Assignment 4

Konrad Dittrich and Simon Spång

Q1:

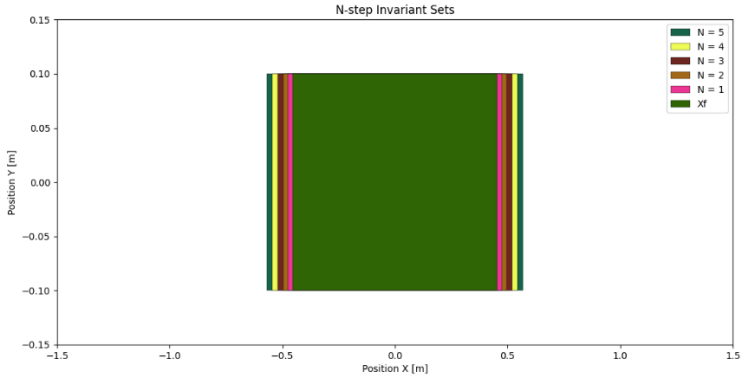

Figure 1 - N = 5 steps

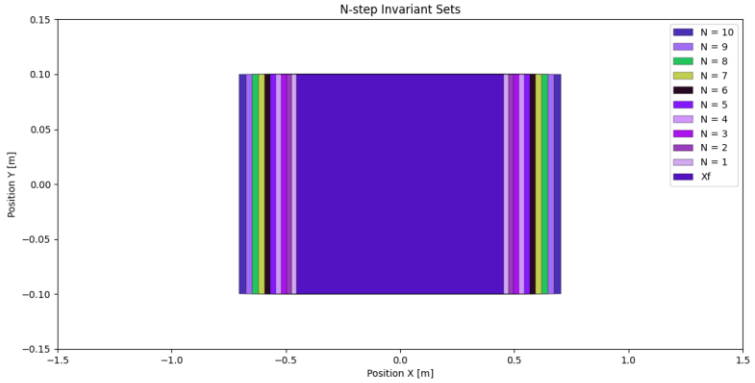

Figure 2 - N = 10 steps

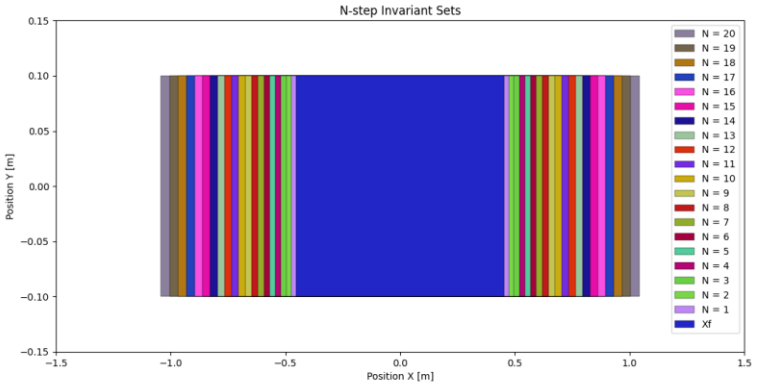

Figure 3 - N = 20 steps

As seen in figure 1-3, the distance from the origin in x-direction increases with the increasing control horizon. The y-direction is constrained between -0.1 and 0.1. In general, an increase of horizon steps *N* increases the initial set of feasible solutions.

Q2: With an increased boundary of the controller the set of feasible solutions increases. Once again, only in x-direction. When increasing the boundaries of u, the invariance set increases proportionally in x-direction until the limit of ±1.2 is reached.

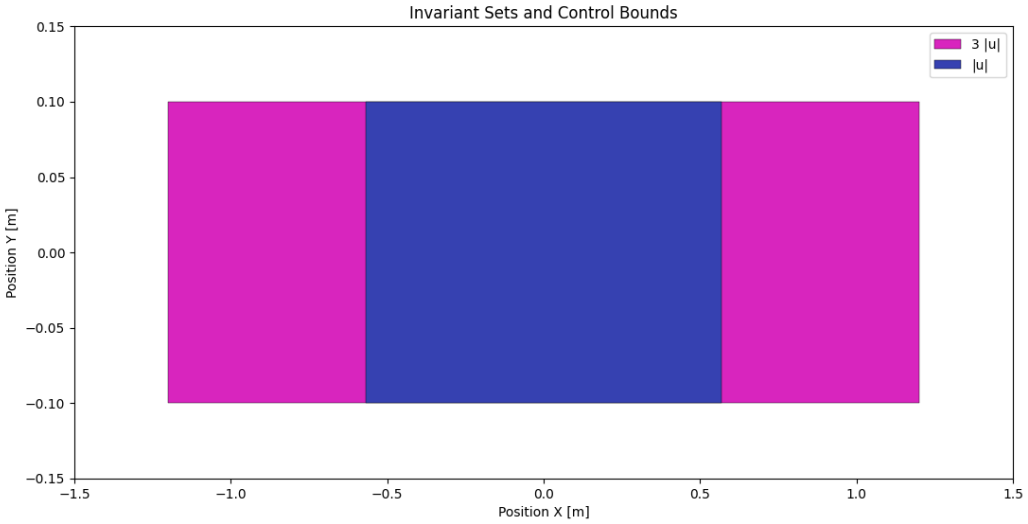

Figure 4 - Varying of control constraints

Q3:

1. The state constraints are set by appending upper and lower bound as inequality constraints separately for every timestep t. Regarding the upper bound, xub, it is set as:

$$-\infty \leq x_t \leq x_{ub}$$

And the lower bound is set as:

$$x_{lb} \leq x_t \leq \infty$$

```
# 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 5 - State constraints

2. The object function is formulated by summing the running cost over every timestep and adding the terminal cost once at the end. The entries of the cost function are the difference between x_t and x0_ref, the weighting matrices Q and R as well as the control signal u_t. Regarding the terminal cost the difference between the last state and the reference state is penalized by matrix P.

```
    # 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 6 - Object function

3. The terminal constraints are included by adding inequality constraints for the final state x_N. The inequality constraints upper bound is set by using the offsets of the polytope and the lower is set to minus infinity. The following inequality holds:

$$-\infty \leq H_N x \leq h_b$$

```
# 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 7 - Terminal constraints

4. The variable param_s concatenates vertically the vectors, x0, x0_ref and u0 in form of a single input vector for the casadi solver. The three input vectors are the start values. The param_s is handed over to the solver as a part of a dictionary to formulate the nonlinear program (nlp). See figure 9.

```
# 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 8 - Declaration of param_s

```
# Build NLP Solver (can also solve QP)
nlp = dict(x=opt_var, f=obj, g=con, p=param_s)
options = {
    'ipopt.print_level': 0,
    'print_time': False,
    'verbose': False,
    'expand': True
}
```

Figure 9 – Dictionary for the solver

5. The variable x0 is used to receive the first control input u0 to control the system. See figure 10.

```
_, u_pred = self.solve_mpc(x0)

return u_pred[0]
```

Figure 10 - Usage of x0

Q4:

The values for energy use, as well as the position and attitude integral errors obtained when running the simulation with and without expanding the boundaries of u_{lim} can be found in table 1.

Table 1

	u_{lim}	$3u_{\text{lim}}$	Percentual change
Energy use	156.2	160.5	+2.75%
Position integral error	3.94	3.55	-9.9%
Attitude integral error	0.9	0.88	-2.2%

From the table it can be seen that an increase of the boundaries of u_{lim} with a factor of 3 results in an increase of energy usage while the integral errors decrease. This is reasonable since a larger set of control inputs has the possibility to decrease the errors further on a limited horizon compared to a small set. But it comes with the cost of being more energy consuming. The percentual changes show that error reduction is higher than the increase of used energy.

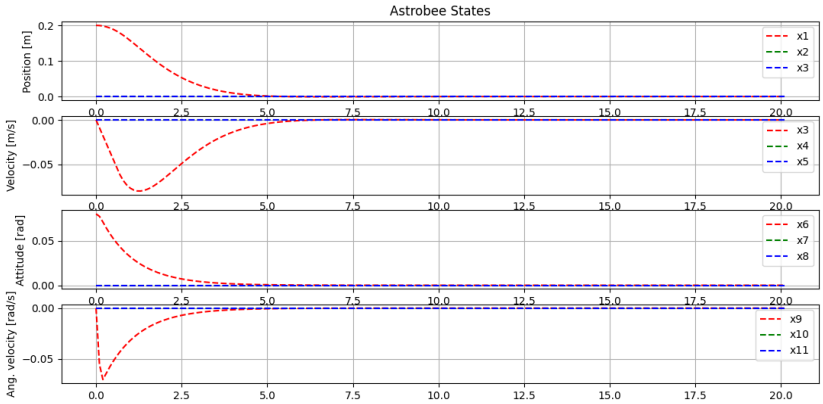

Figure 11 - Simulation with u_lim

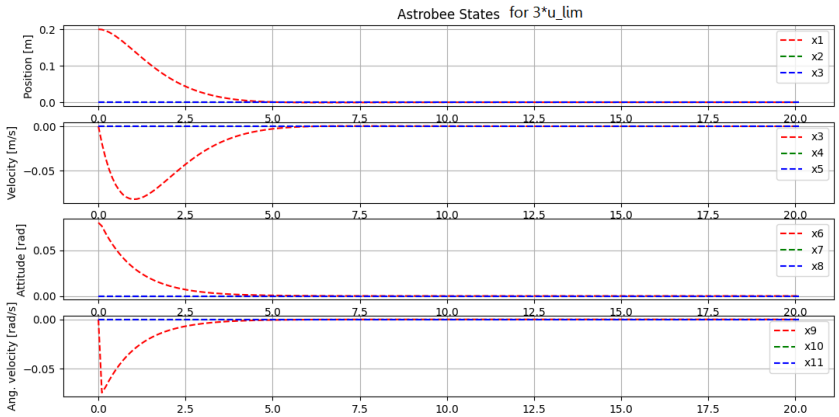

*Figure 12 - Simulation with 3*u_lim*

From the simulations it can also be seen that the position is reached slightly faster when the boundary of the input signal is increased by a factor of 3.

Q5:

When setting the terminal set to $X = \{\mathbf{0}\}$ the problem becomes infeasible. By comparing the invariant set obtained by $X = \{\mathbf{0}\}$ (figure 13) with the invariant set in Q1 (figure 14), it has decreased significantly which makes the initial state appear outside the set. As a result, there is no solution for the MPC problem.

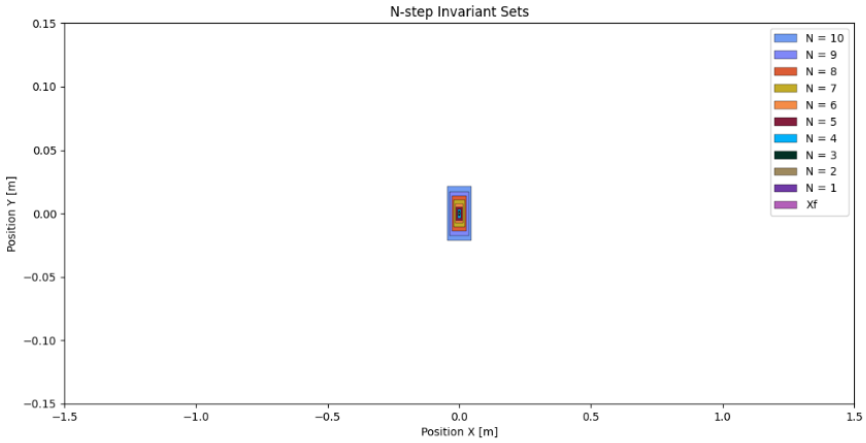

Figure 13 - Invariant set with terminal set as zero

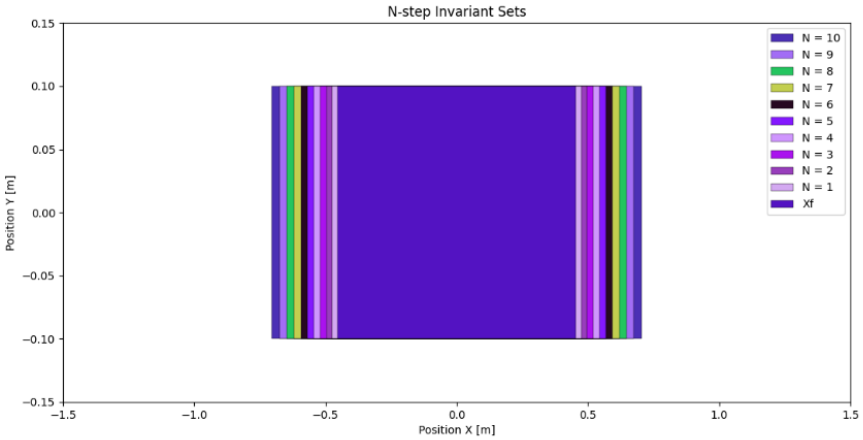

Figure 14 - invariant set from Q1

Q6: The problem becomes feasible again when changing the horizon length from $N = 10$ to $N = 50$. But the computational time is increased. With $N = 10$ the computational time is approximately 18 seconds. This can be compared to the computational time of $N = 50$ which is 44 seconds.

Regarding the time to solve the MPC problem it took around 0.04 to 0.11 seconds for $N = 10$ to solve it. For $N = 50$ it took around 0.12 to 0.2 seconds to solve it which is an increase of approximately factor 2.

Q7:

- **By multiplying R with a factor 10**, the energy used is decreased to 100.61. This decrease is reasonable since the control penalty is increased and therefore the optimal solution is found with smaller control inputs. The integral error of the position is 5.82 and for the attitude 0.91, which is an increase in both cases. As a drawback the astrobee reaches its final position after approximately 6 seconds compared to 4.5 seconds for the standard case. It also uses a smaller force input with a maximum of 0.13 N which can be compared to 0.3 N for the standard case.
- **By adding 100 to the velocity components of Q,** the energy used is reduced to 32 with the drawback of increased positional integral error to 18 and attitude integral error of 7. The astrobee will not reach its final position within simulation time of 20 seconds. The maximum velocity is decreased $0.017\ m/s$. This can be compared to the standard case when the maximum of approximately $0.075\ m/s$.
- **By adding 100 to the positional components of Q,** the system tries to reach the desired states faster. As a result, we overshoot the goal states, e.g., by -0.008 in x_1. The energy consumption is 182.99.
- **By increasing all elements of Q with 100,** we increase the penalty on every state. Hence, the states in total are penalized more than the control input. Since the velocity is penalized higher than before, the energy consumption is decreased to 156.2. The overshot of the position states is decreased, e.g., to 0.0006 of x_1.
- **Conclusion:** Tuning the MPC problem is quite like tuning an LQR-problem in term of tuning rules and intuition. For example, by increasing the penalty on the position and attitude states the controller reaches the desired states faster with a higher tendency to overshoot. By penalizing the control inputs the energy consumption can be highly decreased at the cost of getting a slower system response.