Vertically Landing Rockets with Trajectory Tracking Linear MPC

Jason Pul

December 15, 2022

1 Introduction

This paper explores using linearized rocket state equations and linear MPC optimization to perform 2d state trajectory tracking and vertical landing. It is assumed that an optimal state trajectory $\underline{x}_{optimal}$ is provided for tracking. And that this track terminates at a landing site located at $(0, 0)$ in the upright position with zero linear and angular velocity. A 2d simplified SpaceX Dragon capsule was chosen as the test vehicle. A freebody diagram of Dragon is shown below in figure 1. The simplified capsule has two thrusters aligned with its center-of-gravity and offset by $L_{thruster}$. Both thrusters are limited to $F_{1min} = F_{2min} = 0$ and $F_{1max} = F_{2max} = 10N$. Throughout this paper, underlined variables are used to indicate vectors. All codes for this paper is maintained on a open GitHub repository. Experimental result videos can be viewed there.

github.com/Cylon-Garage/rocket-lander.

2 Related Work

Rocket dynamics are nonlinear and its state equations are non-convex. Mathworks[Mat] solves a similar tracking sample problem, but they use multistage nonlinear MPC for optimization. More state-of-the-art techniques use Successive Convexification (SCvx) to solve such problems [SA18]. Neither method was covered in EE688 and both are beyond my current understanding. Ferrante[Fer17] performs a finite difference linearization using simulation to linearize the rocket state equations. Doing this allowed Ferrante to directly apply linear MPC optimization. However his MPC implementation was unable to maintain track. It's unclear why his implementation fails. This paper extends Ferrante's finite difference linearization technique to successfully track a given state trajectory and vertically land a rocket.

3 Dynamics Equations

The freebody diagram of the dragon capsule is shown in figure 1. The resulting dynamics equations are shown below.

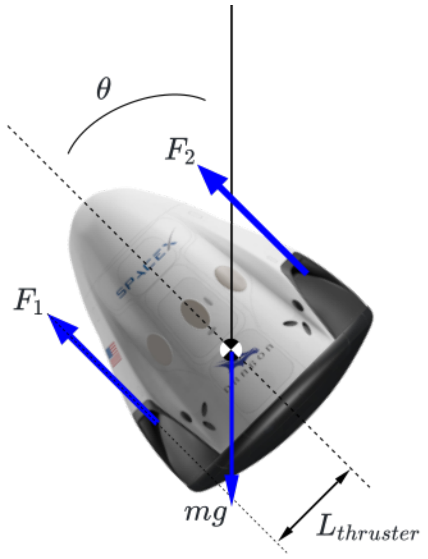

Figure 1: Simplified 2D Dragon Capsule Freebody Diagram

$$\sum F_x = m\ddot{x} = -(F_1 + F_2)\cos\theta \tag{1}$$
$$\sum F_y = m\ddot{y} = -mg + (F_1 + F_2)\sin\theta \tag{2}$$
$$\sum M_{cg} = I\ddot{\theta} = L_{thrust}(F_2 - F_1) \tag{3}$$

4 Nonlinear State Equations

Dragon's state and input vectors are defined below. Its state equations are found by combining those vectors with equations 1-3. The sin and cos functions appearing in equation 6 make the system both **nonlinear** and **non-convex**.

$$\underline{x} = \begin{bmatrix} x & y & \theta & \dot{x} & \dot{y} & \dot{\theta} \end{bmatrix}^T \tag{4}$$
$$\underline{u} = \begin{bmatrix} F_1 & F_2 \end{bmatrix}^T \tag{5}$$
$$f(\underline{x}, \underline{u}) = \dot{\underline{x}} = \begin{bmatrix} x_4 \\ x_5 \\ x_6 \\ \frac{-(u_1+u_2)}{m}\sin x_3 \\ -g + \frac{-(u_1+u_2)}{m}\cos x_3 \\ \frac{L_{thruster}}{I}(u_2 - u_1) \end{bmatrix} \tag{6}$$

5 Problems With Implementing Linear MPC

The state equations are **nonlinear** due to the sin and cos functions. Those functions are also **non-convex**. Therefore, a linear MPC cannot directly be implemented on this system. The system needs to be linearized first.

6 Linearization

6.1 Linearizing Around a Fixed Point (Jacobian Linearization)

$$A = \left[\frac{\partial f_i}{\partial x_j}\right]|_{\underline{x}_{eq},\underline{u}_{eq}} \tag{7}$$

$$B = \left[\frac{\partial f_i}{\partial u_j}\right]|_{\underline{x}_{eq},\underline{u}_{eq}} \tag{8}$$

$$\text{where } f(\underline{x}_{eq},\underline{x}_{eq}) = \dot{\underline{x}}_{eq} = \underline{0} \tag{9}$$

The state matrices A and B are obtained by evaluating the Jacobian at $\underline{x}_{eq}$, $\underline{u}_{eq}$. The fixed point $\underline{x}_{eq}$, $\underline{u}_{eq}$ is obtained by setting equation 6 to $\underline{0}$ and solving for $\underline{x}_{eq}$, $\underline{u}_{eq}$. One equation resulting from doing this is $\frac{-(u_1+u_2)}{m}\sin x_3 = 0$. Which means that $u_1 = -u_2$ and/or $\theta = n\pi$ where $n = 1, 2 \ldots$. The min and max limits on the thrusters mean that u_1 can only equal $-u_2$ when $u_1 = u_2 = 0$. This limits the validity of our state matrices to two different situations. One where the dragon capsule is close to upright or upside down. The other is when both thrusters are off. These two states are too limiting and it will not be possible to track a given state trajectory that will have a wide range of capsule angles that require a wide range of thrust inputs. A different linearization technique must be employed

6.2 Finite Difference Linearization

Ferrante linearized his state equations using finite difference. This linearization is done at the current state, $\underline{x}$, so it is not limited to a fixed point like the technique above. The idea is similar to fixed point linearization as it also uses the Jacobian. However, here the Jacobian is obtained using finite difference. Assume you are at some state $\underline{x}(t)$ with input $\underline{u}(t)$. The following state $\underline{x}(t+\Delta t)$ after Δt seconds can be found by integrating the state equations (6). Lets add and subtract a small perturbation to $x_1(t)$ such that

$$\underline{x}_{+}(t) = \begin{bmatrix} x+\epsilon \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} \text{ and } \underline{x}_{-}(t) = \begin{bmatrix} x-\epsilon \\ y \\ \theta \\ \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix}$$

We can again solve the system of ODEs to obtain $\underline{x}_{+}(t+\Delta t)$ and $\underline{x}_{-}(t+\Delta t)$. These two states can be used to approximate $\frac{\partial f(t)}{\partial x_1}$:

$$\frac{\partial f(t)}{\partial x_1} \approx \frac{\Delta f(t)}{\Delta x_1} = \frac{\underline{x}_{+}(t+\Delta t) - \underline{x}_{-}(t+\Delta t)}{2\epsilon}$$

This partial derivative approximation can be used to estimate the state matrices A and B at time t with

$$A(t) \approx \begin{bmatrix} \frac{\Delta f(t)}{\Delta x_1} & \frac{\Delta f(t)}{\Delta x_2} & \frac{\Delta f(t)}{\Delta x_3} & \frac{\Delta f(t)}{\Delta x_4} & \frac{\Delta f(t)}{\Delta x_5} & \frac{\Delta f(t)}{\Delta x_6} \end{bmatrix} \tag{10}$$

$$B(t) \approx \begin{bmatrix} \frac{\Delta f(t)}{\Delta u_1} & \frac{\Delta f(t)}{\Delta u_2} \end{bmatrix} \tag{11}$$

Ferrante performed this approximation by indirectly calculating each $\underline{x}_{+}(t+\Delta t)$ and $\underline{x}_{-}(t+\Delta t)$ through a simulation. He used Box2D as his physics engine and calculated each finite difference from the resulting simulation state after using ϵ perturbations though a timestep Δt. I thought this could be simplified and replaced the simulator with SciPy's solve_ivp. This is similar to Matlab's ODE45 solver.

6.2.1 Gravity

This finite difference linearization technique was new to me. It made sense that it would work but I had to verify it somehow. Lets call the ground truth solution $\hat{\underline{x}}_{t+1}$. $A(t)$ and $B(t)$ are valid near $\underline{x}(t)$.

So $\underline{x}_{t+1} = A\underline{x}_t + B\underline{u}_t \approx \hat{\underline{x}}_{t+1}$. If you start with an initial state, $\underline{x}_0$ and evolve the system through a short time horizon ($n_{steps} = 10$ with $\Delta t = 0.2$) the two trajectories should be **close**. My initial tests did not show this as seen in figure 2.

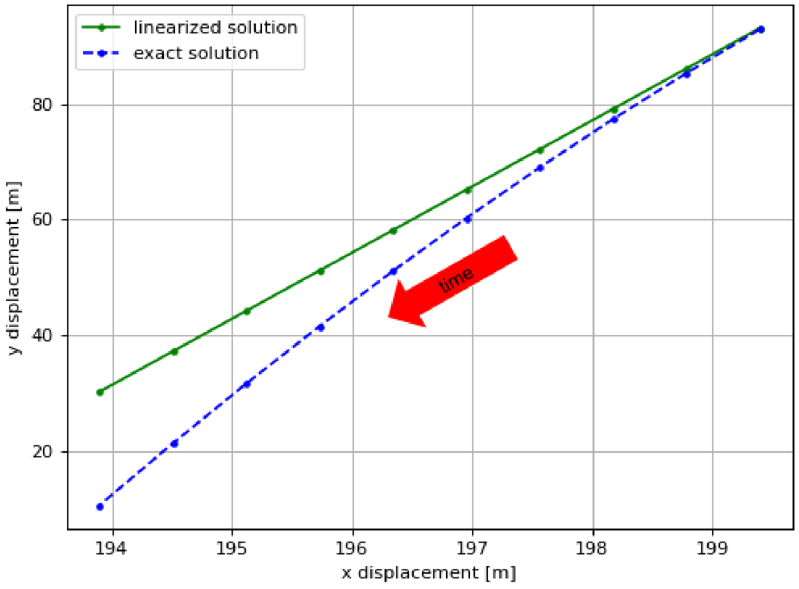

Figure 2: Initial Finite Difference Linearization Verification

Inspecting equations 6, 10 and 11 reveal the issue. Gravity is never perturbed during the calculation of A & B. This means that gravity is essentially subtracted out when calculating the finite differences. To resolve this, gravity has to be involved in the Jacobian. This is done by treating gravity as an input:

$$\underline{u} = \begin{bmatrix} F_1 & F_2 & g \end{bmatrix}^T$$

This modification results in $A\underline{x}_t + B\underline{u}_t \approx \hat{\underline{x}}_{t+1}$, validating the finite difference linearization technique (figure 3).

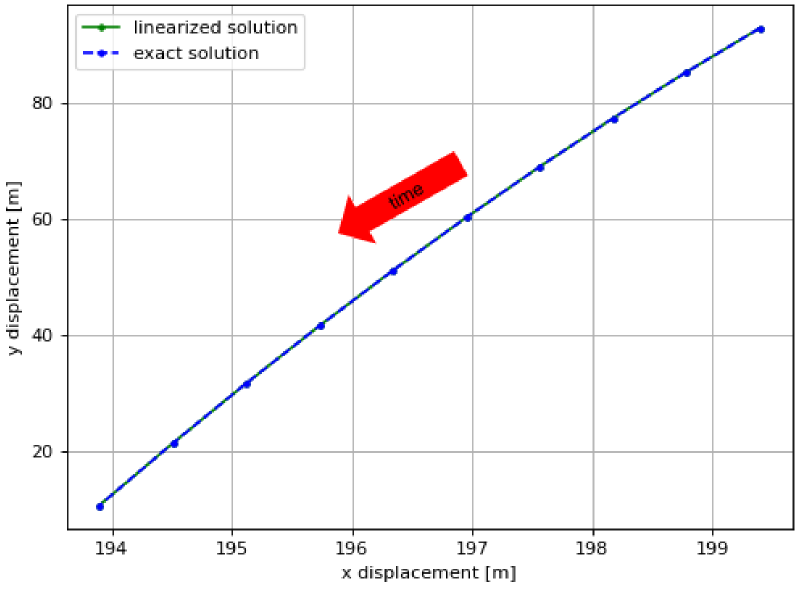

Figure 3: Treating Gravity as an Input

7 Obtaining an Optimal Trajectory

Nonlinear Model Predictive Control (NMPC) was utilized to obtain an optimal trajectory. A NMPC optimizes the nonlinear dynamics equations directly without the need for linearization. The optimizations were performed by minimizing total thrust (fuel) used from an initial state $\underline{x}_0$ to final state $\underline{x}_f = \underline{0}$ The do-mpc library was used to implement the optimization. NMPCs were not covered in EE688 and I do not have a good understanding of them. They are outside the scope of this paper and were solely used to obtain an optimal trajectory.

8 Model Predictive Controller

I implemented the tracking MPC using quadratic cost:

$$\begin{aligned} J = {} & \underline{x}_{error}^T(N) P \underline{x}_{error}(N) + \\ & \sum_{n=0}^{N-1} [\underline{x}_{error}^T(n) Q \underline{x}_{error}(n) + \\ & \quad \underline{u}_{delta}^T(n) R \underline{u}_{delta}(n)] \\ \text{where } & \underline{x}_{error} = \underline{x} - \underline{x}_{ref} \\ \text{and } & \underline{u}_{delta} = \underline{u}_n - \underline{u}_{n-1} \end{aligned} \tag{12}$$

This results in a simple MPC optimization. The optimization constraints are the linear state equation, requirement for the rocket to remain above ground at all times, thrust restrictions, and gravitational acceleration.

$$\begin{aligned} & \min \quad J \\ \text{s.t.} \quad & \underline{x}_{t+1} = A\underline{x}_t + B\underline{u}_t \\ & \underline{x}_t[2] \geq 0 \\ & \underline{u}_t \geq 0 \\ & \underline{u}_t \leq 10 \\ & \underline{u}_t[3] = g \end{aligned} \tag{13}$$

I found through empirical testing that penalizing θ and $\dot{\theta}$ were much more important than penalizing position and linear velocity. Maintaining track was also more important that reducing changes in input. This resulted in the following penalty matrices:

$$P = Q = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 50 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 10 \end{bmatrix} \tag{14}$$

$$R = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix} \tag{15}$$

The linearized state matrices are only valid near $\underline{x}_t$, so I kept the MPC horizon short ($n_{horizon}$). Towards the end of the simulation the horizon will extend beyond data available within the provided optimal track. This is handled by forcing all states beyond the optimal trajectory to equal $\underline{0}$. This data extension was enough to implicitly enforce terminal state requirements $\underline{x}_{terminal}$. Terminal state requirements were never explicitly enforced.

$$
\begin{aligned}
n_{steps} &= 150 \\
n_{horizon} &= 10 \\
\Delta t &= 0.2 \\
\epsilon &= 0.01 \\
G &= 9.80665
\end{aligned}
$$

8.1 Main Program

```
x = x_0
u = [0, 0, G]
x = x_0
u = u_0
inputs = [u]
states = [x]
for t in range(n_steps - 1):
    x_ref = get_reference_state(x[:2])
    A, B = rl.get_linearized_state(
        x, u, TIMESTEP, epsilon, rl.dragon_state_function)

    u_optimal = mpc(x, u, A, B, x_ref)
    u = u_optimal[:, 0]
    x = solve_ivp(rl.dragon_state_function, t_span=[
        0, TIMESTEP], y0=x, args=(u,)).y[:, -1]
```

8.1.1 MPC Program

```
def mpc(
    x_0: np.ndarray,
    u_last: np.ndarray,
    A: np.ndarray,
    B: np.ndarray,
    x_ref: np.ndarray) -> np.ndarray:

    x = cp.Variable((n_states, TRACKER_N_HORIZON + 1))
    u = cp.Variable((n_inputs, TRACKER_N_HORIZON))

    cost = 0
    constraints = [x[:, 0] == x_0.squeeze()]
    for t in range(TRACKER_N_HORIZON):
        du = u_last - u[:, t]

        cost += cp.quad_form(x[:, t] - x_ref[t, :], Q) + \
            cp.quad_form(du, R)
        constraints += [
            x[:, t + 1] == A @ x[:, t] + B @ u[:, t],
            x[1] >= DRAGON_CG_HEIGHT,

            u[[0, 1], t] >= 0,
            u[[0, 1], t] <= MAX_THRUST,
            u[2, t] == G,
        ]
    constraints += [
        x[1] >= DRAGON_CG_HEIGHT,
    ]
    problem = cp.Problem(cp.Minimize(cost), constraints)
    j_star = problem.solve()
    x_star = x.value
    u_star = u.value
    return u_star
```

get_reference_state, **dragon_state_function**, **get_linearized_state** are shown in the Appendix. All code is available on GitHub.

9 Tracking Results

9.1 Track 1

For the first test, we start in a position to the top-right of the landing site. The dragon capsule is descending at a speed of 10 m/s oriented at -45° ($\underline{x}_0 = \begin{bmatrix} 200 & 100 & -45 & -7.1 & -7.1 & 0 \end{bmatrix}^T$). An NMPC optimizer is used to first obtain an optimal state trajectory. This trajectory is then provided to the linear MPC to perform tracking. The resulting track shown in figure 4 is not a direct overlay but still very close. You can see that the orientation track is also very close. And the results show that the capsule successfully lands vertically at the landing site. It's interesting to note that the optimal and tracker trajectory have very different inputs. The overall shape is the same but the tracker inputs are not smooth. This is likely due to having such a short prediction horizon. Also approximating the system as linear probably effects the input differences. A video of this test is available for viewing on GitHub.

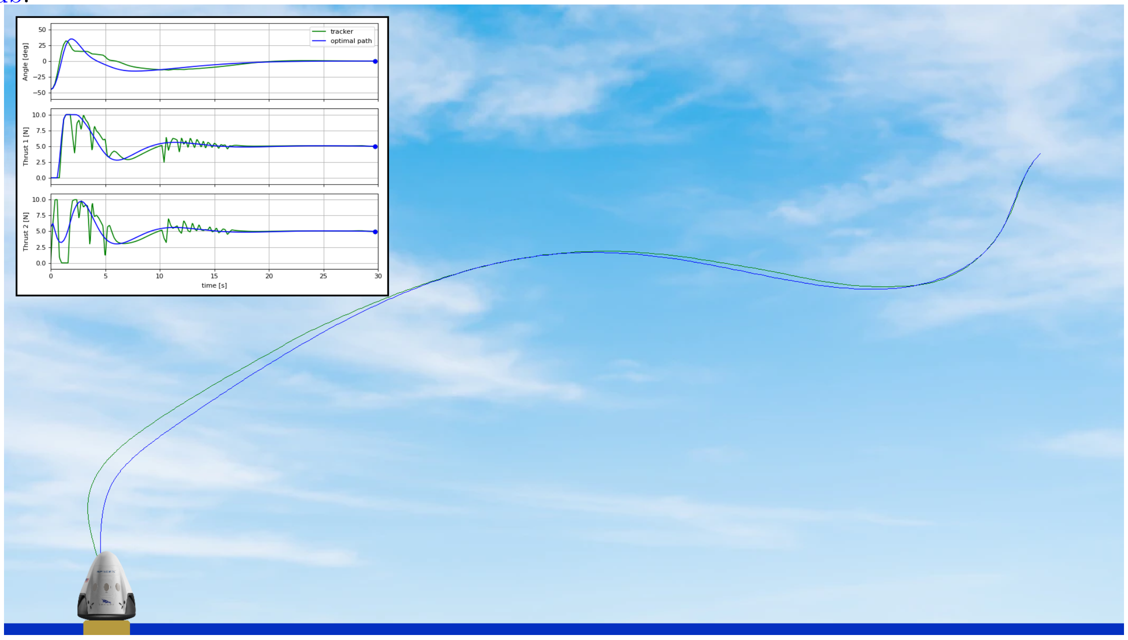

Figure 4: Track 1 Results

9.2 Track 2

For this experiment, we again start at the same position as the last test. However this time the capsule is more upright and descending at a faster speed of 35 m/s ($\underline{x}_0 = \begin{bmatrix} 200 & 100 & -5 & -3.1 & -34.9 & 0 \end{bmatrix}^T$). We see similar results to the first experiment. The linear MPC does a good job tracking and the capsule successfully lands upright. A video of this test is also available for viewing on GitHub.

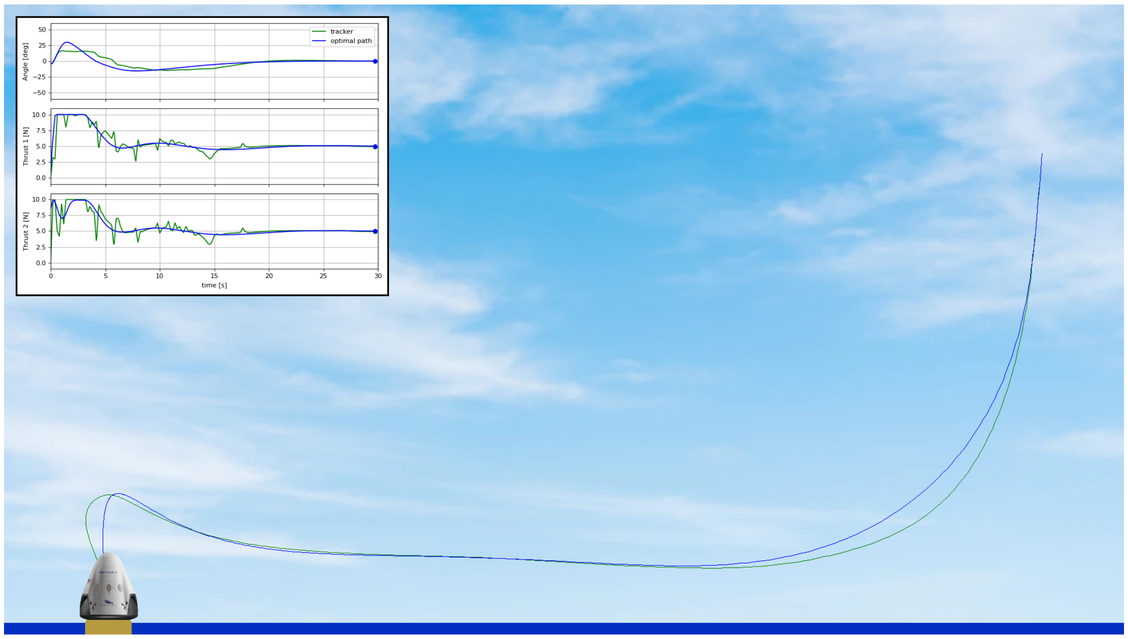

Figure 5: Track 2 Results

10 Where Finite Difference Linearization Breaks Down

I had initially tried to implement the linearized tracking MPC on a rocket with a single gimbaling thruster (figure 6). However, I was not able to get it working. The rocket would track for some time but at some point the optimization would fail to return a solution. I have a working theory as to why things were breaking down but have not been able to test it out.

10.1 Theory

Let's say your gimbaling rocket is at a state where it has some positive angular velocity, $+\dot{\theta}$, and your gimbal angle is much larger than 0 ($\phi \gg 0$). Let's also say that the $\underline{x}_{optimal}$ horizon your MPC optimizer is tracking requires a decrease in angular velocity. You can see from the figure that the gimbal simply needs to be actuated into a negative angle to accomplish this. We can see this, but the MPC cannot. The MPC only knows the state matrices A and B. Recall that we performed finite difference calculations by perturbing each element in $\underline{x}$ and $\underline{u}$ by ϵ. Where $\epsilon \ll 1$. With $\theta \gg 0$, perturbing the system by a small number does not allow the finite difference technique to **see** what would happen to the system if ϕ went negative. Positive and negative perturbations result in a positive gimbal angle. So it can only **see** speeding up $\dot{\theta}$ by a little when reducing ϕ or speeding up $\dot{\theta}$ a lot by increasing ϕ. The MPC optimizer therefore cannot find a tracking solution and fails.
So why does the dragon capsule work? The dragon capsule will never be in a state where small perturbations cant reduce angular velocity. You can see from its freebody diagram that a small positive perturbation on F_1 will always result in a reduced angular velocity. This is only true for the gimbaling rocket when ϕ is near 0.

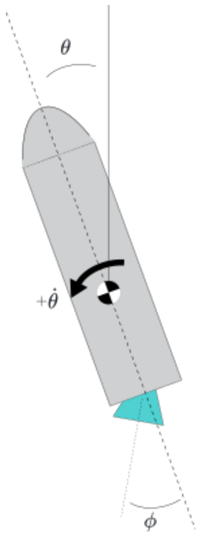

Figure 6: Gimbal Capsule Freebody Diagram

11 Appendix

11.1 get_reference_state

```
def get_reference_state(
    position: np.ndarray) -> np.ndarray:

    nearest_idx = np.argmin(np.linalg.norm(
        optimal_states[:, :2] - position, axis=1))
    idx = np.arange(nearest_idx, nearest_idx + TRACKER_N_HORIZON)
    idx[idx > (optimal_steps - 1)] = optimal_steps - 1
    return optimal_states[idx, :]
```

11.2 dragon_state_function

```
def dragon_state_function(
    t: float,
    x: np.ndarray,
    u: np.ndarray
    gravity_as_input: bool = True) -> np.ndarray:

    t_fwd = u[0] + u[1]
    t_twist = u[1] - u[0]
    g = u[2] if gravity_as_input else G
    sin = np.sin(x[2])
    cos = np.cos(x[2])

    sin = np.sin(x[2])
    cos = np.cos(x[2])

    dx = np.zeros_like(x)
    dx[:3] = x[3:]
    dx[3] = -t_fwd * sin / DRAGON_MASS
    dx[4] = -g + t_fwd * cos / DRAGON_MASS
    dx[5] = DRAGON_THRUST_ARM / DRAGON_I * t_twist
    return dx
```

11.3 get_linearized_state

```
def get_linearized_state(
    x: np.ndarray,
    u: np.ndarray,
    sample_time: float,
    epsilon: float,
    state_function: Callable) -> Tuple[np.ndarray, np.ndarray]:

    x = np.array(x).ravel()
    u = np.array(u).ravel()

    nx, nu = x.shape[0], u.shape[0]
    x_eps, u_eps = np.eye(nx) * epsilon, np.eye(nu) * epsilon

    x_plus = (np.tile(x, (nx, 1)) + x_eps).T
    x_minus = (np.tile(x, (nx, 1)) - x_eps).T
    u_plus = (np.tile(u, (nu, 1)) + u_eps).T
    u_minus = (np.tile(u, (nu, 1)) - u_eps).T
    states_plus, states_minus = np.zeros((nx, nx)), np.zeros((nx, nx))
    inputs_plus, inputs_minus = np.zeros((nx, nu)), np.zeros((nx, nu))
    for i in range(nx):

        states_plus[:, i] = solve_ivp(state_function,
                                      t_span=[0, sample_time],
                                      y0=x_plus[:, i],
                                      args=(u,)).y[:, -1]
        states_minus[:, i] = solve_ivp(state_function,
                                       t_span=[0, sample_time],
                                       y0=x_minus[:, i],
                                       args=(u,)).y[:, -1]
    for i in range(nu):

        inputs_plus[:, i] = solve_ivp(state_function,
                                      t_span=[0, sample_time],
                                      y0=x,
                                      args=(u_plus[:, i],)).y[:, -1]
        inputs_minus[:, i] = solve_ivp(state_function,
                                       t_span=[0, sample_time],
                                       y0=x,
                                       args=(u_minus[:, i],)).y[:, -1]
    A = (states_plus - states_minus) / (2 * epsilon)
    B = (inputs_plus - inputs_minus) / (2 * epsilon)
    return A, B
```

References

[Fer17] Reuben Ferrante. *A Robust Control Approach for Rocket Landing.* PhD thesis, 2017.

[Mat] Land a rocket using multistage nonlinear mpc. https://www.mathworks.com/help/mpc/ug/landing-rocket-with-mpc-example.html. Accessed: 2022-11-18.

[SA18] Michael Szmuk and Behçet Açıkmeşe. *Successive Convexification for 6-DoF Mars Rocket Powered Landing with Free-Final-Time.* Curran Associates, Inc, 2018.