller was working correctly and that the model was providing a good prediction of the simulation, we then replayed the offline controller's outputs during a simulation, beginning with the same initial conditions. If the model were to perfectly match the simulation, the replayed manipulated variables should steer the rocket directly to the landing zone for a perfect landing. In actuality, the rocket missed the landing zone by several hundred meters. A recording of this simulation is available at https://youtu.be/TTQgU7i9S1M.

4.4.2 Online Estimation

Actual rockets use GPS and/or accelerometer data to obtain their position. This presents a number of issues for guidance controllers. GPS sensors typically report position, but not velocity. In addition, GPS acquisition rates can be too slow to rely on for rapid control.

In order to mimic these conditions, we altered the simulation so that position was acquired from a virtual GPS device at a rate of 2 samples/sec. Velocity data were not acquired. To estimate velocity, the x-, y-, and z-coordinates of 5 consecutive positions were each fitted to a 2nd-order polynomial using NumPy's polyFit method. The 2nd-order form allows the

model to account for constant acceleration during the estimation time horizon. The velocity at any moment can then be obtained through differentiation. Because the GPS data might be out-of-date by a fraction of a second, the polynomial function is extrapolated out to the current simulation time to obtain an estimate of the instantaneous position and velocity. These estimates are then fed into the controller.

We ran a number of landing simulations using this online estimation method. Figure 8 shows, for a typical landing, a comparison between the estimated telemetry of the rocket and the instantaneous telemetry, the latter having been obtained "illegally" from the physics engine. The estimation error is small and did not adversely affect the controller performance.

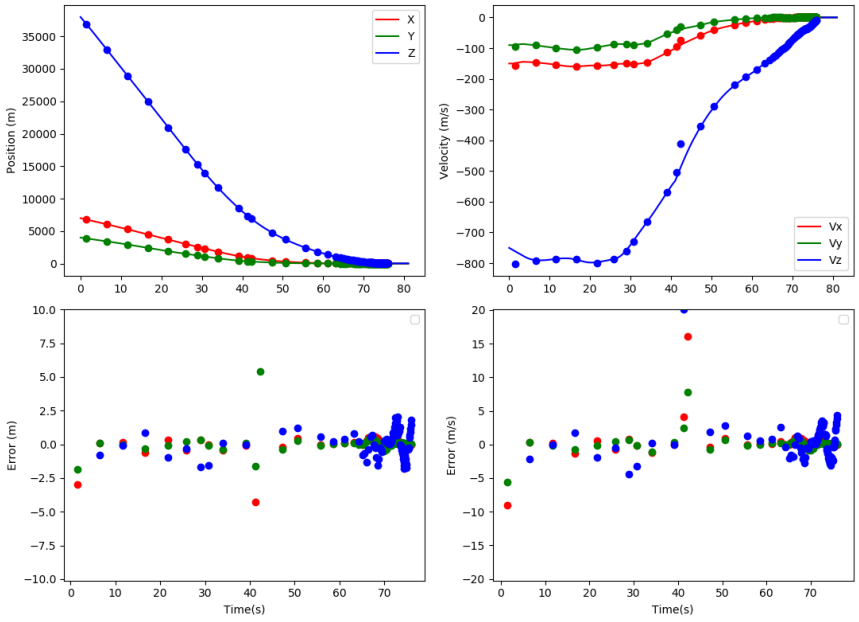

Figure 8: Comparison of exact versus estimated positions and velocities for a typical landing simulation. Solid lines are the exact positions and velocities; points are the estimated positions and velocities each time the controller began to solve. Errors are shown below with outliers removed.

4.4.3 Online MPC

In order to precisely land the rocket on the landing zone, we ran the model predictive controller (MPC) as a second execution thread in tandem with the simulation. The second thread enabled the MPC to solve while the simulation kept running. Essentially, both the controller and the simulation fed updated results to one another, working as a team. Figure 9 shows how the simulation and controller work together:

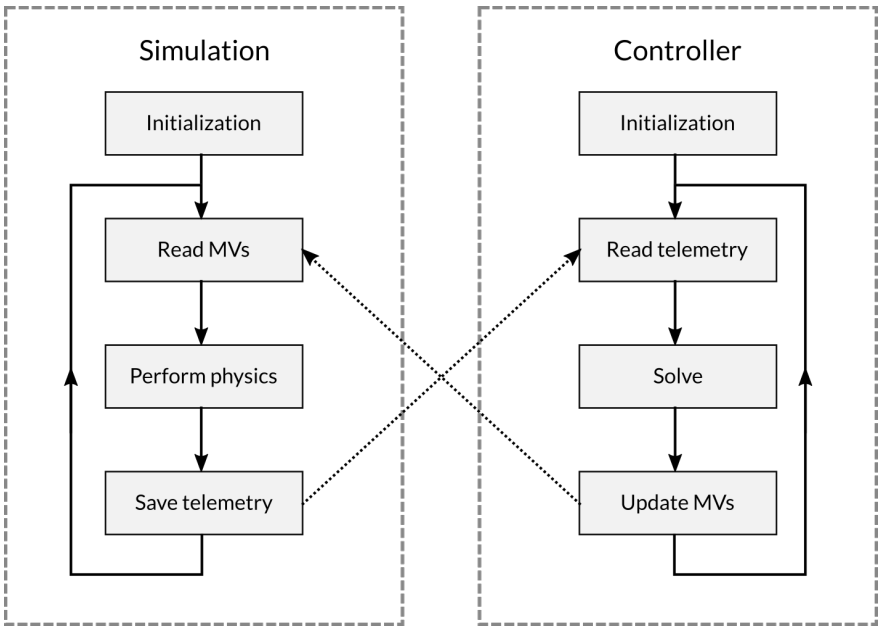

Figure 9: Flowchart

1. Initial conditions are read from a file.
2. The controller cold starts and performs an initial solve, generating optimal manipulated variables for the entire time horizon.
3. The simulation is initialized.
4. The simulation begins running, following the optimal manipulated variables that were calculated by the controller.
5. The controller now solves again in a separate thread, reading the current telemetry from the simulation to set the initial model values.
6. When the controller finishes solving, it updates the optimal manipulated variables.

By updating the optimal manipulated variables after each solve, the simulation is assured to always be following the most up-to-date and accurate predictions. Each time the controller solves, it predicts the entire trajectory until the rocket lands. When the rocket is close to landing, the time discretization is subdivided to increase the accuracy of the solution. Successive solutions converge faster and faster as the calculated time horizon becomes shorter, resulting in a precise touchdown.

To account for solver errors, a maximum solving time of 5 seconds is imposed on the controller. If this time is exceeded, the controller aborts and restarts using updated telemetry from the simulation. The rocket continues to follow the optimal manipulated variables from the last successful solution.

Figure 10 shows the trajectory of the simulated rocket being guided by the online controller. For this simulation, we added wind with an average velocity of 5 m/s from the west. The controller successfully guided the rocket to the landing zone and zeroed its velocity.

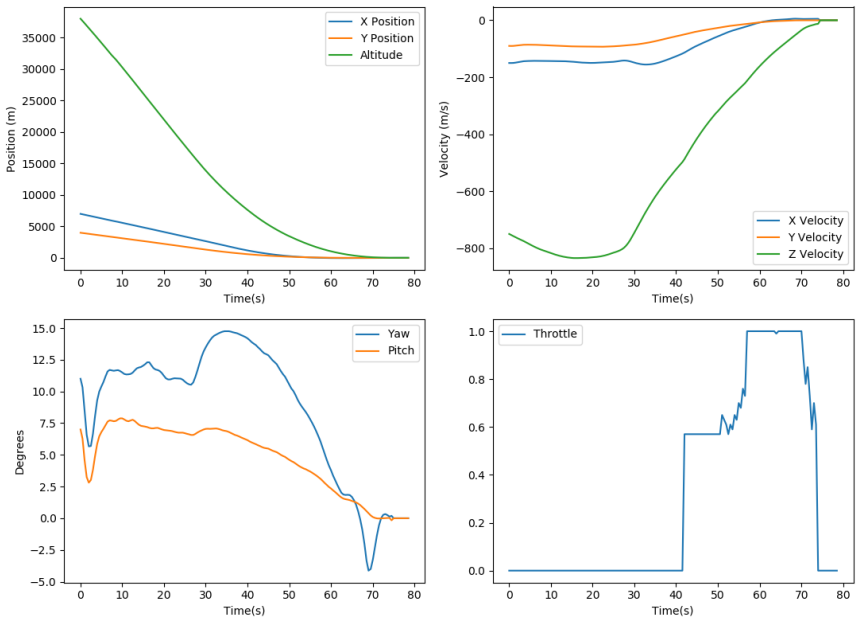

Figure 10: Trajectory of the simulated rocket as guided by the online model predictive controller.

Results vary from run to run due to the way the simulations are timed and integrated by the physics engine. In this particular case, after the initial solution, the controller was unable to find a solution until 30 seconds into the simulation. Until that time, the rocket followed the optimal manipulated variables precomputed by the controller before the simulation started. Once the controller found a solution, it determined the rocket was off course (due to model error) and adjusted the Yaw and $Pitch$ to put the rocket back on course for the landing zone. When the rocket got close to the landing zone, it actually tipped its nose into the wind slightly (negative Yaw) to zero out the horizontal velocity. A recording of a simulation using online MPC is available at https://youtu.be/4z9lpZLSbUk.

5 Discussion

Our first attempt to create a model resulted in a very accurate model which included rotational dynamics, and closely captured all the dynamic features of the simulation. However, we were not able to solve that model in a reasonable amount of time. This led to the creation of the simpler semi-empirical model, in which the rotational dynamics were excluded. This illustrates a recurring theme in numerical methods–the trade-off between speed and accuracy. In our case, because the controller solves so frequently, any deviations from the optimal path due to model error are quickly corrected. As long as the model has no translational

bias, the rocket's calculated path should converge on the landing target.

Vehicles like the Falcon 9 receive position data from GPS satellites. This data is neither continuous nor exact. Updates from GPS receivers may come too infrequently to rely on for instantaneous positioning, and a delay of only one tenth of a second could result in an error of tens, or even hundreds, of meters. Our solution to this was to feed past GPS position data into the model, and then solve for the estimated position and velocity at the present time. But we again ran into the issue of computation speed. Rather than use the GEKKO model to do this estimation, we created a very simple model–a quadratic function–and used least squares to quickly fit the GPS position data to this model. This gave us near instantaneous estimates of both position and velocity.

There is no pause or rewind allowed in rocketry. The time-critical nature of the guided descent problem presents a unique challenge, in regards to calculating an optimal path in a timely manner, and in getting all the pieces of the simulation and controller to work together realistically in real-time. Multi-threading was our solution to the latter. With the simulation and controller running on separate threads, both could run simultaneously, passing information back and forth as needed.

As far as the other challenge, calculating the optimal path in a timely manner, SpaceX has solved that problem by compiling customized solvers that are specially optimized to solve the guided descent problem. GEKKO and APMonitor, on the other hand, are generic solvers that, while they can solve any problem, incur overhead due to the extra time taken to set up each unique problem. In future work, we could turn our attention towards the solver itself to see if any of these kinds of optimizations could be made.

SpaceX and Blue Origin also have to deal with a multitude of disturbances, such as wind, temperature/pressure variations, and instrument or actuator error. Contingencies also are taken into account. It is reported, in fact, that a SpaceX booster knows the positions of buildings, aircraft, and other populated areas, and will steer clear of those areas in the event of a malfunction. In one of SpaceX's recent landings, a grid fin actuator malfunctioned, leading to a loss of control. Although the booster could not reach the landing zone, it was still able to make a soft landing in the water. This kind of contingency planning is far more sophisticated than our humble model is currently capable of, but in the future we could make improvements to the model or objective function to account for such failures.

Future work could also include additional refinements to the model and objective function. For example, we could adjust the objective function to achieve a softer landing, or a more precise landing in the presence of wind. We could also include a term to minimize the consumption of propellant, in case it is in low supply.

Another area of future work includes untethering ourselves from the virtual world and building a model rocket that implements powered descent and landing. This might entail figuring out how to create a mini and simplified Merlin engine that can perform thrusting and throttling. Guidance on the specifics of rocket engines and their chemistry may be sought from

George B. Sutton's Rocket Propulsion Elements and NASA's Chemical Equilibrium with Applications. Additionally, we might add a GPS receiver, a servos motor to actuate the grid fins, and a Rasberry Pi to run the MPC. This venture will demand us to learn new concepts, but will certainly be engaging and enlarge our appreciation for space exploration.

6 Acknowledgements

We would like to acknowledge Dr. Hedengren for his expertise and advise on this project.

7 Appendix A: Preliminary Step Tests

The step tests for the original GEKKO simulation can be found below. The first is a simulation of the rocket falling, without any control variable input. Initial velocities are given, but no thrust is used to slow the descent.

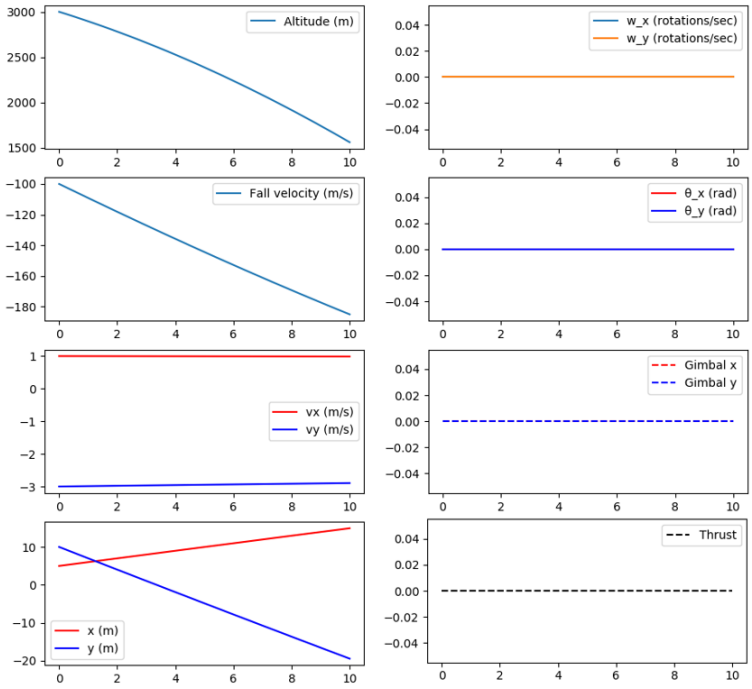

Figure 11: No-thrust step test.

7.1 GEKKO Step Test: Thrust only

The GEKKO model is thus far a physics simulator and thus does not include accurate literature constants such as the correct impulse. Some of these constants will be estimated in the next step of the project. The rocket was able to recover from its descent and gain altitude due to the large amount of thrust, but the impulse is expected to be much lower in actuality. Also notice some machine precision error in the rotation, as the graphs are still at 0 (1e-40).

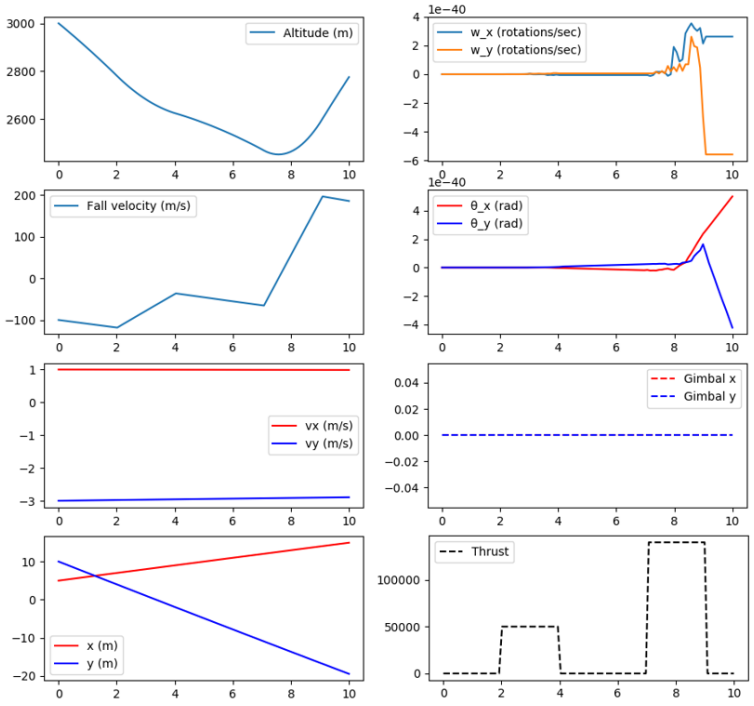

Figure 12: Thrust-only step test.

7.2 GEKKO Step Test: Thrust and Gimbal

The gimbal is the angle of the engine and can rotate or move the rocket. In this case, the gimbal was applied over a small timestep and resulted in a spinning rocket. The gimbal is a control variable but the torque produced by it is also dependant on the amount of thrust. Making a controller that can work with three control variables, and others included in the more complex model, will be a challenge.

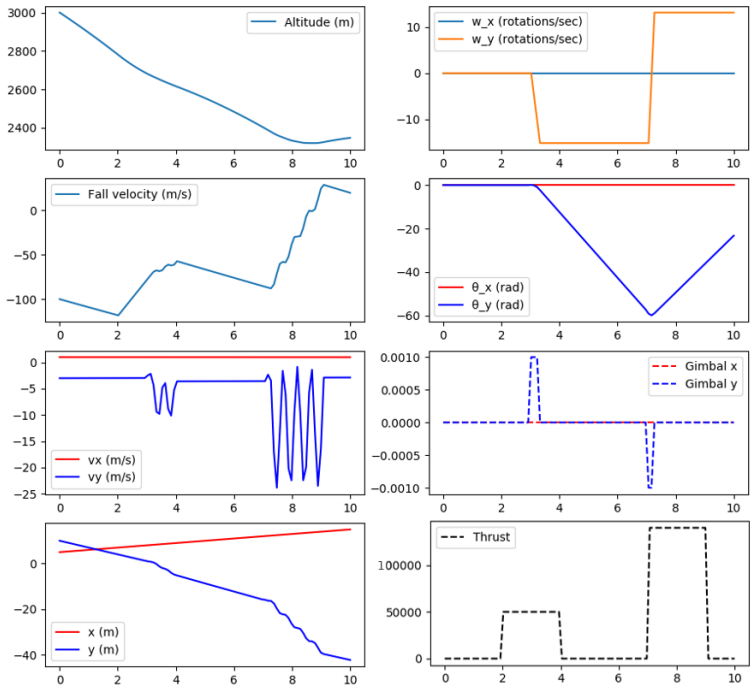

Figure 13: Thrust and gimbal first step test.

7.3 GEKKO Step Test: Thrust and Gimbal

A reduction in gimbal duration and magnitude shows that the direction of the rocket may be changed without the rocket spinning out of control. The adjustment is very fine because the thrust's magnitude is very large.

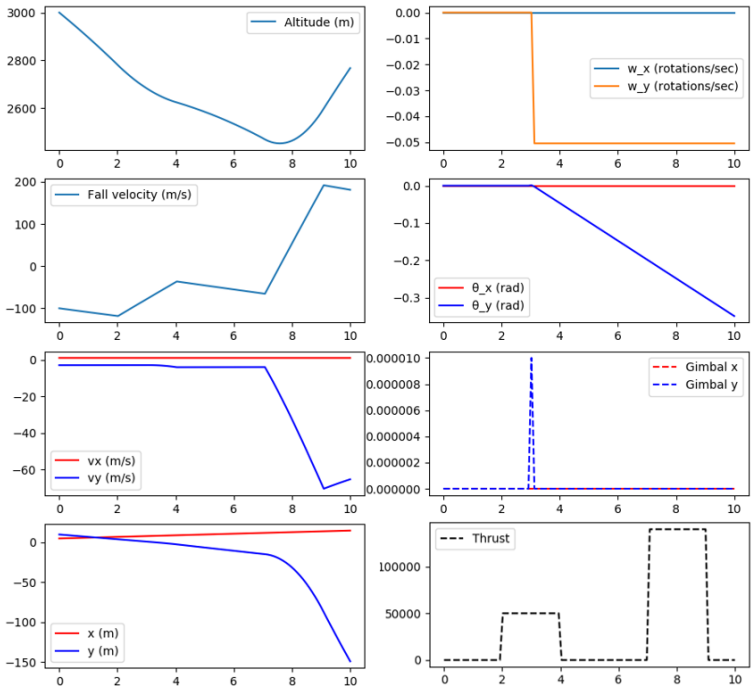

Figure 14: Thrust and gimbal second step test.

8 Appendix B: Source Code

Primary source:
GitHub Repository

Direct links to:
Gekko Model
Simulation
Controller

9 References

[1] N. Bakhtian and M. Aftosmis. Maximum Attainable Drag Limits for Atmospheric Entry via Supersonic Retropropulsion. In *Proceedings of the 8th International Planetary Probe Workshop*, 01 2011.

[2] L. Blackmore. Autonomous precision landing of space rockets. *Winter Bridge on Frontiers of Engineering*, 46:15–20, 01 2016.

[3] L. Blackmore, M. Ono, and B. C. Williams. Chance-constrained optimal path planning with obstacles. *IEEE Transactions on Robotics*, 27(6):1080–1094, Dec 2011.

[4] T. Ecker, S. Karl, E. Dumont, S. Stappert, and D. Krause. A Numerical Study on the Thermal Loads During a Supersonic Rocket Retro-Propulsion Maneuver. In *53rd AIAA/SAE/ASEE Joint Propulsion Conference, At Atlanta*, 07 2017.

[5] Shahar603. Telemetry-Data. A collection of telemetry captured from SpaceX Launch Webcasts. https://github.com/shahar603/Telemetry-Data/tree/master/NROL-76, 2019. Last accessed 21 April 2019.