EL2700 - Assignment 4: Model Predictive Control

Group 14: Chieh-Ju Wu and Charles Brinkley

September 28, 2021

1 Part I: MPC Setup

1.1 Q1: Study the influence of the control horizon

In general, by increasing the horizon length N, we also in turn increase the available N-step controllable sets. Figures 1 - 3 display how maximum control invariant sets for translation subsystem increase with the increment of control horizon.

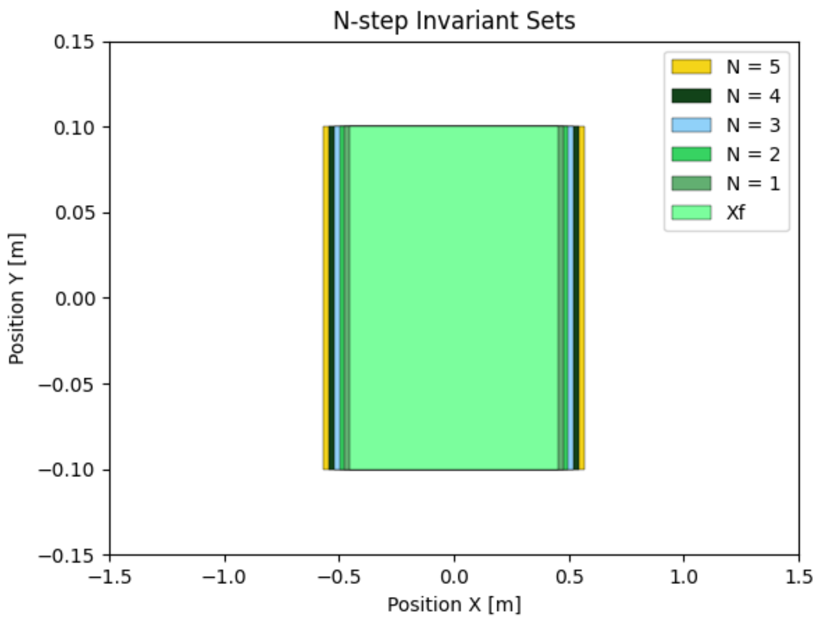


Figure 1: Controllable set of $\mathcal{K}_N(\mathcal{X}_N)$ - Translation (N=5)

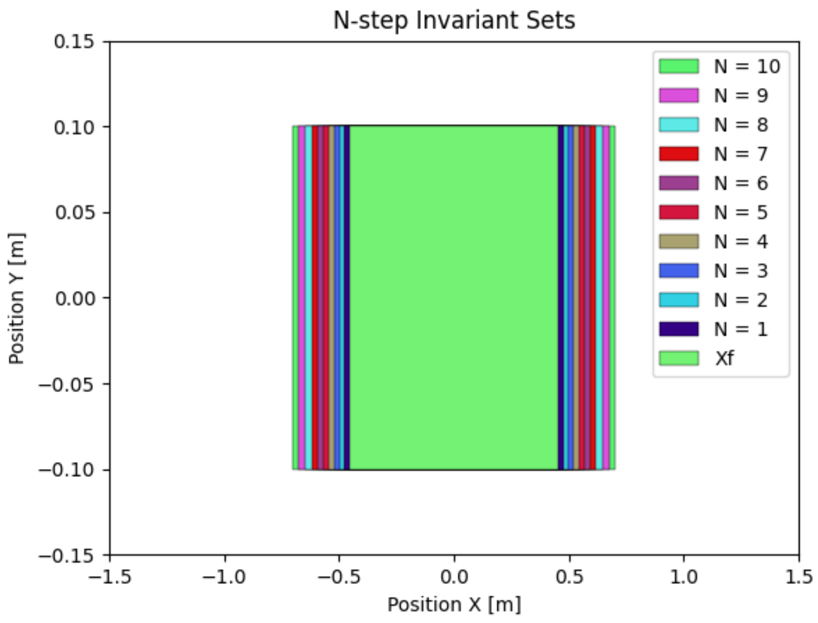


Figure 2: Controllable set of $\mathcal{K}_N(\mathcal{X}_N)$ - Translation (N=10)

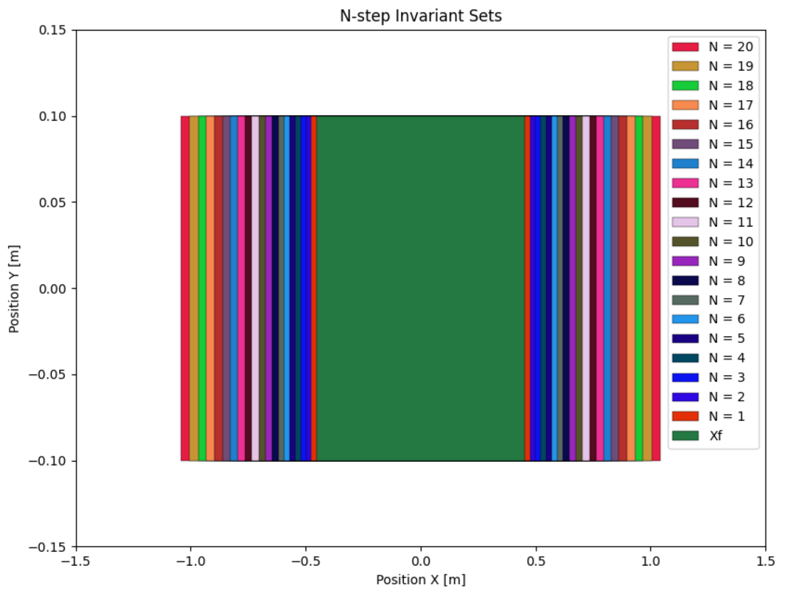


Figure 3: Controllable set of $\mathcal{K}_N(\mathcal{X}_N)$ - Translation (N=20)

The same case can be seen when analysing the attitude N-step controllable sets when the terminal constraint is set to zero. As the N-step horizon increases, so does the range of control invariant sets. Figure 4 shows an initial range of +/- (.04,.03) and increases to +/- (.19,.19) when increased to N=20. The longer the horizon, the bigger the sets of admissible initial states can be used until it reaches its determindness index (ν), which is when the control invariant sets converge.

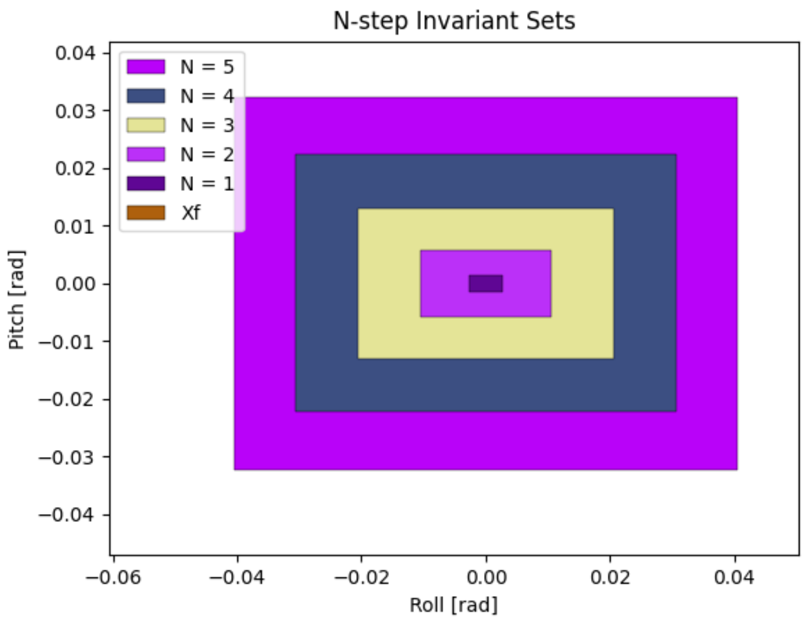


Figure 4: Controllable set of $\mathcal{K}_N(\mathcal{X}_N)$-Attitude (N=5)

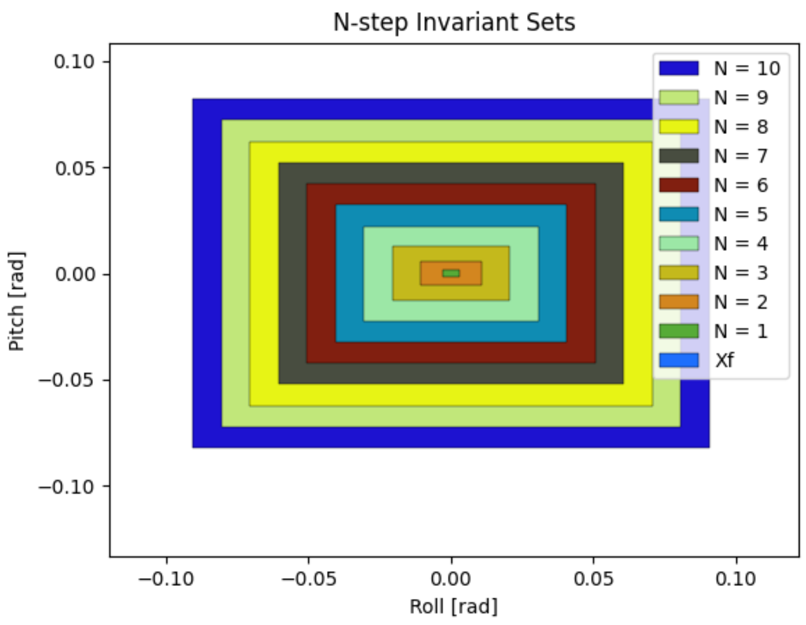


Figure 5: Controllable set of $\mathcal{K}_N(\mathcal{X}_N)$-Attitude (N=10)

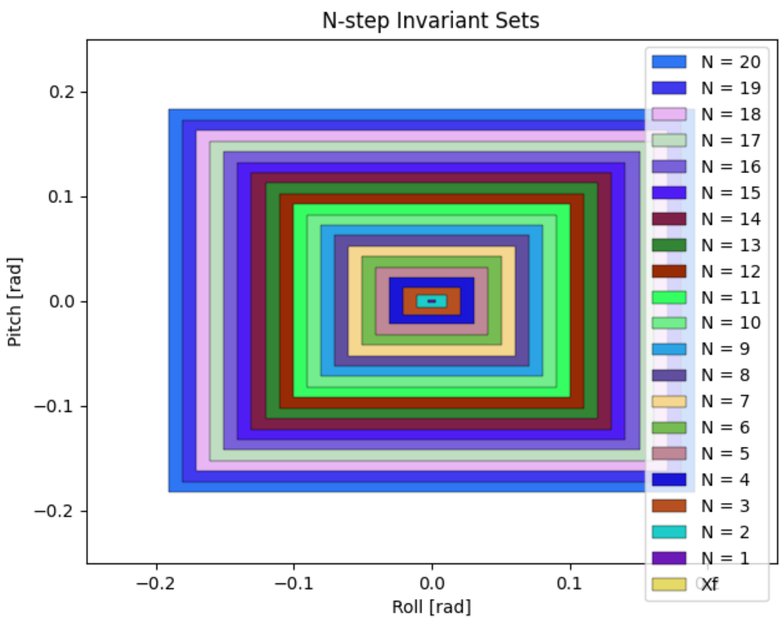


Figure 6: Controllable set of $\mathcal{K}_N(\mathcal{X}_N)$-Attitude (N=20)

1.2 Q2: Influence of control constraints

Stricter constraints (smaller range) of u leads to smaller state control invariant set. By increasing it 3x, we can see that the invariant set increases from +/-.55 to +/-1.25

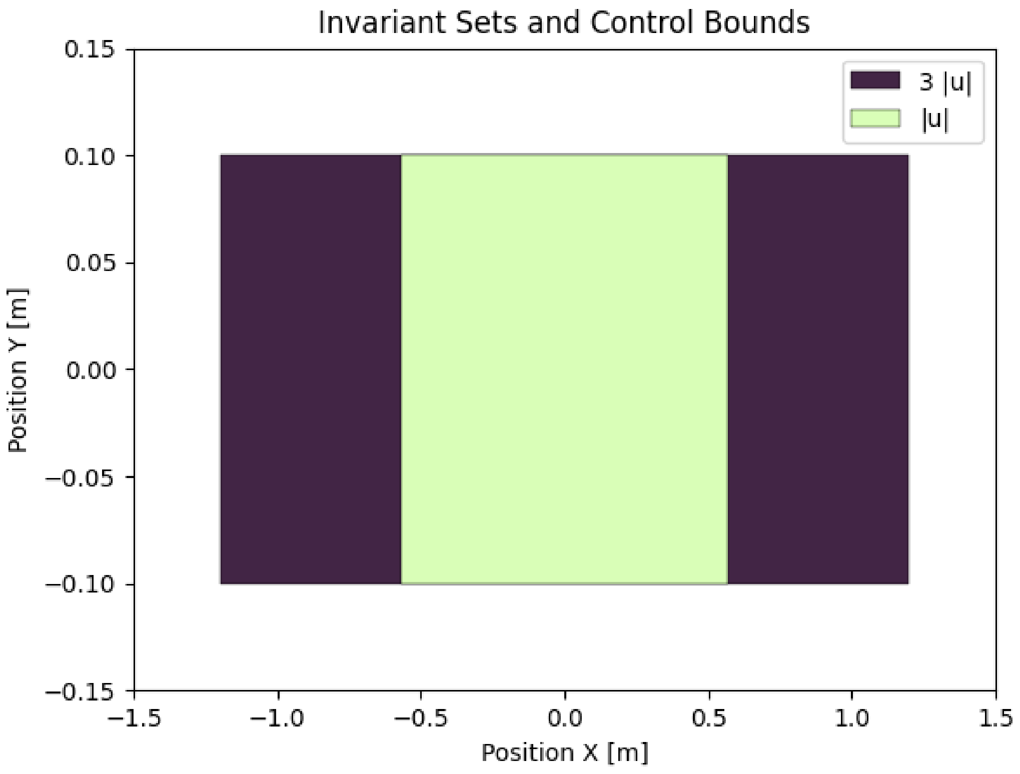


Figure 7: Controllable set of $\mathcal{K}_N(\mathcal{X}_N)$ with N=5 - comparison between different constraints of u

2 Part II: The MPC Implementation

2.1 Q3: Examining MPC Controller Implementation

2.1.1 How the state constraints are set

This part of the code separates into two steps - setting the upper boundary and lower boundary. If xub is not "None", we set the upper boundary of the states according to the desired value and lower boundary to negative infinite. Which should result in equation 2.1.

$$[-inf,\ -inf,\ ...] \leqq [states] \leqq [Upper\ Boundary] \tag{2.1}$$

If xlb is not None, we set the lower boundary of the states according to the desired value and upper boundary to positive infinite. Which should result in equation 2.2.

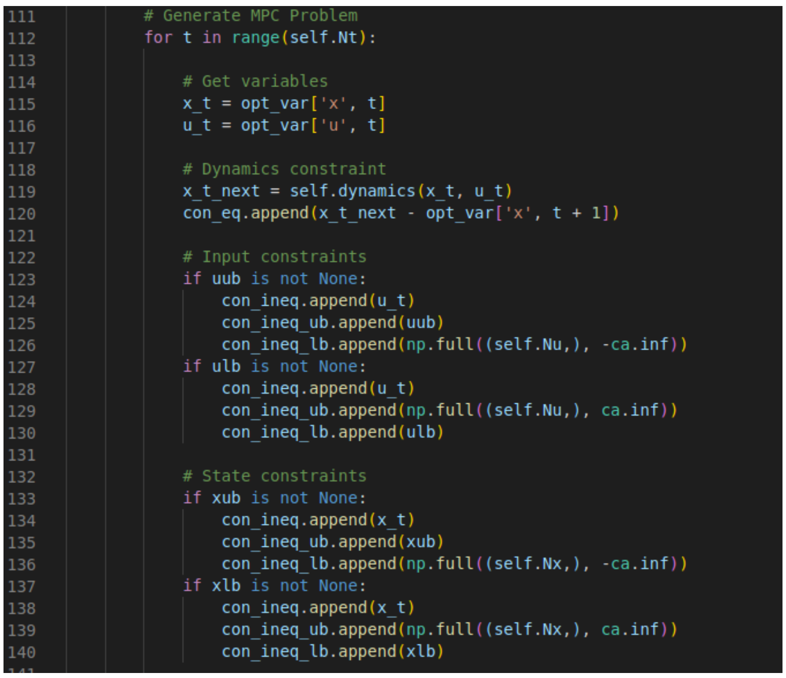
$$[Lower\ Boundary] \leqq [states] \leqq [inf,\ inf,\ ...] \tag{2.2}$$

With the above two constrains set, we have the state constraints as equation 2.3. Since desired upper boundary is smaller than positive infinite and desired lower boundary is bigger than negative infinite, we can simplified the state constraints to equation 2.4.

$$[-inf,\ -inf,\ ...] \leqq [Lower\ Boundary] \leqq [states] \leqq [Upper\ Boundary] \leqq [inf,\ inf,\ ...] \tag{2.3}$$

$$[Lower\ Boundary] \leqq [states] \leqq [Upper\ Boundary] \tag{2.4}$$

This methodology of setting the constraints is represented in the python script below in figure 8.



```python
        # Generate MPC Problem
        for t in range(self.Nt):

            # Get variables
            x_t = opt_var['x', t]
            u_t = opt_var['u', t]

            # Dynamics constraint
            x_t_next = self.dynamics(x_t, u_t)
            con_eq.append(x_t_next - opt_var['x', t + 1])

            # Input constraints
            if uub is not None:
                con_ineq.append(u_t)
                con_ineq_ub.append(uub)
                con_ineq_lb.append(np.full((self.Nu,), -ca.inf))
            if ulb is not None:
                con_ineq.append(u_t)
                con_ineq_ub.append(np.full((self.Nu,), ca.inf))
                con_ineq_lb.append(ulb)

            # State constraints
            if xub is not None:
                con_ineq.append(x_t)
                con_ineq_ub.append(xub)
                con_ineq_lb.append(np.full((self.Nx,), -ca.inf))
            if xlb is not None:
                con_ineq.append(x_t)
                con_ineq_ub.append(np.full((self.Nx,), ca.inf))
                con_ineq_lb.append(xlb)
```

Figure 8: Setting up the state constraints for MPC problem

2.1.2 How the object function is formulated

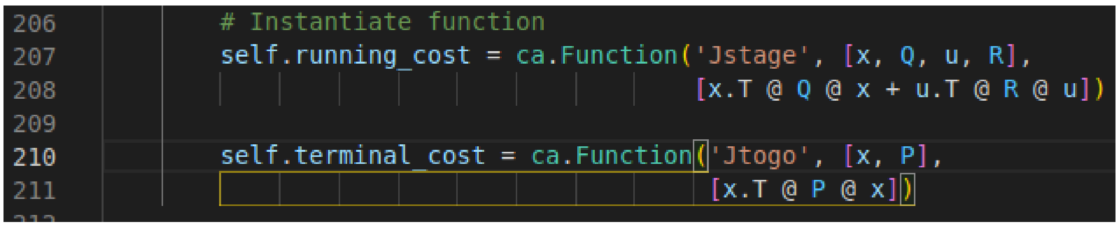
The object function is formulated as equation 2.5, where the infinite-horizon cost is transformed into finite-horizon cost with a tail. The variable *running_cost* corresponds to the control horizon, meanwhile, the variable *terminal_cost*, which is a quadratic cost-to-go function (equation 2.6), is used to approximate the cost after the control horizon. Since this is a reference tracking MPC problem, the x and u term here are defined as the deviations from (x^{ref}, u^{ref}) as equation 2.7.

$$J = \min \sum_{t=0}^{\infty} \Delta x_t^T Q \Delta x_t \; + \; u_t^T R u_t \; = \min \sum_{t=0}^{T-1} \Delta x_t^T Q \Delta x_t \; + \; u_t^T R u_t \; + \; \hat{v}(x_T) \tag{2.5}$$

$$\hat{v}(x_T) \; = \; \hat{x}_T^T P \hat{x}_T \tag{2.6}$$

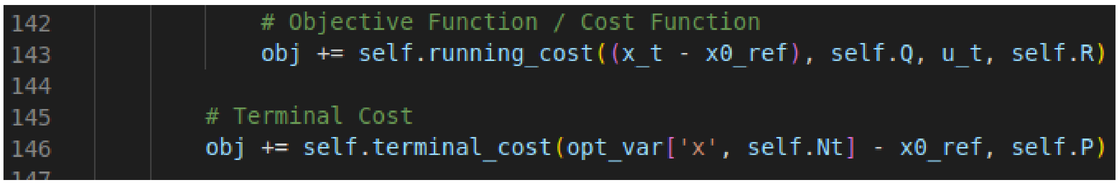
$$\Delta x_t \coloneqq \hat{x}_t - x^{ref} \tag{2.7}$$

Figure 9 shows the initialization of these objective cost functions (running cost and terminal cost) to be used in Casadi. The figure below shows how the summation occurs after each step in t recursively.



```
        # Instantiate function
        self.running_cost = ca.Function('Jstage', [x, Q, u, R],
                                        [x.T @ Q @ x + u.T @ R @ u])

        self.terminal_cost = ca.Function('Jtogo', [x, P],
                                         [x.T @ P @ x])
```

Figure 9: Defining stage and cost-to-go in Casadi function

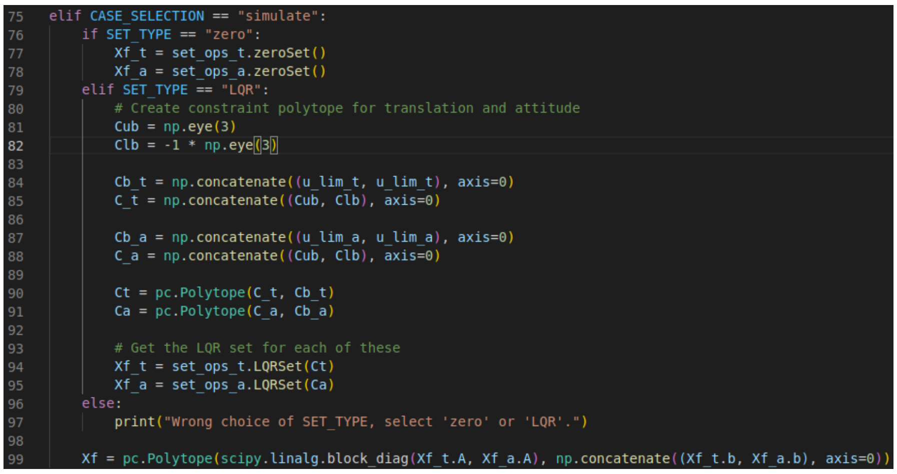


```
            # Objective Function / Cost Function
            obj += self.running_cost((x_t - x0_ref), self.Q, u_t, self.R)

        # Terminal Cost
        obj += self.terminal_cost(opt_var['x', self.Nt] - x0_ref, self.P)
```

Figure 10: Updating sum of costs for each time 't' with horizon

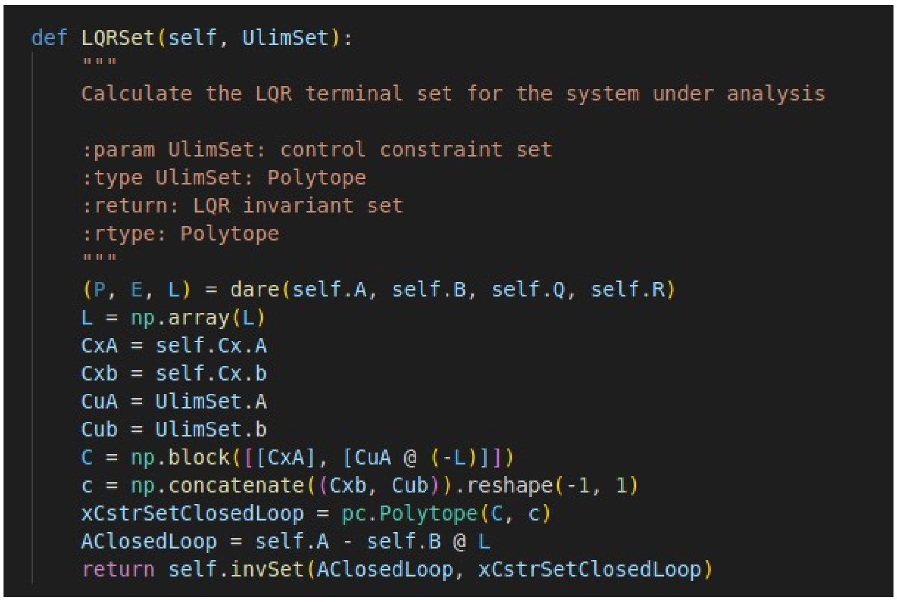
2.1.3 How the terminal constraint is set

The control invariant set polyhydra is first created from the control and state limits and saved as ”Ct” and ”Ca” (see Figure 11). These are sent to the ”LQRSet” function in set_operations.py. While in this function, a polyhedra is formed using the closed loop dynamics (derived from the LQR solution) and the the earlier polyhedra formed (formed Ct and Ca). This is shown in figure 12.



```python
elif CASE_SELECTION == "simulate":
    if SET_TYPE == "zero":
        Xf_t = set_ops_t.zeroSet()
        Xf_a = set_ops_a.zeroSet()
    elif SET_TYPE == "LQR":
        # Create constraint polytope for translation and attitude
        Cub = np.eye(3)
        Clb = -1 * np.eye(3)

        Cb_t = np.concatenate((u_lim_t, u_lim_t), axis=0)
        C_t = np.concatenate((Cub, Clb), axis=0)

        Cb_a = np.concatenate((u_lim_a, u_lim_a), axis=0)
        C_a = np.concatenate((Cub, Clb), axis=0)

        Ct = pc.Polytope(C_t, Cb_t)
        Ca = pc.Polytope(C_a, Cb_a)

        # Get the LQR set for each of these
        Xf_t = set_ops_t.LQRSet(Ct)
        Xf_a = set_ops_a.LQRSet(Ca)
    else:
        print("Wrong choice of SET_TYPE, select 'zero' or 'LQR'.")

    Xf = pc.Polytope(scipy.linalg.block_diag(Xf_t.A, Xf_a.A), np.concatenate((Xf_t.b, Xf_a.b), axis=0))
```

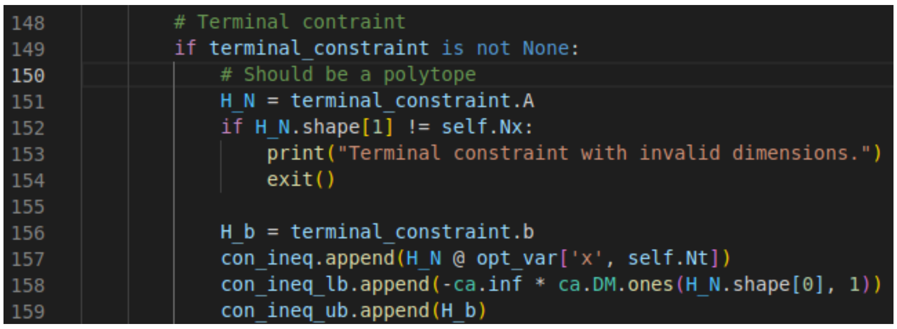
Figure 11



```python
def LQRSet(self, UlimSet):
    """
    Calculate the LQR terminal set for the system under analysis

    :param UlimSet: control constraint set
    :type UlimSet: Polytope
    :return: LQR invariant set
    :rtype: Polytope
    """
    (P, E, L) = dare(self.A, self.B, self.Q, self.R)
    L = np.array(L)
    CxA = self.Cx.A
    Cxb = self.Cx.b
    CuA = UlimSet.A
    Cub = UlimSet.b
    C = np.block([[CxA], [CuA @ (-L)]])
    c = np.concatenate((Cxb, Cub)).reshape(-1, 1)
    xCstrSetClosedLoop = pc.Polytope(C, c)
    AClosedLoop = self.A - self.B @ L
    return self.invSet(AClosedLoop, xCstrSetClosedLoop)
```

Figure 12

After this, these polytopes are passed to the ”invSet” function that intersects these two polytopes and iterates until convergence occurs (resulting in determindness index ν). Finally, the LQR terminal set is sent into mpc.py where the constraint is set as $\mathcal{X} = \{x : H_x X \leq h_x\}$, meanwhile satisfying $\mathcal{U} = u : H_u X \leq h_u$ (figure 13).

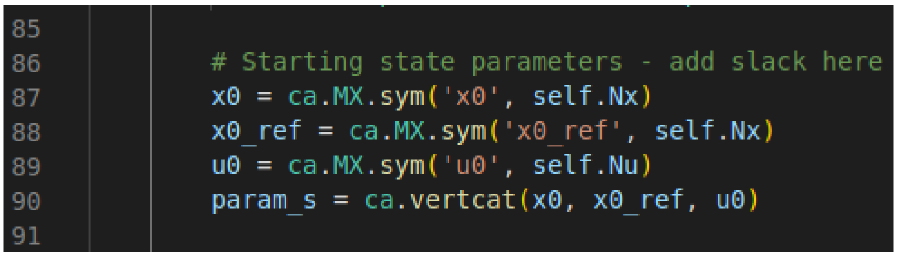


```python
        # Terminal contraint
        if terminal_constraint is not None:
            # Should be a polytope
            H_N = terminal_constraint.A
            if H_N.shape[1] != self.Nx:
                print("Terminal constraint with invalid dimensions.")
                exit()

            H_b = terminal_constraint.b
            con_ineq.append(H_N @ opt_var['x', self.Nt])
            con_ineq_lb.append(-ca.inf * ca.DM.ones(H_N.shape[0], 1))
            con_ineq_ub.append(H_b)
```

Figure 13

2.1.4 What the variable ”param_s” holds

The variable param_s is defined as $[x_0,\, x_{0ref},\, u_0]$, which is the initial value of the system state, reference state and control input. The decleration in the code can be seen in the snippet of figure 14.

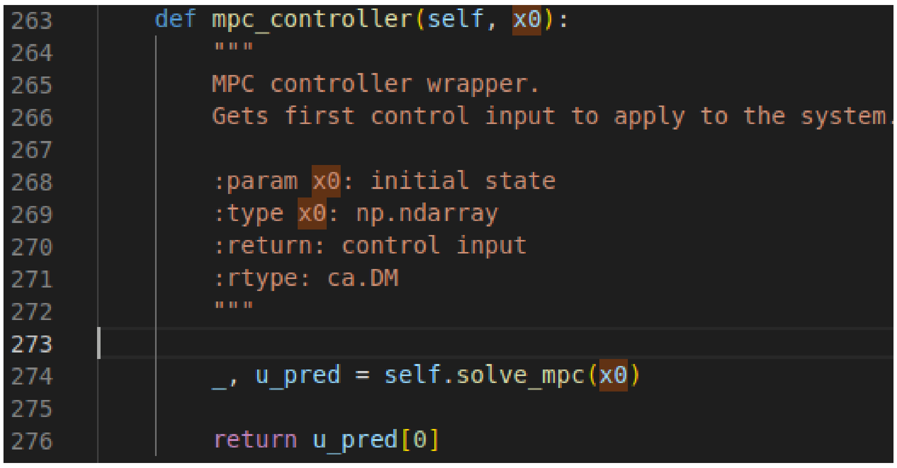


```python

        # Starting state parameters - add slack here
        x0 = ca.MX.sym('x0', self.Nx)
        x0_ref = ca.MX.sym('x0_ref', self.Nx)
        u0 = ca.MX.sym('u0', self.Nu)
        param_s = ca.vertcat(x0, x0_ref, u0)

```

Figure 14: Initial state, input, and reference parameters

2.1.5 Why we only select the first element of the control prediction horizon

Because we are using receding horizon control. We recursively measure states, plan optimal controls over a finite-horizon, implement first control action, and wait until next sampling instance. The x_0 seen in this function updates each iteration and thus returns the first optimal predicted control input.



```
    def mpc_controller(self, x0):
        """
        MPC controller wrapper.
        Gets first control input to apply to the system.

        :param x0: initial state
        :type x0: np.ndarray
        :return: control input
        :rtype: ca.DM
        """

        _, u_pred = self.solve_mpc(x0)

        return u_pred[0]
```

Figure 15: Code screenshot showing the "mpc_controller" function

Figure 16 shows the sequence of the measure → plan → do sequence utilized to implement the MPC controller. Below that is the code that shows how the state is measured, the optimal control input is calculated and the next state is thus calculated.

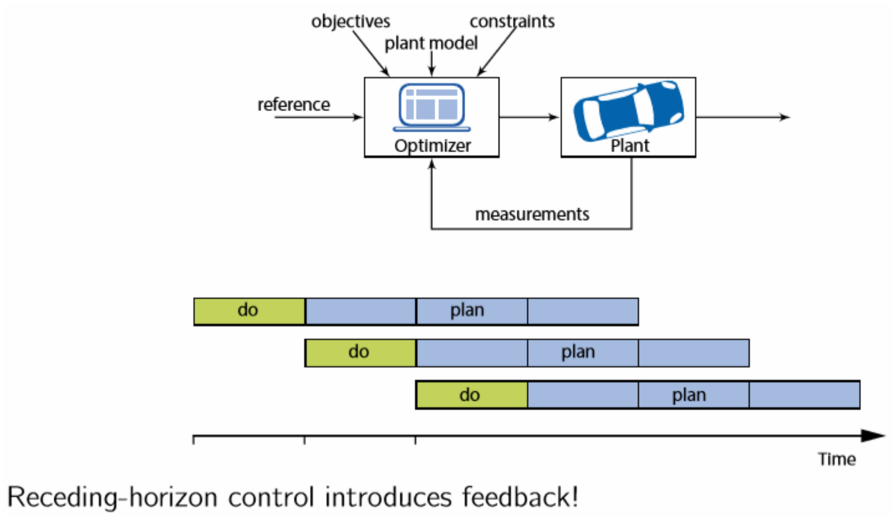
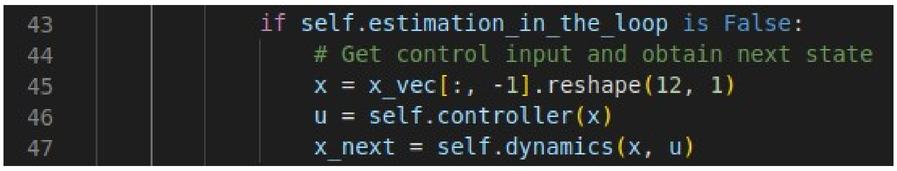


Figure 16: Sequence diagram showing the receding-horizon control sequence



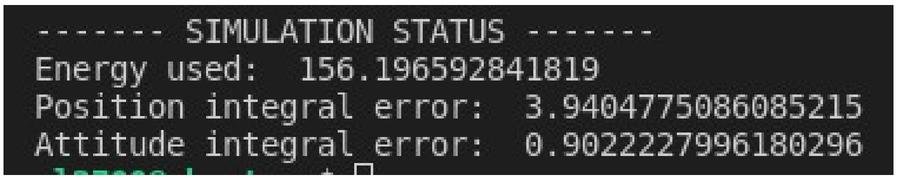
```
if self.estimation_in_the_loop is False:
    # Get control input and obtain next state
    x = x_vec[:, -1].reshape(12, 1)
    u = self.controller(x)
    x_next = self.dynamics(x, u)
```

Figure 17: Latest measured state being fed into mpc controller for next optimal control action

3 Part III - MPC for Stabilization

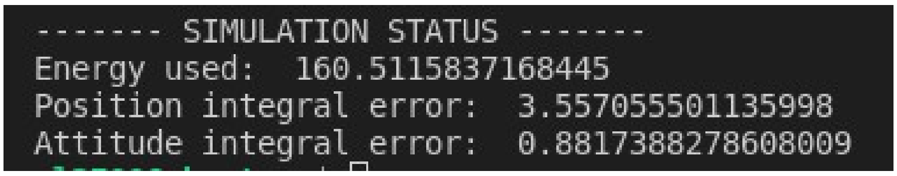
3.1 Q4: Comparing performance for $3 * u_{lim}$

From our intuition, we deduct that the system with stricter control constraints will consume less energy at the cost of slower system response, which further leads to bigger integral errors. The simulation which are presented in Figure 18, 19 support our deduction.



```
------- SIMULATION STATUS -------
Energy used:  156.196592841819
Position integral error:  3.9404775086085215
Attitude integral error:  0.9022227996180296
```

Figure 18: Energy consumption and position/attitude integral errors for original control constraints



```
------- SIMULATION STATUS -------
Energy used:  160.5115837168445
Position integral error:  3.557055501135998
Attitude integral error:  0.8817388278608009
```

Figure 19: Energy consumption and position/attitude integral errors for looser control constraints

3.2 Q5: Comparing with terminal sets

- SET_TYPE= "zero", CASE_SELECTION= "simulation"

  By setting the terminal constraint to zero states, we try to drive the system state to the reference at the end of the control horizon T. However, the small terminal sets reduces the set of admissible initial states for the controller. In our case, as we can see from Figure 20, the given initial state is outside the invariant sets, thus, solving the MPC planning problem become infeasible (Figure 21).

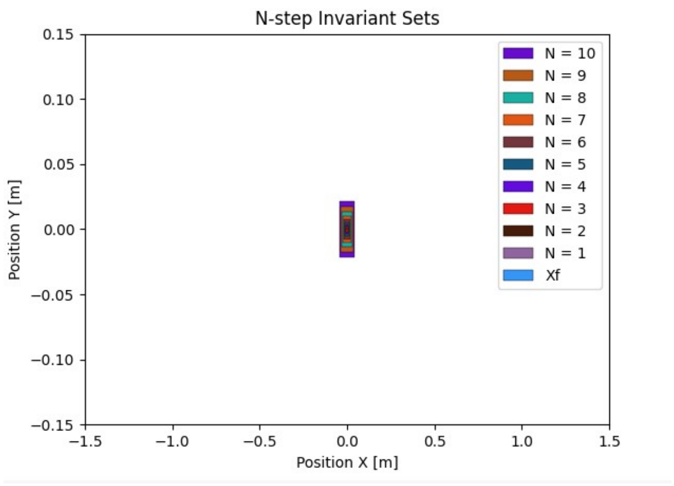
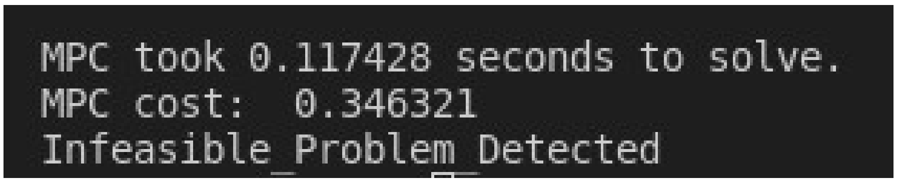


Figure 20: Control invariant - SET_TYPE= "zero", CASE_SELECTION= "translation"



```
MPC took 0.117428 seconds to solve.
MPC cost:  0.346321
Infeasible_Problem_Detected
```

Figure 21: MPC planning problem is not feasible

- SET_TYPE= "LQR", CASE_SELECTION= "simulation"

  Here, we add in the terminal constraints $\mathcal{X}_f^{LQR}$. This allows a larger control invariant sets for our initial state. As shown in Figure 22, the given initial state is now within the control invariant sets. Hence, the MPC planning problem become feasible. The solution can be seen in Figure Since now $X_T \subseteq X$ is control invariant for $x_{t+1} = Ax_t + Bu_t$ under the control constraint $u_t \in U$, then the MPC planning problem is recursively feasible from all initial states x_0, which admits a feasible solution. Thus, the solution can be seen in Figure 23

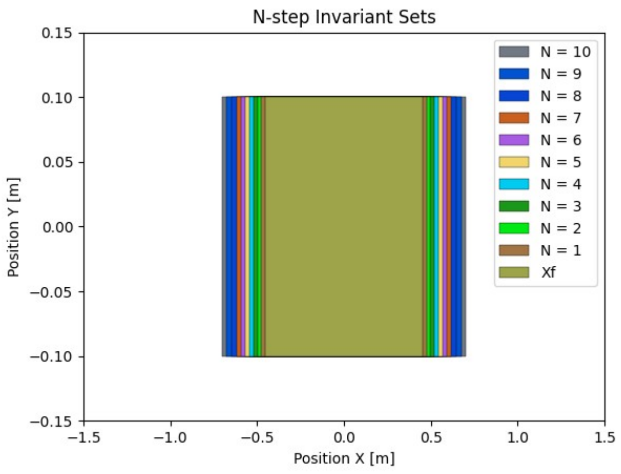
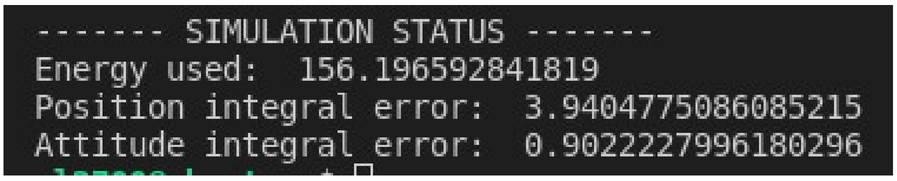


Figure 22: Control invariant - SET_TYPE= "LQR", CASE_SELECTION= "translation"

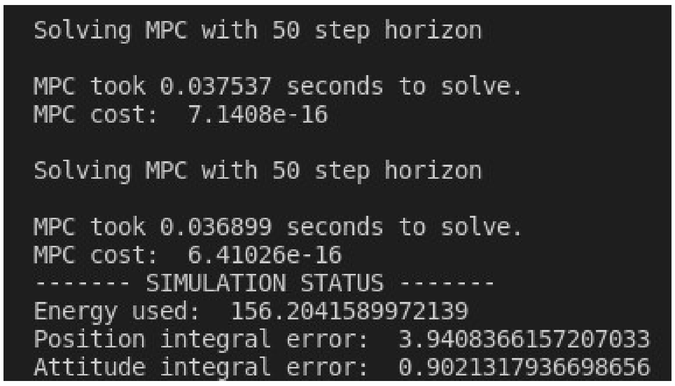


```
------- SIMULATION STATUS -------
Energy used:  156.196592841819
Position integral error:  3.9404775086085215
Attitude integral error:  0.9022227996180296
```

Figure 23: MPC planning problem becomes feasible with new terminal constraints

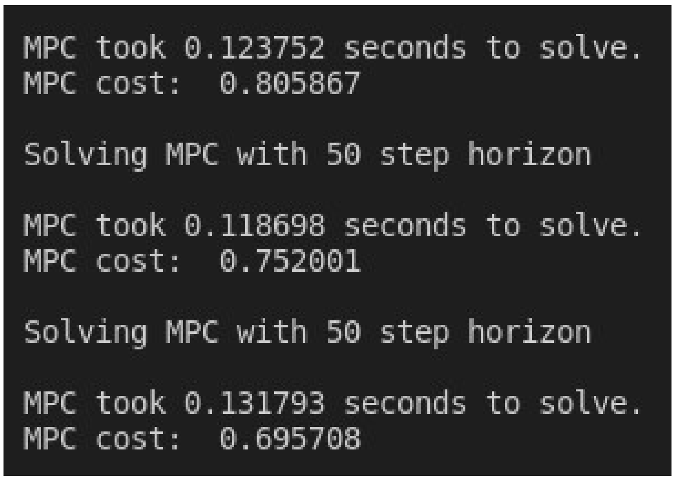
3.3 Q6: Comparing with $N = 50$ control horizon

By increasing the MPC horizon to 50 while keeping $\mathcal{X}_f = 0$, we are able to solve the planning problem. The solving time started from 0.13 sec/iter, but as the parameters of the LQR controller becomes optimized and get closer to the converged value, the time required decreases with the receding-horizon. At the end of the iteration, the solving time dropped down to approximately 0.04 sec/iter.



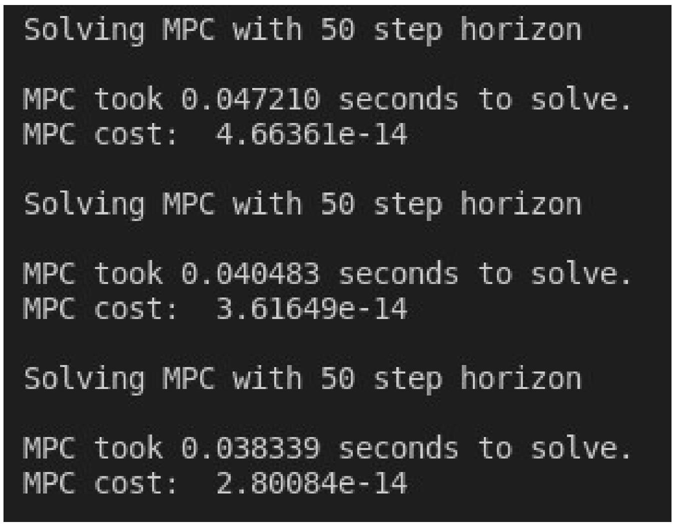
```
Solving MPC with 50 step horizon

MPC took 0.037537 seconds to solve.
MPC cost:  7.1408e-16

Solving MPC with 50 step horizon

MPC took 0.036899 seconds to solve.
MPC cost:  6.41026e-16
------- SIMULATION STATUS -------
Energy used:  156.2041589972139
Position integral error:  3.9408366157207033
Attitude integral error:  0.9021317936698656
```

Figure 24: MPC planning solved with horizon 50



```
MPC took 0.123752 seconds to solve.
MPC cost:  0.805867

Solving MPC with 50 step horizon

MPC took 0.118698 seconds to solve.
MPC cost:  0.752001

Solving MPC with 50 step horizon

MPC took 0.131793 seconds to solve.
MPC cost:  0.695708
```

Figure 25: Each iteration takes approx. 0.13 second at start



```
Solving MPC with 50 step horizon

MPC took 0.047210 seconds to solve.
MPC cost:  4.66361e-14

Solving MPC with 50 step horizon

MPC took 0.040483 seconds to solve.
MPC cost:  3.61649e-14

Solving MPC with 50 step horizon

MPC took 0.038339 seconds to solve.
MPC cost:  2.80084e-14
```

Figure 26: Each iteration takes approx. 0.13 second in the end

3.4 Q7: Comparing effects of tuning parameters

3.4.1 Multiplying R by 10

By increasing the size of R by 10 times, we make the control input become more expensive. The LQR controller will thus have smaller control input, which will further result into lower total energy consumption (Figure 29) at the cost of slower system (Figure 27, 28)

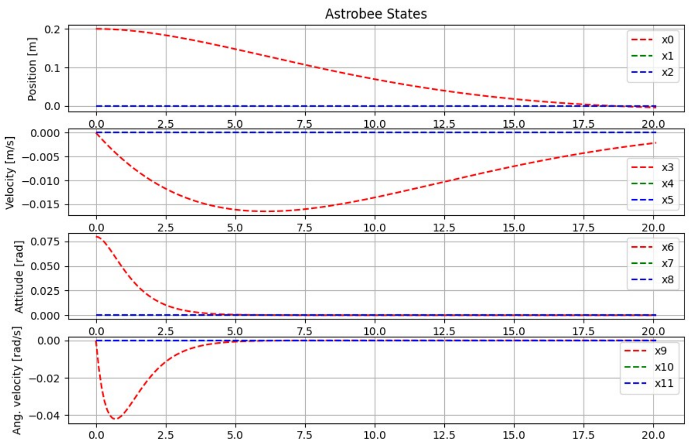


Figure 27: Astrobee states plot - Multiply R by 10

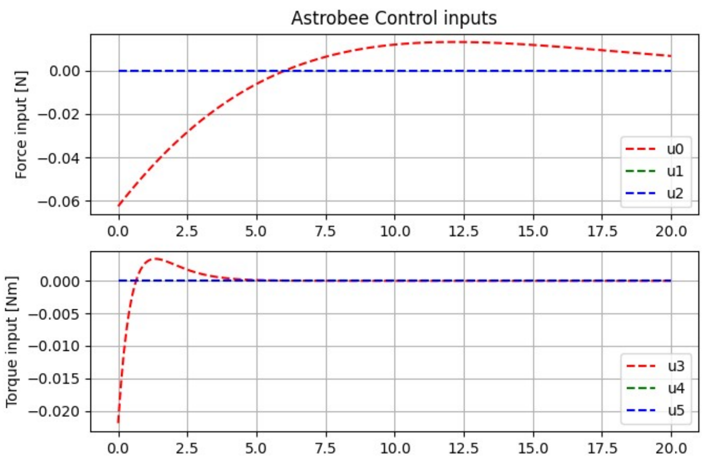
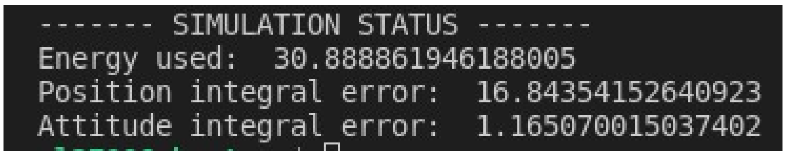


Figure 28: Astrobee control inputs - Multiply R by 10



```
------- SIMULATION STATUS -------
Energy used:  30.888861946188005
Position integral error:  16.84354152640923
Attitude integral error:  1.165070015037402
```

Figure 29: Total energy consumption - Multiply R by 10

3.4.2 Adding 100 to velocity components

By adding 100 to the velocity components of matrix Q, we try to decrease the time required for the system's speed to reach to the reference speed, which are zero. Comparing Figure 27 and 30, we can see that both the translational and angular velocity of the system takes less time to reach the reference speed. Since the both translational and angular reference speed is zero, we would need less energy as seen in Figure 32. However, this is at the cost of sacrificing the time length of reaching the position reference.

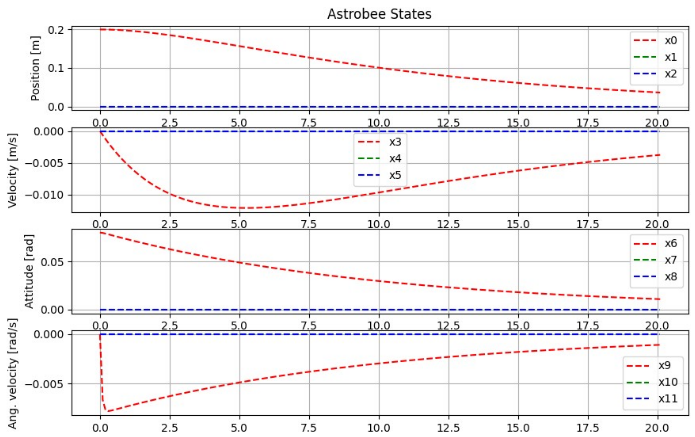


Figure 30: Astrobee states plot - Add velocity components by 100

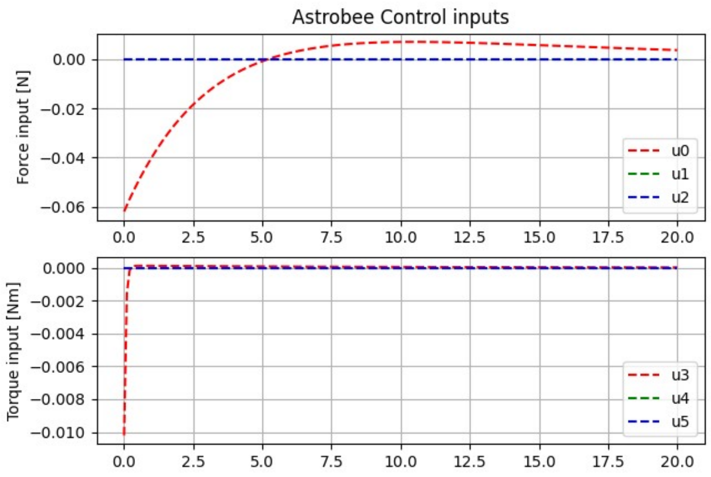
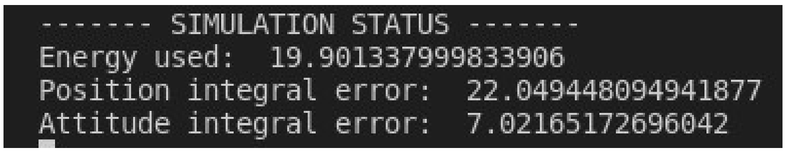


Figure 31: Astrobee control inputs - Add velocity components by 100



```
------- SIMULATION STATUS -------
Energy used:  19.901337999833906
Position integral error:  22.049448094941877
Attitude integral error:  7.02165172696042
```

Figure 32: Total energy consumption - Add velocity components by 100

3.4.3 Adding 100 to position and attitude

Reverting the velocity components to 1 and adding 100 to position and attitude components of matrix Q, we try to decrease the time required for the system's position and attitude to reach the reference, which are zero. Comparing Figure 27 and 33, we can see that both the position and attitude of the system takes less time to reach the reference. To enable this, we would need larger control input which will cause a higher total energy consumption as seen in Figure 35. We also notice that there is a slight undershoot for both position and attitude. We propose that we could try solving this issue by softening the position and attitude constraints with slack variables.

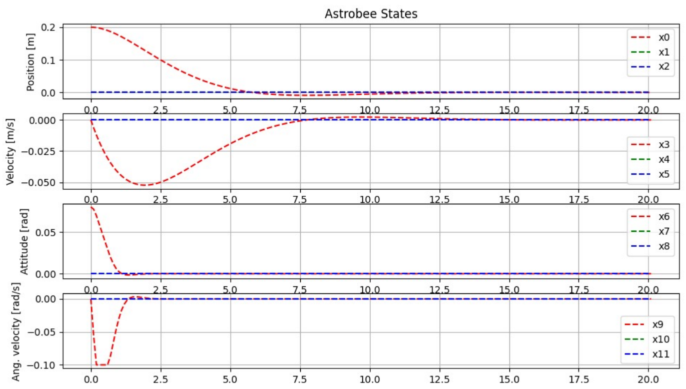


Figure 33: Astrobee states plot - Add position and attitude components by 100

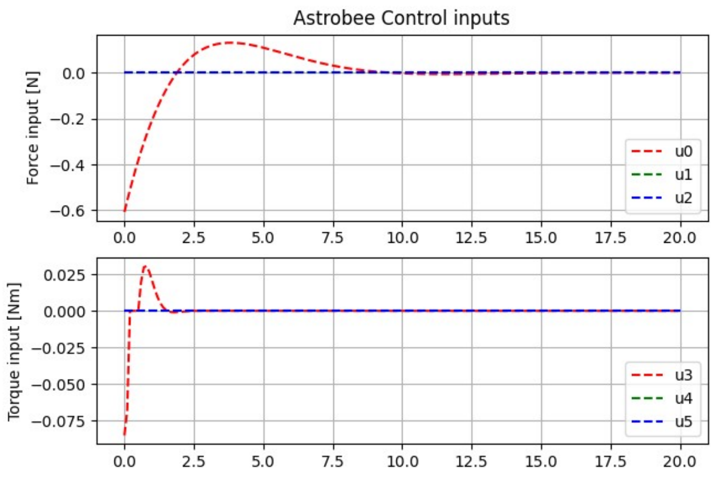
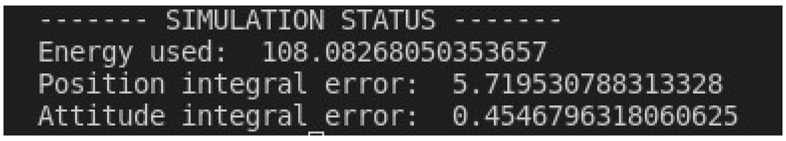


Figure 34: Astrobee control inputs - Add position and attitude components by 100



```
------- SIMULATION STATUS -------
Energy used:  108.08268050353657
Position integral error:  5.719530788313328
Attitude integral_error:  0.4546796318060625
```

Figure 35: Total energy consumption - Add position and attitude components by 100

3.4.4 Increasing all elements of Q by 100

By adding 100 to all components of matrix Q, we try to decrease the time required for all system states to reach the reference states, which is zero. Comparing Figure 27 and 36, we can see that all the system's states (position, attitude, translational and angular velocity) take less time to reach the reference states. The total energy consumption is slightly decreased (Figure 38) compared to 35 due to the added penalties to the velocities components in Q matrix.

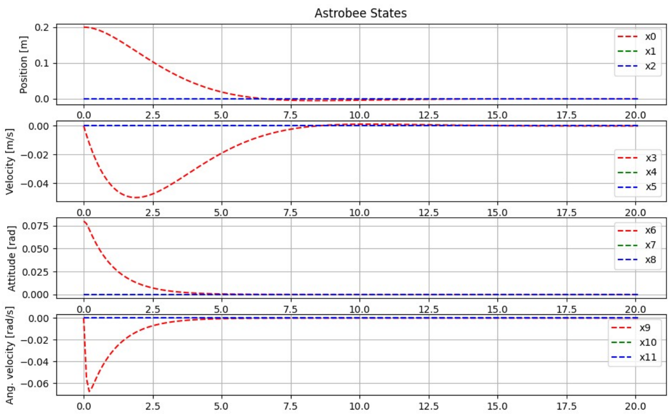


Figure 36: Astrobee states plot - Add all Q components by 100

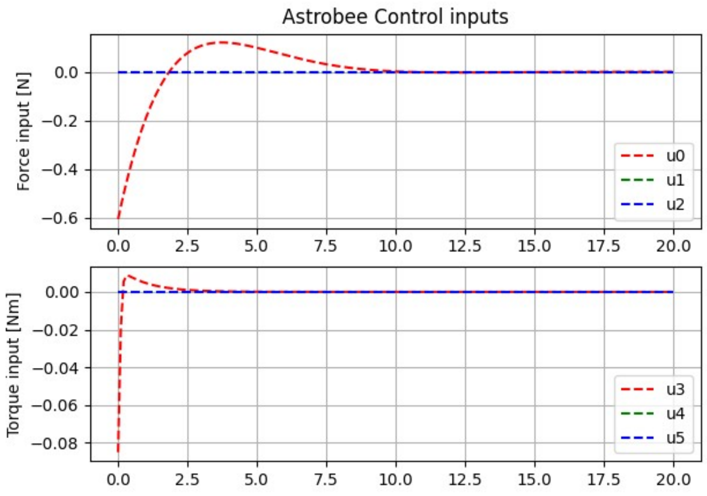
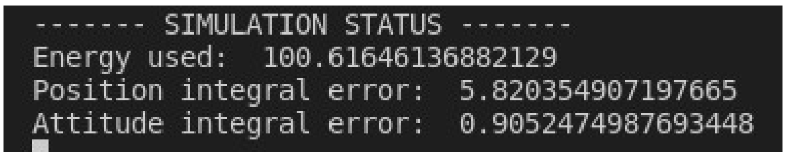


Figure 37: Astrobee control inputs - Add all Q components by 100



```
------- SIMULATION STATUS -------
Energy used:  100.61646136882129
Position integral error:  5.820354907197665
Attitude integral error:  0.9052474987693448
```

Figure 38: Total energy consumption - Add all Q components by 100