Landing the Falcon 9: Control Methods to Achieve Vertical Landing of a 2D Rocket

David Lu
6-3 Undergraduate
davidlu@mit.edu

Michael Lu
6-2 Undergraduate, 2 Minor
mlu0708@mit.edu

Mindy Long
6-2 Undergraduate
mflong@mit.edu

***Abstract*—The Falcon 9 rocket, developed by SpaceX, is the first commercially successful rocket to achieve vertical landing of its booster via thrust vectoring. In this project, we seek to use underactuated control methods to land a simplified version of the Falcon 9's booster. First, we create a 2D simulation of the rocket dynamics that captures its translational and rotational degrees of freedom using Pydrake. Next, we implement a number of control methods to enable self-landing including linear–quadratic regulator (LQR) and trajectory optimization techniques based on direct transcription (DT) and direction collocation (DC). We also build two trajectory stabilizing controllers on top of DC: a model predictive control (MPC) controller, which iteratively solves a DC optimization problem, and LQR-stabilizing controller, which pushes the rocket toward a nominal trajectory generated by DC or DT. Our results show that, when compared against all other methods, our LQR trajectory stabilization approach with DC performs best and is most robust across initial rocket states. MPC with DC also shows significant promise but requires careful tuning to ensure that the optimizations run quickly enough while achieving sufficiently optimal solutions.**

I. Introduction

Rockets are cool. Flying rockets are cool. But what may be cooler than a rocket that can fly is a rocket that can land. In this work, we simulate the landing of a SpaceX Falcon 9 falling from space.

The Falcon 9 is the world's first reusable rocket as well as the only U.S. rocket currently certified to carry humans to the International Space Station. One key feature of the Falcon 9 is its partial reusability. The rocket consists of two stages: a first stage, known as the booster, and a second, smaller stage. Shortly after launch, the first stage detaches from the second stage. While the second stage goes on to lift the rocket's payload to its destination, the first stage descends back to Earth and performs an upright landing by using thrust vectoring—changing the angle and magnitude of its engine thrust—to steer itself to a landing pad. This allows the Falcon 9 to be reused and also serves as an interesting control problem for engineers to solve. In this project, we propose several ways to land a Falcon 9 booster using underactuated control algorithms.

In our research, we explore various methods for controlling rocket systems. Among the methods we investigate, LQR emerges as the fastest approach, known for its computational efficiency. However, we discover that LQR's performance diminishes when faced with more challenging scenarios, rendering it unsuccessful in such cases.

Another method we examine is the use of direct transcription for trajectory optimization. Unfortunately, we find that this approach results in significant inaccuracies due to Euler integration, making it unsuitable for achieving the required precision in many practical scenarios. However, direct collocation shows promise in generating good trajectories, but we observe that without proper stabilization, the rocket is unable to accurately follow the desired trajectory.

In our exploration of DC, we also investigate its combination with Model Predictive Control (MPC). While this approach proves to be feasible, we encounter the challenge of tuning requirements to achieve optimal performance. Ultimately, we find that the most effective and consistent method involves integrating DC with LQR stabilization. This combined approach outperforms the other methods we study, providing reliable and desirable outcomes for rocket trajectory optimization.

II. Prior Work

Some authors have explored using control techniques to simulate self-landing rockets in the past. Wang et al. [4] used an MPC-based approach to solve the rocket landing problem. They attempted to minimize fuel usage and used an online convex optimization formulation. Ferrante [5] explored the rocket landing problem from an MPC perspective as well. Ferrante also developed reinforcement learning-based approaches using both Q-learning and policy gradients that do not require the explicit definition of the system dynamics. Kamath et. al [3] also approached the problem from a real-time optimization perspective. Focusing on optimization feasibility, they proposed a formulation that involves the iterative solving of conic optimization problems.

While our work explores the feasibility of MPC, we also explore the viability of a number of other approaches including LQR and LQR-based controllers that stabilize around a nominal trajectory.

III. Problem

We first explore our problem in detail including our assumptions and our state and control vector definitions.

A. Assumptions

We seek to model the Falcon 9 booster after it detaches from its payload and returns to Earth. Our control comes

from the booster's engines, which generates thrust. There are several key assumptions we make in our model to simplify the dynamics of the system.

First, we simplify the representation of the Falcon 9 by modeling it as a solid cylinder of constant mass m, whereas in real life, the rocket's mass decreases as it consumes fuel. This would require us to model mass as a function of time, which we did not incorporate into our model. In addition, while the Falcon 9 has nine engines, we adopt a simpler model of thrust with one engine, where both the magnitude and angle of the thrust produced by the engine can be varied instantaneously. We also assume the gravitational acceleration g remains constant at 9.81 m/s^2 throughout. This is unreasonable because the Falcon 9 booster roughly separates from the payload at an altitude of over 100 km above the Earth's surface, but it simplifies our analysis.

B. State Definitions

Our model allows the rocket to perform a successful landing starting from various altitudes, horizontal distances from the landing pad, and angles to the horizontal, as well as various translational and angular velocities. Thus, we define the following state vector for the rocket:

$$X = \begin{pmatrix} q & \dot{q} \end{pmatrix}^T = \begin{pmatrix} x & y & \theta & \dot{x} & \dot{y} & \dot{\theta} \end{pmatrix}^T$$

Here, x is the horizontal distance from the landing pad, y is the vertical distance from the landing pad, and θ is the angle of the vector pointing in the direction of the rocket head with respect to the vertical (measured in a clockwise fashion). The rocket spins about its center of mass. We can also define the position as $q = \begin{pmatrix} x & y & \theta \end{pmatrix}^T$, where $X = \begin{pmatrix} q & \dot{q} \end{pmatrix}^T$.

Our control vector is defined as

$$U = \begin{pmatrix} u & \phi \end{pmatrix}^T$$

and consists of the magnitude u of the rocket's thrust and the angle ϕ along which the thrust is directed, where the angle is measured with respect to the vector pointing in the direction of the rocket's head (measured in a clockwise fashion). This means that that relative to the vertical (measured clockwise), the thrust is directed at an angle of $\theta + \phi$. Figure 1 illustrates the components of the state and control vectors in red and black, respectively.

We would like the rocket to come to a rest on top of the landing pad. We call that the desired state of the rocket and define its position to be the origin (the velocity is zero because the rocket is at rest):

$$X_f = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}^T.$$

We denote the initial state as

$$X_0 = \begin{pmatrix} x_0 & y_0 & \theta_0 & \dot{x}_0 & \dot{y}_0 & \dot{\theta}_0 \end{pmatrix}^T$$

where y_0 is some positive value and other values can theoretically take on arbitrary values. We expect θ_0 and $\dot{\theta}_0$ to be small because rockets tend to face upwards toward the sky and tend to either rotate slowly or not rotate at all.

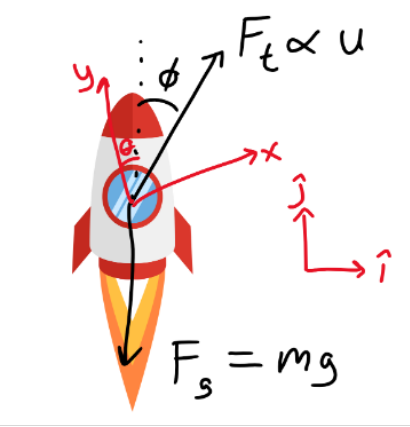


Fig. 1: A depiction of our system, state vector, and control vector

C. System Constraints

We impose several constraints on our system. We assume the thrust's angle is limited:

$$-\phi_{max} \leq \phi \leq \phi_{max}.$$

We also assume a thrust limit u_{max}, so

$$0 \leq u \leq u_{max}.$$

In our experiments, we take $u_{max} = 2mg$, so that the rocket's thrust is double the minimum thrust needed to keep it elevated at a fixed vertical distance from the ground. We also take $\phi_{max} = \pi/9$, as it is unlikely for a rocket's gimbal angle to exceed this value. The rocket also cannot go below the ground:

$$y \geq 0.$$

D. System Dynamics

We assume that our rocket lives in the 2D plane and can be modelled as a solid cylinder with radius r and heigh h, with its mass uniformly distributed across its body. This means that the moment of inertia of the rocket with respect to rotation in the 2D plane is

$$I = \frac{m \cdot (3r^2 + h^2)}{12}.$$

In Figure 1, we see that the rocket experiences a downward gravitational force of $F_g = mg$, where m denotes the mass of the rocket, and g denotes the gravitational acceleration of the Earth's surface. It also experiences a force F_t with magnitude u directed in the direction exactly opposite the thrust: if the thrust pushes downwards, the rocket gets pushes upwards. Using the diagram as a helper, we can model the dynamics of the system as follows. From Newton's second law, we have

$$\sum F_x = u \sin(\theta + \phi) = m\ddot{x} \tag{1}$$

$$\sum F_y = u \cos(\theta + \phi) - mg = m\ddot{y} \tag{2}$$

The thrust also produces a torque:

$$\begin{aligned} \sum \tau &= I\alpha \\ &= -u\sin(\phi)\frac{h}{2} = I\ddot{\theta}. \end{aligned} \tag{3}$$

Once again, we assume our system mass m is constant throughout time (i.e., fuel loss is not modeled) and uniformly distributed across the rocket's length. These equations give us our first-order dynamics:

$$\dot{X} = f(X,U) = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \\ \frac{u\sin(\theta+\phi)}{m} \\ \frac{u\cos(\theta+\phi)}{m} - g \\ -\frac{u\sin(\phi)h}{2I} \end{pmatrix}.$$

IV. Simulation Implementation

Next, we describe how we simulate 2D rocket physics.

A. Pydrake

We use Pydrake, a Python binding to Drake, to model our system. Drake allows us to simulate the physics of the system, including gravity. We construct our system with a MultibodyPlantSceneGraph and DiagramBuilder. Our equations of motion and constraints are added as a MathematicalProgram. Pydrake classes we use include but are not limited to LinearQuadraticRegulator, PiecewisePolynomial, Solve, DirectCollocation. We also use Pydrake's MeshcatVisualizer and Simulator to make a 2D simulation of our model.

B. Rocket Definition

We create a URDF for our system. Figure 2 shows an annotated URDF, and Figure 3 is our Python code showing a high-level overview of the URDF's components.

Our URDF contains multiple links: the world, the landing pad, two virtual links, the rocket link, the rocket's engine link. The `world_link` frames the scene, and is a long rectangle that extends horizontally; it serves as a marker for where ground level is. Within the world, we define a landing `pad_link`, which sits on top of the ground. In addition, we have two virtual links, `virtual_link1` and `virtual_link2`, which allow movement in the x- and y-directions, respectively. The itself is made of a cylindrical `rocket_link`, with an `engine_link` at its end.

Joints include the `pad_joint`, which fixes the landing pad to the world. They also include `x_joint`, which allows the rocket to move in the x-direction, and `y_joint`, which allows the rocket to move in the y-direction. Moreover, `theta_joint` allows the rocket to rotate about its center of mass while `engine_joint` attaches the engine to the bottom of the rocket.

Finally, we define three control transmissions: `ux_transmission` for force in the x-direction; `uy_transmission` for force in the y-direction; and `tau_transmission` for actuation torque about the rocket's center of mass.

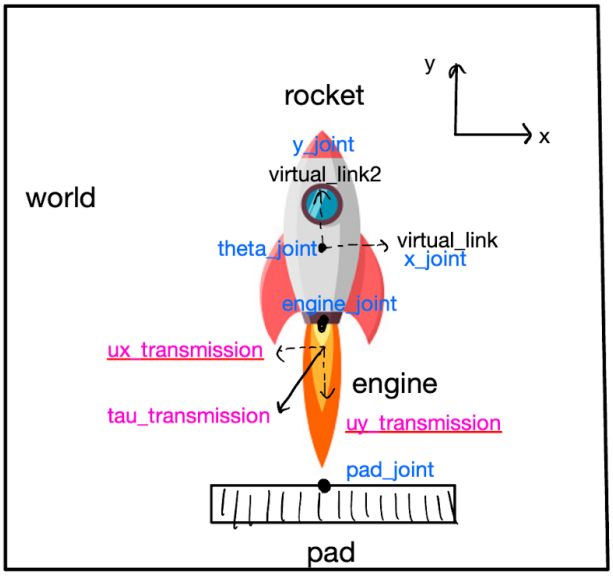


Fig. 2: Annotated URDF of rocket, showing links (in black), joints (in blue), and transmissions (in pink).

Note that the controller itself does not have access to these three transmission levers; rather, the controller outputs a particular u and ϕ, which can be used to calculate the resulting forces and torque on the rocket. In our code, we create an IntermediateSystem LeafSystem, which serves as a converter of the controller's outputs to the corresponding forces and torque applied to the rocket plant itself.

In Figure 4, we show a screenshot of our actual simulation. The rocket is black, and the engine is in red. The landing pad is gray.

Note that that to simplify the simulation, we did not implement collision dynamics between the rocket and the landing pad or the ground. In later sections, we note cases in which controllers generated trajectories that made contact with the landing pad or ground, and thus may have resulted in collision in the real world.

V. Controller Implementation

With the simulation in place, we now construct controllers designed to help the rocket land successfully onto the landing pad.

A. Null Controller

We first implemented a baseline test controller without any control output by setting

$$U = \begin{pmatrix} 0 & 0 \end{pmatrix}^T$$

at all times t. In this scenario, the rocket simply drops downwards due to the force of gravity, as expected.

```
rocket_urdf = f"""
<?xml version="1.0"?>
    <robot name="rocket">
        {world_link}
        {pad_link}
        {virtual_link}
        {virtual_link2}
        {rocket_link}
        {engine_link}

        {pad_joint}
        {x_joint}
        {y_joint}
        {theta_joint}
        {engine_joint}

        {ux_transmission}
        {uy_transmission}
        {tau_transmission}
    </robot>
</xml>
"""
```

Fig. 3: Python snippet of our Rocket URDF, with the names of each link, joint, and transmission, as referred to in the paper.

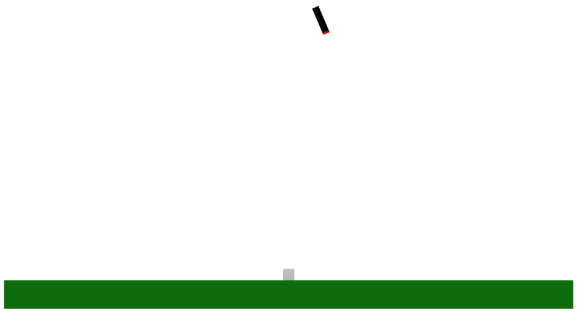


Fig. 4: Screenshot of our Pydrake simulation with a rocket (black), engine (red), ground (green), and landing pad (gray).

B. LQR Controller

We create a LQR controller using Drake's LinearQuadraticRegulator class and choosing state cost matrix

$$Q = \text{diag}\begin{pmatrix}1 & 1 & 1 & 1 & 1 & 1\end{pmatrix}$$

and control cost matrix

$$R = \text{diag}\begin{pmatrix}1 & 1\end{pmatrix}$$

While we do try several different values of Q and R, we find that adjusting the relative costs does not affect the rocket trajectories significantly.

Moreover, LQR provides no natural way to account for control limits. As a result, we clip the control outputs if the solutions outputted by LQR like outside our bounds for the magnitude and angle of the thrust. The rocket dynamics are consequently updated using those clipped values. In practice, we implement this by creating a VectorSystem called Clipper that performs the clipping, and connect its input to the output of Pydrake's LQR controller.

C. Trajectory Optimization Controller with DT

We implement a direct transcription trajectory optimization problem manually by making use of Pydrake's MathematicalProgram class.

Our trajectory optimization problem's decision variables are six-dimensional $X[t]$ and two-dimensional $U[t]$ at discrete times $t = 0, h, 2h, \cdots, (N-1) \cdot h, N \cdot h = T$, where h is the time step and N is the number of time steps we optimize over. (For $U[t]$, we exclude the last time step as usual.) In our experiments, we set $h = 0.2$ seconds and $N = 100$. This means that the rocket is optimized over a total time interval of $T = 20$ seconds, which implies that we expect the rocket to be in flight for at most 20 seconds. While setting smaller h can help better approximate the continuous dynamics, we found that the optimizer would take a very long time when h was set to values significantly smaller than 0.2.

Before running the optimization, we generate an initial, non-optimized guess of the rocket's state at each time step by doing a linear interpolation across time of each $X[t]$ assuming that $X[0]$ is the initial state X_0 and $X[T]$ is the final desired state X_f. We then add a bit of Gaussian noise to this interpolation. We hope that this educated guess can help guide the optimizer to the solution more quickly.

Next, we define the following constraints for the MathematicalProgram:

1) **Initial state:** The rocket's starting state $X[0]$ should be set to the initial state X_0.
2) **Dynamics constraints:** For each time step t, constrain $q[t+1]$ such that
   a) $x[t+1] = x[t] + h \cdot \dot{x}[t]$
   b) $y[t+1] = y[t] + h \cdot \dot{y}[t]$
   c) $\theta[t+1] = x[t] + h \cdot \dot{\theta}[t]$

   Similarly, constrain $\dot{q}[t+1]$ such that
   a) $\dot{x}[t+1] = \dot{x}[t] + h \cdot \ddot{x}[t]$
   b) $\dot{y}[t+1] = \dot{y}[t] + h \cdot \ddot{y}[t]$
   c) $\dot{\theta}[t+1] = \dot{\theta}[t] + h \cdot \ddot{\theta}[t]$

   where $\ddot{x}[t]$, $\ddot{y}[t]$, $\ddot{\theta}[t]$ are derived from equations 1, 2, 3, respectively. Note that the constraints here are non-linear, as the dynamics themselves are non-linear.
3) **State constraints:** For each time step t, the rocket should:
   a) Be above ground, which implies $y \geq 0$.

b) Have a non-negative thrust that is less than or equal to the maximum thrust, which means

$$0 \leq u \leq u_{max}.$$

c) Have its thrust vector be within the engine angle limit, which means

$$-\phi_{max} \leq \phi \leq \phi_{max}.$$

d) Be within $\pi/2$ of the vertical (i.e., the rocket should not have its head lower than its engine). This is an artificial constraint that is not a physical necessity. However, we found that in many cases, it prevented the rocket from spinning in loops and thus taking on fast-moving trajectories that were difficult for the discrete dynamics approximations to model accurately.

Finally, for each time step and component of the state $X[t]$, we add a quadratic cost $X[t]^T W X[t]$, where

$$W = \text{diag}\begin{pmatrix}10 & 1 & 10 & 1 & 1 & 1\end{pmatrix}.$$

We put a higher cost on $x[t]$ to guide the rocket over launch pad early on, which we found generally helped to increase the probability of landing success. We also put a higher cost on $\theta[t]$ to discourage tilting of the rocket, which we found was negatively correlated with a successful landing.

Finally, after optimization, we create a piecewise linear function from our discrete control output solution. Our trajectory optimization controller works as follows: it simply queries for the current time and outputs the control output of the optimized trajectory corresponding to that time. Note that there is no feedback involved. As we will see in the results section, the lack of feedback leads to situations in which the rocket misses the landing pad.

D. Trajectory Optimization Controller with DC

We also experiment with using direct collocation to solve for the optimal trajectory rather than direct transcription because of the discretization errors introduced by the simple Euler approximation used in our manually-implemented DT.

DC solves the non-convex trajectory optimization problem by defining a series of spline breakpoints known as sample points. In DC, given a set of sample points denoted by t, and assuming a piecewise-linear function $u(t)$ over those sample points, we can fully define $X(t)$ as a cubic spline. DC then enforces derivative constraints at the collocation points, which fall between the sample points: $\dot{x}(t_c) = f(x(t_c), u(t_c))$ for each collocation time t_c.

We use Pydrake's DirectCollocation class to implement the core of DC. We enforce $h = 1$ in our optimization problem, as well as the equal time step constraint. Our time step for DC (1) is higher than the time step used in DT (0.2) because we found that DC was generally slower than DT, so setting a time step of around 0.2 was too computationally expensive. As in DT, we also enforce the initial state constraint, the limits on the control outputs, the non-negativity of y, and the $-\pi/2 \leq \theta \leq \pi/2$ constraint to discourage spinning. Moreover, just as in DT, we use linear interpolation to set the initial guess for X at the sample points.

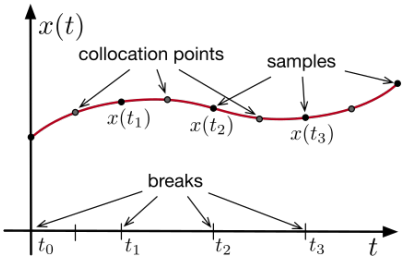


Fig. 5: An illustration of direct collocation with cubic spline $x(t)$.

We also add an additional quadratic cost that penalizes thrust (fuel) usage by adding a $u[t]^2$ term to the cost at each sample point t. We did not add this term to the cost function in DT because it appeared to degrade performance for the DT case.

As in DT, our trajectory optimization controller works by querying for the current time and outputting the control output corresponding to that time in the optimized trajectory.

E. MPC Controller

As we will discuss in the results, vanilla feedback-free controllers based on DC and DT tended to miss the target state in simulation due to approximation errors. To try to address this issue, we explore using model-predictive control to repeatedly re-optimize the trajectory as the rocket begins its descent to the landing pad.

Since we obtained better results with DC, we iteratively re-optimize our DC-computed trajectory. In practice, our controller attempts to re-optimize every 0.2 seconds, which it does so in a separate thread so that the controller can still be queried by the simulator with virtually no blocking. Once the optimization finishes, the trajectories, which are member variables of the controller, are updated. To prevent concurrency issues, we use locks when reading and writing to these variables.

Because MPC requires the optimization to be fast (otherwise the optimized trajectory will be quite stale when it is finally generated), we only optimize over three sample times, so $N = 3$. Furthermore, h decreases over time depending on how much time has passed. At the beginning, h is set to $\frac{20}{3}$. After t seconds have passed, h is set to $\frac{20-t}{3}$, although we lower bound h to $\frac{0.5}{3}$ to allow the MPC to continue to generate trajectories indefinitely (rather than stop optimizing after 20 seconds has passed). If the optimization takes longer than than the 0.2 second re-optimization time period, the controller will not kick off another optimization but rather wait for the currently running one to finish. In practice, the DC runs took between 0.1 and 0.2 seconds to complete.

F. LQR Stabilization Controller

We also attempt to achieve trajectory stabilization using LQR by stabilizing over the nominal trajectory generated by trajectory optimization. We try using an LQR stabilizing controller for both DC and DT, which can be achieved via FiniteHorizonLinearQuadraticRegulator in Pydrake.

Similar to the vanilla LQR controller, we use costs

$$Q = \text{diag}\begin{pmatrix}1 & 1 & 1 & 1 & 1 & 1\end{pmatrix},$$

and

$$R = \text{diag}\begin{pmatrix}1 & 1\end{pmatrix},$$

where we optimize from the start to the end time of the trajectory outputted by either DT or DC. We once again saw little benefit of changing the cost matrices.

As before, because of the difficulty of naturally including constraints in the LQR formulation, we instead solve the unconstrained problem and clip the control output U if it falls outside the range allowed for the thrust angle and magnitude.

VI. Results

We tested these rocket controllers in a variety of different scenarios, but for this paper, we report graphs on three, which we call *easy*, *medium*, and *hard*. This allows us to more easily compare between the different controller methods.

In the *easy* scenario, the rocket starts in an upright stationary position directly above the landing pad. In this paper, we set $y_0 = 40$, where all other state variables are initially 0. To perform self-landing successfully in this case, the controller would not need to worry too much about horizontal movement and changing the angle of the thrust. In the *medium* scenario, the rocket starting in an upright stationary position, as before. However, this time, it has both a horizontal and vertical positional offset. In particular, it starts to the left and above the landing pad rather than directly above it. In this paper, we set $y_0 = 40$ and $x_0 = -5$ (with other state variable initially 0). In the *hard* scenario, all six values in the initial state vector are non-zero. In this paper, this initial state is

$$X_0 = \begin{pmatrix}-35 & 30 & \pi/8 & -1 & 5 & -0.2\end{pmatrix}^T.$$

In Figure 6, we provide a summary of each controller's performance in each of the scenarios. The asterisks indicate cases in which the landing succeeded with overshoot, which we will address later. Moreover, in Figure 7, we show plots of each of the generated trajectories. Specifically, we plot the initial state in red, the goal state in green, and the rocket's true trajectory, as experienced in the simulation, in black. The blue arrows show the orientation of the rocket's head at sample times during the trajectory; only the arrows' orientations are meaningful. The orange arrows show the magnitude and direction of the thrust force experienced by the rocket at the same sample times; the arrows' orientations and magnitudes are both meaningful. When the orange and blue arrows are pointing in the same direction, the rocket is directing force directly downwards (i.e., the thrust angle is 0).

Controller	**Scenario**		
	Easy	***Medium***	***Hard***
Null controller	N	N	N
LQR	Y*	Y*	N
DT	N	N	N
DC	Y	N	N
LQR-Stabilized DT	Y	N	N
LQR-Stabilized DC	Y	Y	Y
MPC (with DC)	Y	Y	Y

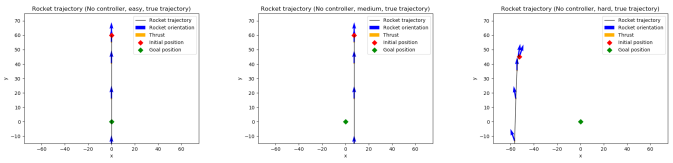
Fig. 6: Table of each controller's success or failure in landing the rocket in easy, medium, and hard scenarios. Y means the rocket landed successfully; N means it failed to landed successfully. The asterisks indicate cases in which the landing succeeded with overshoot.

Overall, only MPC with DC and LQR-stabilized DC led to successful rocket landings across all three scenarios. We now go through each controller's performance in turn.

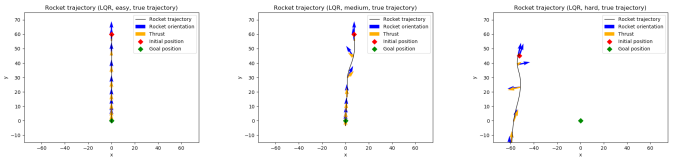
As expected the null controller led to the rocket taking on the same dynamics as if the controller did not exist at all. In particular, the rocket would eventually through to the ground past the landing pad (recall that we did not model collision dynamics here).

Using just LQR, the easy and medium scenarios succeed, as the rocket ends up stationary on the landing pad after a sufficient amount of time has passed. However, we mark the success of LQR with an asterisk because it tends to overshoot the landing pad (go below it), before undershooting it (go above it), before overshooting it again. This oscillation continues with decreasing amplitude until the rocket essentially comes to a rest at the landing pad. This overshooting means that the rocket may not successfully land in practice, although that depends on the extent of the overshooting as well as the ability of the landing pad and rocket to absorb a temporary shock. Perhaps a slight overshooting may potentially be fine if the landing pad has sufficient elasticity. The hard scenario is extremely off-target as the change in the rocket's x-position is very small, which suggests that the linear approximation is very poor at this distance. Thus, LQR appears to be a decent controller when the initial state is fairly close the final state. However, it does suffer from the drawback of not being able to implement constraints like $y \geq 0$ natively, which can cause overshooting during rocket landing.

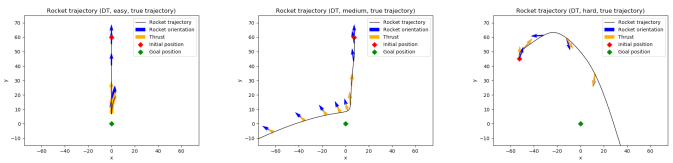
The DT case failed in every scenario. In the easy scenario, the rocket verges upwards and to the right just before it reaches the launchpad, and never reaches launch-pad level again. In the medium scenario, the rocket similarly over-corrects itself when it is very close to the launchpad; it ends up moving quickly to the left, significantly missing the launchpad. Finally, in the hard scenario, the rocket initially thrusts and orients itself in the correct direction but overshoots as it moves rightward, ultimately missing the goal state by a large margin. Our results show that the discretization errors imposed by our Euler-integration-based DT are causing significant problems



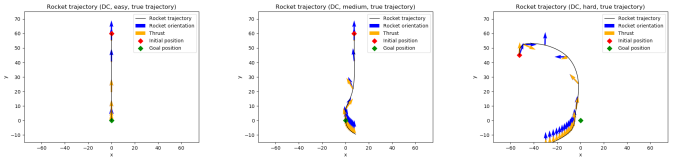
(a) Null Controller



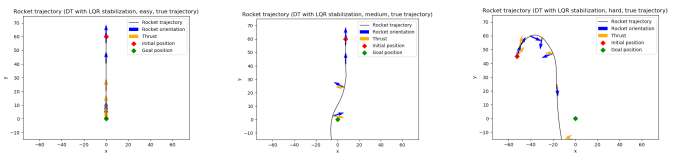
(b) LQR



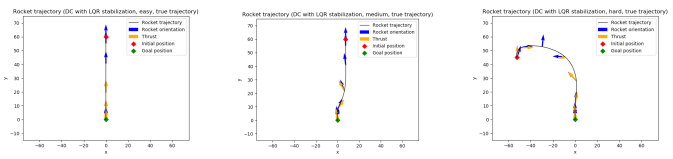
(c) DT



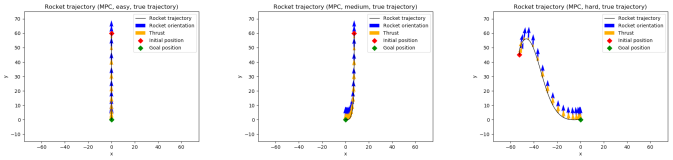
(d) DC



(e) LQR-Stabilized DT



(f) LQR-Stabilized DC



(g) MPC (with DC)

Fig. 7: Rocket trajectories generated by each controllers on easy, medium, and hard scenarios. Initial and goal states, thrust vectors, and rocket orientation vectors are also depicted.

in simulation; even in the easiest cases, the rocket cannot get to the launch pad. In theory, these issues could be corrected by decreasing the time step h from 0.2 to a much smaller number like 0.01. However, this is computational infeasible, which implies that vanilla controllers using a single DT trajectory are not effective here.

DC only succeeds in the easy scenario, but this marks an improvement over DT, which fails in all three. In the medium scenario, the rocket actually overshoots, passing through the target state and moving erroneously to the right. In the hard scenario, the rocket once again nearly reaches the target state before veering to the left. Despite mediocre results, DC's improvement over DT suggests that the approximating the dynamics by enforcing derivative constraints on the cubic spline at the collocation points is a more effective approximation of the rocket's physics compared to Euler integration. Indeed, even though we set $h = 1$ for DC, the results are notably better than our results when setting $h = 0.2$ for DT.

We may hope that adding LQR stabilization to DT can lead to successful landing in a variety of scenarios, but we find that this is not the case. Only the easy landing succeeds (which at least is an improvement to vanilla DT). The medium and hard landings once again fail with overshooting and undershooting in the x-direction, respectively. However, we note that the amount by which the landing pad is missed is significantly smaller than the case without stabilization. Clearly, LQR stabilization is helping here.

Despite somewhat lackluster results with DT, LQR stabilization with DC succeeds in each of the three scenarios. This suggests that a good linear stabilizer itself is not enough if the trajectory optimization itself is a bad approximation of real physics. As shown in our video and on the plots, there is no overshooting, despite the fact that we are using LQR, which cannot enforce the $y \geq 0$ constraint natively. While only three scenarios are shown here, we also test this controller for a much wider variety of starting states and find very good stability properties and landing success rates. It appears to be the most reliable controller of the ones we implemented.

Finally, MPC with DC also succeeds in all three scenarios. However, its optimal trajectory differs significantly from the trajectory found by DC. Compared to LQR stabilization, MPC appears to generate a slower rocket landing that makes adjustments in the x-direction at the last moment. Additionally, MPC appears to generate an upright rocket at all times compared to LQR stabilization, which tolerates deviations from $\theta = 0$. In our experience, we saw that MPC was less reliable than LQR stabilization and more frequently failed to land the rocket in cases where LQR stabilization succeeded (although the converse was rarely true). Moreover, MPC was very difficult to tune, since we needed to strike a balance between good optimization quality and a small enough optimization time to allow sufficiently frequent updates of the desired trajectory.

Overall, our findings highlight the importance of considering the specific requirements and challenges of self-landing rocket control when selecting the appropriate method. The integration of DC with LQR stabilization demonstrates a

promising avenue for achieving optimal performance and precision in trajectory calculations.

Our project video can be found here, which shows some clips of the rocket landing in the Pydrake simulation.

VII. Discussion and Future Work

There are a few aspects of our system that we would like to make more realistic. Our current simulation only deals with the 2D landing case, which is obviously not realistic in the real world. In the future, we would like to move to a three-dimensional representation of the problem. This would pose the challenge of having larger state and control vectors, and the higher dimensionality may affect the computational viability of our trajectory optimization methods.

Another item we would like to tackle would be to include more complex dynamics, the first of which would be accounting for changing values of m and g, both of which we assumed to be constant. The rocket's mass would change with the amount of fuel burned, and the gravitational acceleration would also change with respect to elevation.

Furthermore, our current work only deals with a stationary landing pad with no external forces other than gravity and thrust. However, the Falcon 9 has demonstrated the capability to land on a moving ship as well and to handle real weather conditions. In the future, we are interested in attempting to land the rocket on a simulated drone ship in the water, which would force the rocket to stabilize in real time to a potentially moving trajectory. Another challenge we could tackle would be landing with wind, which would affect the optimal trajectory computed and could make stabilization more difficult.

Finally, we constrain $|\phi|$ to be no more than $\frac{\pi}{9} = 20^\circ$ in this work. In the future, we would like to see if $|\phi|$ can be constrained to 5° to 10°, which is a more realistic gimbal angle. Also, we would like to incorporate the constraint that ϕ cannot be changed instantaneously but rather must be accelerated with a force, as the engine itself has mass.

VIII. Team Member Contributions

David worked on writing the direct collocation and LQR stabilization code, as well as the code for building the simulation and generating trajectory plots. Mindy worked on the MPC code and writing of the report. Michael worked on the simulation setup code and direct transcription code as well as the development of the video.

References

[1] Wikipedia. "Falcon 9."

[2] The Verge. "SpaceX lands rocket at sea, makes history."

[3] Abhinav G. Kamath, Purnanand Elango, Yue Yu, Skye Mceowen, John M. Carson III, Behcet Acikmese "Real-Time Sequential Conic Optimization for Multi-Phase Rocket Landing Guidance." https://arxiv.org/pdf/2212.00375.pdf

[4] Jinbo Wang, Naigang Cui, and Changzhu Wei. "Optimal Rocket Landing Guidance Using Convex Optimization and Model Predictive Control."

[5] Reuben Ferrante. "A Robust Control Approach for Rocket Landing."